Multi-agent deep reinforcement learning for cooperative data distribution optimization

Master Thesis

Author:
Andrey Kharitonov

Examiner and Supervisor:
Prof. Dr. rer. nat. habil. Gunter Saake

2nd Examiner:
Dr.-Ing. Sascha Bosse

Supervisor:
MSc. Gabriel Campero Durand

Magdeburg, 08.07.2020
Andrey Kharitonov

Multi-agent deep reinforcement learning for cooperative data distribution optimization

Abstract

Machine learning is becoming increasingly commonplace in various systems and operations that comprise them. Traditional approaches to machine learning are successfully employed as tools in such tasks as log and data stream analysis, automatically detecting failures and anomalies. Nowadays, there is an ongoing effort to put a solution, based on machine learning, in direct control of complex systems, instead of machine learning algorithms serving only as tools for data analysis. Deep reinforcement learning is a subset of machine learning algorithms aimed at learning how to directly control complex systems such as self-driving cars, drones, or various software systems and subsystems, without oversight. It is an active and broad area of research.

The application of deep reinforcement learning in systems consisting of multiple autonomous components capable of semi-independent actions is a more recent direction that is not yet deeply explored. In this work, we review the state of multi-agent deep reinforcement learning (MDRL) and the multi-agent environments employed by researchers. We experimentally evaluate the influence of selected hyperparameters on the performance of an off-the-shelf MDRL algorithm, called MADDPG, implemented in Ray RLlib, with a reference multi-agent environment.

We design and implement a data distribution environment devised for integration with MDRL frameworks and stand-alone algorithms. Our multi-agent data distribution environment (MDDE) is designed to be easily compatible with third party MDRL implementations. MDDE utilizes a real-world in-memory database for data storage, and can be deployed in a distributed infrastructure to capture the effects of real-world infrastructure on MDRL agents, controlling the allocation of data fragments. Additionally, MDDE also includes an estimation mechanism for core metrics, such as throughput and data fragment access, facilitating the use of our environment with limited computational resources.

Within our evaluation setup we were unable to discover a configuration for the combination of selected off-the-shelf MDRL algorithms and MDDE that is capable of solving a data distribution environment where the correctness of actions is highly nuanced and depends fully on the current state. Agents were not able to construct meaningful policies based on the collected observations. Explicitly eliminating incorrect actions at every step improved the resulted evaluation metrics, but it did not result in the emergence of meaningful policies.

By outlining the challenges faced and by making MDDE extensible, configurable, and by sharing it with the community, we lay the groundwork for further research in the field.
Acknowledgements

I want to take this opportunity to thank Gabriel Campero Durand for guiding me through the field of deep reinforcement learning and for helping me develop the wholesome appreciation for this topic that has kept me inspired throughout this project.

At the same time, my work with the DBSE research group and this thesis would not have been possible without Prof. Dr. nat. habil. Gunter Saake, so I also wish to extend my gratitude to him and his team.

I would also want to express my gratitude to Dr.-Ing. Sascha Bosse for providing valuable feedback on the scope, framing, and structure of this thesis.
Statement of Authorship

I hereby declare that I am the sole author of this master thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

__________________________________________
Signature

_______________________________
Place, Date
## Contents

### List of Figures

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>65</td>
</tr>
</tbody>
</table>

### 1 Introduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>8</td>
</tr>
</tbody>
</table>

### 2 Background

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2</td>
<td>12</td>
</tr>
<tr>
<td>2.1.3</td>
<td>13</td>
</tr>
<tr>
<td>2.1.4</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2</td>
<td>17</td>
</tr>
<tr>
<td>2.2.3</td>
<td>22</td>
</tr>
<tr>
<td>2.3</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1</td>
<td>26</td>
</tr>
<tr>
<td>2.3.2</td>
<td>31</td>
</tr>
<tr>
<td>2.4</td>
<td>32</td>
</tr>
<tr>
<td>2.5</td>
<td>34</td>
</tr>
<tr>
<td>2.5.1</td>
<td>34</td>
</tr>
<tr>
<td>2.5.2</td>
<td>35</td>
</tr>
<tr>
<td>2.5.3</td>
<td>36</td>
</tr>
</tbody>
</table>

### 3 Design of the environment

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>39</td>
</tr>
<tr>
<td>3.2.1</td>
<td>40</td>
</tr>
<tr>
<td>3.2.2</td>
<td>47</td>
</tr>
<tr>
<td>3.2.3</td>
<td>53</td>
</tr>
<tr>
<td>3.2.4</td>
<td>56</td>
</tr>
</tbody>
</table>

### 4 Implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>65</td>
</tr>
<tr>
<td>4.1.1</td>
<td>65</td>
</tr>
<tr>
<td>4.1.2</td>
<td>66</td>
</tr>
<tr>
<td>4.1.3</td>
<td>68</td>
</tr>
</tbody>
</table>
## 5 Evaluation and Results

5.1 Evaluation of hyperparameters influence ........................................... 77
5.2 Benchmark estimation evaluation ....................................................... 81
5.2.1 Preliminary observations ............................................................. 82
5.2.2 Metrics estimation evaluation results ............................................. 84
5.2.3 Summary ....................................................................................... 86
5.3 Environment with cooperative scenario evaluation .............................. 89
5.3.1 MADDPG ................................................................................... 90
5.3.2 MAAC ....................................................................................... 98
5.3.3 Summary .................................................................................... 100

## 6 Conclusion and Future Work

6.1 Conclusion ....................................................................................... 101
6.2 Threats to validity ........................................................................... 103
6.3 Future Work ................................................................................. 104

## Bibliography

A Appendices .......................................................................................... 119
A. Verification of the selected off-the-shelf algorithms implementation. .... 119
A.1 MADDPG ..................................................................................... 119
A.2 MAAC ......................................................................................... 122
A.3 DQN ............................................................................................. 126
B. Plots: MADDPG hyper-parameter tuning for MPE. ......................... 129
B.1 Learning rate ............................................................................... 129
B.2 Replay buffer .............................................................................. 136
B.3 Discount factor ........................................................................... 142
B.4 DNN weights adjustment ............................................................. 148
B.5 Number of steps between updates ................................................ 154
C. YCSB workloads ............................................................................. 160
D. Environment with cooperative scenario evaluation .......................... 161
D.1 MADDPG (without action mask) agents behavior in MDDE ......... 161
D.2 MAAC agents behavior in MDDE .................................................. 162
D.3 Plots: MADDPG with Action-mask: failed actions ....................... 162
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Reinforcement learning (Markov decision process).</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Multi-agent reinforcement learning - fully observable environment.</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Multi-agent reinforcement learning - partially observable environment.</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Identified research directions.</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Concept of the multi-agent data distribution environment</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>MDDE Components</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Sorted sequence $G$ reflecting current allocation of fragments.</td>
<td>50</td>
</tr>
<tr>
<td>3.4</td>
<td>Sequence of RL framework interaction with the environment</td>
<td>54</td>
</tr>
<tr>
<td>3.5</td>
<td>Environment initialization</td>
<td>55</td>
</tr>
<tr>
<td>3.6</td>
<td>Learner step sequence</td>
<td>57</td>
</tr>
<tr>
<td>3.7</td>
<td>Benchmark sequence (example with two threads)</td>
<td>58</td>
</tr>
<tr>
<td>3.8</td>
<td>Example observation space (for agent $w_0$)</td>
<td>61</td>
</tr>
<tr>
<td>4.1</td>
<td>YCSB core workload main components</td>
<td>68</td>
</tr>
<tr>
<td>4.2</td>
<td>YCSB: Distributions</td>
<td>70</td>
</tr>
<tr>
<td>4.3</td>
<td>MADDPG actor's DNN action mask principle</td>
<td>74</td>
</tr>
<tr>
<td>5.1</td>
<td>Allocation change in the 'starving' edge-case</td>
<td>85</td>
</tr>
<tr>
<td>5.2</td>
<td>Throughput dynamic comparison in the 'starving' test case</td>
<td>86</td>
</tr>
<tr>
<td>5.3</td>
<td>Allocation change in the 'greedy' edge-case</td>
<td>87</td>
</tr>
<tr>
<td>5.4</td>
<td>Throughput dynamic comparison in the 'greedy' test case</td>
<td>88</td>
</tr>
<tr>
<td>A.1</td>
<td>MADDPG: MPE - Physical deception</td>
<td>120</td>
</tr>
<tr>
<td>A.2</td>
<td>MADDPG: MPE - Covert communication</td>
<td>121</td>
</tr>
<tr>
<td>A.3</td>
<td>MADDPG: MPE - Keep-away</td>
<td>121</td>
</tr>
<tr>
<td>A.4</td>
<td>MADDPG: MPE - Cooperative communication</td>
<td>122</td>
</tr>
<tr>
<td>A.5</td>
<td>MADDPG: MPE - Cooperative navigation</td>
<td>122</td>
</tr>
<tr>
<td>A.6</td>
<td>MADDPG: MPE - Predator-prey</td>
<td>122</td>
</tr>
<tr>
<td>A.7</td>
<td>MAAC: MPE - Physical deception</td>
<td>123</td>
</tr>
<tr>
<td>A.8</td>
<td>MAAC: MPE - Covert communication</td>
<td>124</td>
</tr>
<tr>
<td>A.9</td>
<td>MAAC: MPE - Keep-away</td>
<td>124</td>
</tr>
<tr>
<td>A.10</td>
<td>MAAC: MPE - Cooperative communication</td>
<td>124</td>
</tr>
<tr>
<td>A.11</td>
<td>MAAC: MPE - Cooperative navigation</td>
<td>125</td>
</tr>
<tr>
<td>A.12</td>
<td>MAAC: MPE - Predator-prey</td>
<td>125</td>
</tr>
<tr>
<td>A.13</td>
<td>DQN: MPE - Physical deception</td>
<td>126</td>
</tr>
<tr>
<td>A.14</td>
<td>DQN: MPE - Covert communication</td>
<td>127</td>
</tr>
<tr>
<td>A.15</td>
<td>DQN: MPE - Keep-away</td>
<td>127</td>
</tr>
<tr>
<td>A.16</td>
<td>DQN: MPE - Cooperative communication</td>
<td>127</td>
</tr>
<tr>
<td>A.17</td>
<td>DQN: MPE - Cooperative navigation</td>
<td>128</td>
</tr>
<tr>
<td>A.18</td>
<td>DQN: MPE - Predator-prey</td>
<td>128</td>
</tr>
</tbody>
</table>
1 Introduction

We begin this work describing the motivation and the focus this thesis in Section 1.1. In Section 1.2 we list the main contributions of our work. Finally, in Section 1.3 we outline the overall structure of this thesis.

1.1 Motivation

Deep reinforcement learning (DRL) has recently gained significant attention among the broader research community after the success demonstrated by DeepMind’s AlphaGo [SHM+16], and more recently, the multi-agent AlphaStar [ACT19]. Reinforcement learning (RL) is a subset of machine learning. The purpose of RL [Sut88] and DRL differs from common machine learning techniques employed for data analysis. RL is not a technique for classification, clustering, or regression based on some dataset. RL is instead a technique where an algorithm instance, referred to as an agent, learns to interact with the world around it, to achieve a certain goal within this world. The world, or environment, can be the real world, for example in robotics, or a virtual world, such as a video game. The idea behind RL can be best summarized as learn by doing. The agent quite literally executes some actions in its surrounding environment and judges the outcome. Based on the outcome, it decides if taking the action, in the specific state of its surrounding environment, led to beneficial results or not.

DRL, is a subset of RL that relies on deep neural networks (DNN) and is often regarded as the closest thing we have now to actual artificial intelligence. Humans, especially in their infancy, learn certain things purely by trial and error, like walking or holding a pen. Similarly, DRL-based agents learn to perform certain tasks by trial and error while each failure or success is reflected in the internal artificial neural network, which is also in principle designed to mimic neural networks in human brain.

As shown [MKS+15] by researchers in the field, DRL-based algorithms can solve a wide range of complex problems that would otherwise require human intervention. For example, DRL in database management systems (DBMS) is an active research area in which we can distinguish two major directions according to DBMS components where DRL is applied: query optimizers and storage engines.

Query optimization through replacing heuristic cost models with a DRL approach for the select, project, and join operations was proposed in a recent work [KYG+18]. For the selected case study, the agent is initially trained with data generated by a traditional heuristic query optimization algorithm. Later the deployed agent receives feedback, which is judged as positive or negative, from the execution of queries and adjusts its behavior accordingly. Another published work, which is concerned with join order optimization,
presents ReJOIN [MP18a] algorithm. ReJOIN has also been generalized to a query optimizer named Neo [MP18b] by the same authors. The core goal of such generalization is the creation of a self-tuning general-purpose query optimizer. A different angle of research in the application of DRL in query engines is cardinality estimation [OBGK18]. The proposed approach views a query as an environment where states are sub-queries that are processed by the query engine in succession.

A significant effort in DBMS storage engine research direction goes into alleviating the need for time-consuming manual storage configuration. Correct index selection requires analysis and understanding of the DBMS workload. A proposed DRL-based concept called NoDBA [SSD18] has been designed to automatically recommend attributes for the creation of optimal secondary indexes according to the workload.

Another important consideration is data partitioning. Researchers propose a DRL-based Partition advisor [HBR19]. Its purpose is to automatically determine whether to replicate a table or partition it over one of the attributes. The advisor makes its decision based on the workload that is learned by a DRL agent.

Alongside DRL, a field of multi-agent DRL (MDRL) [NNN19] also started to gain traction in areas where multiple intelligent agents naturally exist and must interact with the same environment and with each other. Interaction might have a collaborative nature, where multiple agents learn to cooperate to achieve some common goal [GEK17].

Distributed data storage systems constantly grow in complexity, volumes of data, and requirements for access efficiency. The optimization of data allocation is an important challenge that affects not only end-user experience but also the costs of running such a system. In addition, distributed systems are affected by outside factors, such as network speed and latency, speed of data retrieval, quality of connections. These challenges can be mitigated by sophisticated heuristics and machine learning-based algorithms but often require complex configuration and maintenance by the expert users of such systems.

We believe that the application of MDRL in this domain potentially opens a way to efficient, self-driving distributed data storage systems, where the distributed storage nodes, are capable of learning semi-independently to achieve the best individual performance while at the same time not disregarding the system as a whole. Such systems would not require experts for setting them up and maintain them. Instead these systems would adapt to the infrastructure, workloads, and potential shifts in both. The multi-agent concept is natural for distributed systems where nodes are represented as individual agents and capable of distributing the computational load of complex optimization processes. Therefore, considering the ongoing research of single-agent DRL in DBMS, the emerging popularity of MDRL, and the distributed nature of modern data storage systems, we believe there is an opportunity to bring MDRL research to the DBMS domain.

To the best of our knowledge, there is no published work that attempts to apply MDRL to data distribution optimization in DBMS. We believe the idea of collaboration between distributed nodes to achieve the optimal distribution of data can be explored anew by employing modern MDRL techniques instead of the heuristic rules. However, in order to even begin with the exploration of such concepts, we need an evaluation framework, a training environment capable of supporting scenarios where multiple agents could shuffle...
data records around, while at the same time being able to receive feedback on the quality of the distribution.

In this work we make our goal to create such an environment where an MDRL-based algorithm could be employed to facilitate a scenario where multiple agents collaborate in order to achieve the optimal allocation of the data records across distributed data nodes. Our intention is to design an MDRL environment that can be deployed in real-world infrastructure to capture its effects on the efficiency of data retrieval from a distributed system. Additionally, we design the environment in the way such that it can be also deployed within the confines of a single system and used efficiently for simplifying its use for preliminary hypothesis testing or debugging.

1.2 Main contributions

1. We review existing publicly available open source multi-agent environments suitable for MDRL research. We report on the common design points of these environments, and the most notable individual features. Upon the results of this review we base the core concepts of our own data distribution environment design.

2. We present our design and publicly available implementation\(^1\) of a multi-agent oriented environment designed for scenarios where MDRL agents cooperate to achieve an optimal data distribution. We evaluate the scenario where MDRL agents directly manipulating the data fragments allocation. By making the implementation publicly available, we encourage extension and modification of our own design, and addition of custom scenarios, for further research in the area.

3. We conduct a series of experiments where two off-the-shelf MDRL algorithms, MAD-DPG [LWT+17] and MAAC [IS18], attempt to solve our environment scenario. We report the results and performance metrics. We discuss the observations and perceived reasons behind the behavior of the agents within the scenario and the environment as a whole.

4. We outline future research directions in the area of data distribution with MDRL, while taking into account the evaluation results and discovered design shortcomings of our own designed environment. We believe these directions for further research are promising improvements upon our work and potentially result in better performance of MDRL in data distribution, compared to the results we have observed in our evaluation.

\(^1\)https://github.com/akharitonov/mdde
1.3 Thesis structure

The rest of the thesis is structured as follows:

- In Chapter 2, we present the explanation of the fundamental concepts required for the understanding of the problem solved within this work. Specifically, we explain the theory behind deep reinforcement learning as a concept applicable to both single-agent and multi-agent scenarios. We examine an existing survey on general aspects of MDRL research and conduct our own brief literature review identifying specific directions in the current research. Additionally, we survey existing open source reference multi-agent environments and their design.

- Chapter 3 formulates the specific research goals of this work, and provides a detailed explanation of the conceptual design for a multi-agent environment suitable for cooperative data distribution MDRL scenarios. We discuss the design and intended use workflow. In this section we also present the idea and the concept of the data distribution scenario where agents have the capacity to directly control allocation of each fragment and its replicas, without a centralized manager.

- Within Chapter 4, we briefly discuss the implementation specifics of the key components in our environment architecture, conceptual design for which we presented in Chapter 3. We present the description of the integration between our environment and an MDRL framework, as well as a standalone algorithm.

- In Chapter 5, we provide comprehensive evaluation results of the MDRL environment, that we designed and implemented within the confines of this work, for the cooperative data distribution scenario.

- Finally, with Chapter 6 we conclude this work and outline possible directions for future research.
2 Background

In this section, we provide the theoretical background relevant to our prototypical design for a multi-agent data distribution environment, and our experimental evaluation. We introduce the reader into the multi-agent deep reinforcement learning research field and providing evidence for the relevance of our work. The rest of the section is organized as follows:

- In Section 2.1, we present a short introduction to the fundamental ideas behind deep reinforcement learning and the classes of algorithms that are relevant to current research. This subsection introduces the relevant terminology as well as basic concepts on necessary to efficiently read the remainder of work.

- In Section 2.2, we discuss the core concepts of MDRL. Specifically, we provide the general formal description of the common approaches as well as common challenges. In Subsection 2.2.2, we demonstrate the relevance and variance of ongoing MDRL research by reviewing the current state of the field. We take a look into a general MDRL categorization of an existing survey and propose our own categorization, taking into account the current directions of in MDRL based on specific solved problems. Additionally, we provide a brief overview of the current specific real world applications.

- Section 2.3 is dedicated to the review and discussion of the specific existing multi-agent environment implementations typically used in the process of evaluation of the published MDRL research. We discuss the design and commonalities among these environments.

- In order to illustrate the importance of the data distribution optimization challenge, as well as to provide a brief introduction into the field, within Section 2.4 we briefly examine the state of research in adaptive data distribution by looking into a recent survey.

- Finally, in Section 2.5, we focus on the discussion of the ideas behind the selected off-the-shelf MDRL algorithms, which we apply in the evaluation of our own work. Additionally, we perform a verification of the implementations of the selected algorithms against an open source reference multi-agent environment, and report the results.

2.1 Deep reinforcement learning

Traditionally RL is described in terms of a Markov decision process (MDP) [PT87], the general concept of which is depicted in Figure 2.1. Components of MDP are traditionally
represented as a tuple $\langle S, A, T, R \rangle$. Where $S$ is a set of the environment states (observation spaces), $A$ is a set of agent’s actions (action space), $T$ represents state transition probability $T(s_{t+1}|s_t, a_t)$ at a time step $t+1$ given the state $s$ and an action $a$ from the time step $t$, and $R$ is a reward function $R(s_t, a_t, s_{t+1})$.

The reward function is a mathematical function that outputs a numerical value measuring the quality of actions taken by the agent judging by the state of the environment to which these actions led. It is imperative to design a reward function that would reflect the intended behavior of the agent. For example, in a hypothetical situation where we design an agent which we task with fixing typos in long texts. We give the agent actions such as insert a letter and delete a letter. Then we base the reward solely on the number of typos found in the text. Now it is easy to imagine that the agent easily learns that there are no typos if there is no text and just removes everything instead fixing typos. This example is simplistic and rather obvious, but in more complex scenarios, such as controlling data records distribution, pitfalls of designing a correct reward function might not be as obvious.

Essentially, the objective of an RL agent in MDP is to find an optimal relation between states and actions. This relation is the optimal policy $\pi$, that maximizes the reward within only the current state $s_t$ or cumulative reward of consecutive actions, depending on the goals of MDP application. The end goal of the agent learning process is devising an optimal policy, which is essentially a rule determining which specific action an agent should take under a specific set of conditions, or state of the environment.

In this work, we focus on RL algorithms using deep neural networks (DNN) [LBH15], hence the name deep reinforcement learning. DNN can be seen as a series of interconnected layers consisting of nodes (neurons), that contain a pre-defined mathematical function, with adjustable parameters, called weights. The purpose of such a structure is to map ingested data to a specific output value. Currently, researchers in the field often differentiate two major categories of DRL methods: Q-learning based algorithms and policy gradient (PG) based algorithms.

### 2.1.1 Q-Learning

The idea of Q-learning [WD92] is to keep track and refine a table of estimated discounted rewards (discount factor is discussed later in Subsection 2.1.4), denoted as $Q$, hence the name Q-learning, in a state $s \in S$ after taking action $a \in A$. Essentially, the goal is to produce a table of estimates of $Q(s, a)$, the rewards for every action and state combination.
Values in the table are computed and updated at every iteration until convergence, which leads to high memory and computational overhead. Then the agent can simply search through the $Q$ table for $Q(s, a)$ pairs where $s$ corresponds the current state and pick the action $a$ where the estimated $Q$ value is the highest. Of course, these $Q$ values are updated as the agents acts and accumulates experiences.

The deep Q-network algorithm (DQN)[MKS+15], developed by DeepMind, is based on Q-Learning and uses DNNs to approximate $Q(s,a)$, which allows reducing the computational and memory requirements of Q-learning. The deep neural network in this case maps states $s$ to the tuples $\langle a, Q \rangle$ for each $a$ in the action space of the agent. This algorithm demonstrated performance improvement over the state of the art at the time and, in many cases, has served as a basis for multiple later publications.

The authors of the original DQN paper introduced this approach as a technique that was able to achieve human-level performance in a set of simple classic Atari video games, that authors used as a benchmark for complex learning problems. The notable difference of this work from the previous publications is the fact that the introduced algorithm was capable of learning directly from the video output of the video games, and not from a specially created set of features. The latter would be required in the original Q-Learning because otherwise, it would be required to store and update a $Q$ table for a scenario where a single pixel difference between frames would mean a new state. This would be computationally too expensive. Using DNNs for estimating $Q$ values alleviates this issue.

More specifically, DQN brings two significant modifications into Q-learning. The purpose of both of the key DQN innovations is to facilitate a more stable learning process of the agent when $Q$ values are estimated by a DNN. The first addition to Q-Learning principle is "experience replay", alternatively called "replay buffer". Previously gained experience is stored in a buffer. Previous experience at time step $t$ is defined as $e_t = (s_t, a_t, r_t, s_{t+1})$, where $s_t$, $a_t$, $r_t$ are the state at the time point $t$, action taken at the state and the reward received respectively, $s_{t+1}$ is the next observed state. The purpose of the replay buffer is shown by the authors as an efficient mechanism to train the underlying DNN in scenarios with a high cost of gaining experience from the environment. Past experiences $e_t$ are stored in the buffer. Content of this buffer is sampled at random, and the samples are reused in the DNN training process.

In addition to the replay buffer, authors introduce a mechanism that copies and saves the current trained DNN as a static copy. However, in case there is already a stored DNN, it is not overridden completely but its weights adjusted to a degree that is controlled by a specific hyperparameter (discussed later in this chapter). Then the agent performs learning on the original network, while the copy stays at rest for a number of time-steps. This mechanism works in iterations, also called learning rounds. The optimization process uses output parameters from both of these networks, where the saved network copy makes predictions of the $Q$ value for the next state. The use of two networks is shown by the authors to be beneficial specifically when Q-Learning is used with DNN for estimating $Q$ values.
2.1.2 Policy gradients

Second major class of DRL algorithms is policy gradients [SMSM00] (PG). The major difference from the Q-learning based algorithms is that PG algorithms are optimizing policies directly, as opposed to \(Q\) (value) function optimization. PG algorithms attempt to model action probabilities directly. Meaning, instead of estimating a table of action-state-rewards, PG estimates the probabilities of the agent to take the action at a specific state. In the case of DRL, such estimation is specifically done using a DNN, that maps states \(s\) to actions \(a\). Also, PG doesn’t generally use replay buffer or any other comparable technique to store previous experiences. The policy is learned by the agent directly, and then the training data is discarded. Additionally, unlike Q-Learning based algorithms, PG is capable of functioning in continuous action spaces\(^1\). PG can be employed in scenarios where actions do not need to be executed in discrete steps (e.g., pressing on-off switch), but instead continuous (e.g., car’s steering wheel turn).

There is a number of different policy gradient approaches, here, as with Q-learning, we provide enough background to give sufficient intuition for the further discussion of the aspects relevant to this work. Detailed justification and mathematical foundation can be found in the referenced source literature [SML+15]. Core principle of the policy gradient suffers from the problem of inability to make too steep adjustments in the parameters, meaning the learning rate (hyperparameter discussed in Subsection 2.1.4) has to be low as otherwise there is a chance that the policy adjustment goes too far and this can lead to an action that is extremely inappropriate and the policy might never recover. However, a low learning rate might require an unnecessarily large number of iterations to learn the policy, which is also a problem if collecting experiences in the environment is expensive. The strive to overcome this challenge has led to an algorithm called Trust Region Policy Optimizations (TRPO) [SLA+15]. TRPO policy learning process is meant to keep the policy changes within the acceptable bounds, while not sacrificing learning speed by assigning a low learning rate. It is done by introducing a so-called trust region, which is a constraint added to the optimization objective, a range within the policy updates is guaranteed to be "trustworthy" and such that will not lead to dramatic policy degradation. Basically, this constraint makes sure that the new policy is not moving too far away from the old one and stays within an acceptable range of parameters. For example, if a TRPO-based agent controls a car’s steering wheel, it is guaranteed that the agent does not learn to suddenly turn the wheel so much that it instantly throws the car off the road resulting in a state and a learned policy that are not recoverable.

However, TRPO has shortcomings outlined by researchers [SWD+17] in the field. Specifically, while achieving its goal of keeping the policy adjustments contained, it adds a significant computational overhead because the trusted region is a consideration added on top of the core PG optimization calculations. This specific problem led to the development of Proximal Policy Optimization (PPO) [SWD+17] algorithm, which is based on the same premise as TRPO but the constraint is incorporated within the optimization calculation itself, which results in a smaller computational overhead.

\(^1\)Q-learning can work in continuous spaces with some discretization, but this is, arguably, not the intended use of the approach.
2.1.3 Actor-critic

So far, we have described value (Q-Learning based), and policy-based (PG based) approaches as separate branches of the reinforcement learning algorithms. However, it is possible to combine these within a single algorithm that is designed using an actor-critic architecture [SLH+14]. In this architecture, there are two key components in the agent: an actor and a critic. An example of such an algorithm in DRL is Deep Deterministic Policy Gradient (DDPG) [LHP+15]. The actor component of the agent employs policy-based, meaning PG, methods and is directly responsible for the actions taken. The critic component works in parallel to the actor but uses a value-based, meaning Q-learning, approach to measure how good are the choices made by the actor. Essentially, critic operates a DNN that estimates Q-values, while the actor’s DNN directly recommends the action that the agent should take. Actor updates its policy taking into account the critics output. Similarly to DQN, critics in DDPG rely on replay buffer. One of the key advantages of this approach is the adaptation of a Q-Learning based approach to continuous action spaces. Actor-critic as a concept is especially interesting in the context of MDRL as some prominent state-of-the-art MDRL algorithms are based on such architecture.

2.1.4 Hyperparameters

There is a number of parameters that are common to the reinforcement learning algorithms and directly affect the learning efficiency. These are commonly referred to as hyperparameters.

To control the agent’s behavior strategy of obtaining the reward, a discount factor $\gamma \in [0,1)$ is added to the MDP tuple. Thus, the MDP tuple is extended to $\langle S, A, T, R, \gamma \rangle$. The discount factor is a hyper-parameter applied to a reward value obtained at every step of MDP. Values of the discount factor that are closer to 1 would make agents strive towards the higher reward in the future while paying less importance to the immediate reward. It means that the agent might take a suboptimal action in the current state $s_t$ if the policy dictates that taking this action would eventually lead to the transition of the environment at some state $s_{t+n}$ where the reward value exceeds what the agent can gain in the immediate state. Contrary to that, discount factor $\gamma$ closer to 0 will make the agent pay higher importance to maximize immediate reward values, instead of necessarily 'planning' for the future.

A hyperparameter that is typically adjustable in DRL algorithms is learning rate. It is a straightforward parameter that controls the magnitude of adjustments made in the values directly affecting the agent’s policy $\pi$. As the name suggests, the higher the value, the faster agent overrides its own policies according to the new observations. Low learning rate essentially cases agent to be cautious about changes in own learned behavior. For example, if the agent’s actions led to a previously unseen state $s_t$, that might be a rare or faulty occurrence, low learning rate would not allow the agent to override its policy immediately, for the previous state-action combination $(s_{t-1}, a_{t-1})$. Instead, the policy will be effectively changed if this new $s_t$ would become a consistent result for $(s_{t-1}, a_{t-1})$. 

13
Some hyperparameters specific to approaches based on DQN. Specifically, as mentioned before, DQN operates two neural networks per agent. One online that is adjusted at every step, one at rest. Periodically the online network is synchronized to the one that is saved at rest. Hyperparameter $\tau$ (tau) specifies the magnitude of weights adjustment in the saved neural network of the agent. In other words, higher $\tau$ values would cause updates in the neural network, which is stored at rest in the DQN-based algorithm, to be more radical whenever networks are synchronized, while lower $\tau$ leads to more gradual updates. The value should be selected depending on the desired behavior of the agent. If the intention for the agent to quickly adjust to the changes in the way the environment behaves, higher $\tau$ would be preferable. $\tau = 1$ would cause the entire saved network to be overridden by the online network. Such behavior is preferred in case of expected sudden and long-lasting changes in the environment. Otherwise, lower $\tau$ makes the agent’s saved DNN adjustments more gradual, and by proxy slows down agent’s adaptation to the sudden changes. It is useful in case of expected sudden but short-lived alterations in the environment or other agents’ behavior.

In the case of DQN and any other algorithm employing a replay buffer, the length of the replay buffer is a hyperparameter. With each agent’s step, new samples are added to the replay buffer. Making an infinite replay buffer is memory expensive and might also be impractical from the learning efficiency standpoint in cases older experiences are no longer relevant. Therefore, it is typical to define a fixed-length replay buffer functioning as a circular buffer. It allows discarding the oldest, possibly irrelevant experiences, as well as to carefully control the amount of memory used by the agent.

Another hyperparameter relates to multi-step learning [HS19], often referred to as simply n-step. This hyperparameter, with range $[1, \infty)$ of integer values, which control the frequency of policy updates for the agent. Or in simple terms, how many steps the agent takes before the accumulated rewards within this number of steps are evaluated and the policy is updated accordingly.

This is not an exhaustive list, but the mentioned hyperparameters can typically be found in various algorithms and their implementations. Each specific algorithms can possess a number of own unique hyperparameters. The effects of the same hyperparameters, when applied to different algorithms, will differ. Hyperparameters are generally tuned for each algorithm-environment combination individually.

### 2.2 Multi-agent deep reinforcement learning

MDRL is an active but relatively new area of research. MDRL is a subset of multi-agent reinforcement learning (MARL), with the core differentiating trait of employing DRL. The core difference of MDRL from DRL is the fact that instead of training only one agent that observes the environment and then acts, now a few separate agents exist, which are capable of learning and affecting the environment. In the case of MARL systems, it is possible that each agent is also capable of observing not the full state of the environment but only a fraction of it. A recently published framework for multi-agent systems [TS18] provides a comprehensive basis for understanding the peculiarities of multi-agent systems. In this framework, authors define multi-agent learning as the study of multi-agent systems in which one or more of the autonomous entities improve through experience.
The framework and the definition described by the authors are generic for multi-agent systems. Therefore, it provides a solid basis for the discussion of MDRL within our own work.

The aforementioned MARL survey distinguishes the following components of a multi-agent system:

1. The environment.
2. Multiple agents, all existing in the same environment.
3. Interaction mechanism between the agents.
4. Learning mechanism (in the context of our work it is DRL-based).

The environment can be a software system or a real-world setting in the case of robotics or self-driving cars. This environment is the same for all of the agents existing in it. Agents are learning and acting entities that try to achieve a goal corresponding to their individual or collective task. Agents might be able to interact with each other through specific mechanisms.

2.2.1 Challenges

![Multi-agent reinforcement learning - fully observable environment.](image)

The existence of multiple learning agents that interact with the same environment presents new challenges. In a recent review of MDRL [NNN19], the authors identify the five critical MDRL challenges.

First, the environment is non-stationary, where multiple learning, independent agents are present. Unlike in the case with single-agent systems, multiple agents are taking actions and affecting the environment at the same time. The presence of multiple active agents leads to the situation where an agent’s policy has to be adjusted to reflect actions and changes in the policies of the other agents present in the system.

Second challenge of multi-agent systems, is partial observability, a possible condition of an MDRL system where each agent can only see a part of the environment and might
not be able to observe all of the other agents present in it. Such condition is depicted in Figure 2.3. While in fully observable scenarios, all agents see the entirety of the environment, as depicted in Figure 2.3. This means that when developing an MDRL system, it must be taken in to account that each agent might not know the true state of the entire system but can only observe a specific part of it. It also goes for observation of other agents, meaning that it is not necessary that every agent can observe every other agent and their actions. This adds an additional layer of complexity to the previously outlined non-stationarity problem as an agent must anticipate and account for the action of the agents, which it might not be able to observe directly.

Learning in multi-agent systems is one of the most prominent and often addressed challenges. Training every agent independently, while treating other agents as just part of the environment, is computationally expensive and prone to overfitting, as outlined in the survey [NNN19].

Transfer learning is mentioned in the survey as a separate challenge of MDRL. However, we believe that it is not a challenge in itself but rather a solution to the previously mentioned challenge of agents learning in MDRL. Transfer learning is one of the ways to alleviate the problem of high computational cost for training agents independently. The idea is to transfer policies, partially or in their entirety from one agent to another.

Handling continuous action spaces is, as mentioned before, challenging for Q-Learning based approaches, including DQN. Policy gradient based DRL algorithms or actor-critic [LHP+15] approaches can be used for an agent instead when required.

Additionally, it was also shown in recent publications [MLBW17], that it is challenging to build an agent that is able to generalize the learned behavior over drastically different tasks. This problem is not restricted to MDRL, but also applies to single-agent DRL. This means that building an agent which can potentially fulfill every subsystem role in, for example, DBMS system is problematic. It is hard to make the agent that optimized allocation, indexing, and select fragmentation scheme at the same time. It is better to make agents that specialize on specific narrow tasks.
2.2.2 Current state of MDRL research

A recent survey [HLKT19] provides a high level categorization of fundamental MDRL research. Presented categories are: analysis of emergent behaviors, learning communication, learning cooperation, agents modeling agents. This categorization is aimed at the development and refinement of the fundamental MDRL algorithms and behavior of the agents. Analysis of emergent behaviors is a research direction that deals with single-agent DRL agents’ behavior in multi-agent scenarios. Learning communication deals with explicit communication protocols between agents. Learning cooperation category researches algorithms dealing with cooperative scenarios where agents must work together to achieve a common goal. Agents modeling agents category is essential fundamental research solving the challenge of non-stationarity of the environment, where an agent must account for the actions of other independent agents in the same environment.

However, for the purposes of this work, we are interested in a more fine-grained MDRL specific categorization that reflects potentially useful features that can be applied in the systems reliant on MDRL, rather than fundamental algorithmic research, even if a certain idea presented by a single publication and falls outside of the categorization of provided in the mentioned survey [HLKT19]. We believe it is important to have a detailed overview for the research ideas currently explored in the research community, especially for possible future problem solving or scenario designs in our MDRL environment.

To independently assess the current state of the research in the field of MDRL, we have conducted a structured literature review, utilizing the Google Scholar database. We do not entirely discard non-peer reviewed publications, such as those published in Cornell University’s arXiv. ArXiv is the publication venue of choice for many DRL research papers from prominent industry practitioners such as Alphabet’s DeepMind2, OpenAI3, Microsoft4. We consider only fully accessible publications discussing specific MDRL related research in English. We focus on the general research directions while discarding publications concentrating on the specific application solutions within specific domains or algorithms optimization. Which means that, since we’re interested in the feature-oriented survey, we generally omit the publications concerned with iterative improvements of the existing concepts while bringing no new features to the field. Additionally, we perform a backward and forward reference search for the retrieved publications. No filtering by publication date is applied. The search query is provided in Table 2.1. Conducting our own survey, we seek supplement and validate the categorization presented in the aforementioned survey. We believe it is important to understand, that MDRL is a very active field of research with a lot of new ideas explored all the time that do not have a clear cut category.

Table 2.1: Literature search query

<table>
<thead>
<tr>
<th>Query</th>
<th>Returned</th>
<th>Relevant</th>
<th>Not in [HLKT19]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;multi-agent deep reinforcement learning&quot;</td>
<td>439</td>
<td>43</td>
<td>29</td>
</tr>
</tbody>
</table>

We present the list and relationships of identified fields of research in Figure 2.4. There is a clear and expected overlap with the previously mentioned survey [HLKT19], but we have also discovered ideas from the field of MDRL, that fall outside of its general categorization. It’s also worth noting that not all publications can be clearly put into a single category and may be covering more than one research direction. The purpose of the presented categorization is to highlight the ideas discussed in MDRL research community.

This survey also serves as a validation for the importance of our own work, which is concentrating on developing a universal evaluation environment for MDRL in the domain of data distribution optimization. A universal evaluation environment is genuinely needed when there is a variety of hypotheses to evaluate.

Learning

Design aspects that encompass the whole learning process constitute, when taken together, one of the most important research directions in MDRL and thus receives the most attention. We have identified several sub-directions in learning research.

A relevant subcategory of learning approaches is based on encouraging cooperation [WHF+18, TMK+17, YHL+20] to make several agents perform tasks in a way that results in achieving a common goal. Cooperation is especially useful in scenarios where different agents have different capabilities so that each can work a specific sub-task in which the agent is specialized. Alternatively all agents can be similar but using multiple semi-independent agents is more beneficial or makes sense, in the case of parallelization or distribution for example. In fact, it is the core premise of our own hypothesis that cooperating agents, each of which manages own data storage, cooperate with other agents to achieve the best possible data retrieval performance.
However, competitive behavior is also studied, as shown in already mentioned publications [TMK+17]. Specifically, competitive behavior is introduced in some research work, by creating an adversarial agent that seeks to help the learning process [BPS+17]. Authors propose that this is an applicable approach to facilitate learning when the complexity of the task exceeds that of the environment.

Of particular interest is learning stability. As mentioned previously, one of the key challenges of MDRL is the non-stationary nature of the environment, as well as agents’ policies, leading to the problem of learning stability. In case of agents relying on experience replay, past samples provide outdated versions on the policies of other agents and hence are not that good for training. An actor-critic-based [LWT+17] algorithm called MADDPG was proposed, intended to address this challenge. This approach is general-purpose and suitable for both competitive and cooperative scenarios. Another way to tackle the issue is a direct modification of DQN by introducing a decay mechanism in the experience replay sampling, where older samples are less likely to be chosen over the most recent ones [FNF+17]. LDQN [PTBS18] introduced a concept of leniency to DQN, where the importance of old state-action transitions decays over time, which diminishes the influence of old records in experience replay on current decisions.

To improve learning scalability, researchers in the field have proposed an improvement upon an actor-critic approach presented by MADDPG consisting of decentralized actors and a centralized critic but attempting to pay specific attention to the part of observations that at the most critical for learning. Such algorithm is called MAAC [IS18]. It is presumed that MAAC is capable of developing policies with better correlations to the specific portions of state observations than MADDPG. An algorithm called STEP [PSB+19] presents another take on centralized learning where agents are learning in centralized fashion but execute policy in a distributed manner. In STEP, decentralized planners orchestrate agents, according to the decentralized learned policies. Centralized learning approaches allow for a smaller computational load for policy and/or value approximation.

A specific case of agent learning is if the agent in question does not directly receive the reward feedback. One of the ways to tackle the issue is through imitation learning. Imitation learning refers to the scenario, where the agent learns by observing and imitating other agents in both cooperative [SRSE18] and competitive [LZZ+20] scenarios.

In most cases, approaches for learning consider some form of cooperation, communication, or observability between agents, which means that agents know about other agents and do not learn independently. However, that is not always a suitable scenario. An actor-critic based approach [VVL18] has been proposed to specifically deal with cases where both the environment and other agents are only partially observable, serving as an efficient way to learn agents independently.

The ability to model other agents or human experts can be employed in both cooperative and competitive scenarios. AlphaGo [SHM+16], while not a multi-agent approach, is interesting in the context of this survey. The first stage of learning in AlphaGo was supervised learning of DNN that would predict an expert player’s action. However, while the strategy of modeling opponents to anticipate their behavior is explored in several publications [FCAS+18, ZMH+18, RDSF18], it is not the only reason why MDRL algorithms attempt to use it. For example, $M^3RL$ [ST18] algorithm introduces a manager agent that
serves as a controller distributing tasks to other agents to achieve the optimal performance of the system. The manager has to be able to know the ability of any other agent to solve a given task. Since agents aren’t static and learn continuously, the manager has to be able to model other agents to infer their capabilities. We believe such approaches can potentially be employed for developing a system where multiple different components are driven by MDRL, while cooperating within the same overall system and also anticipating each others behavior.

Of course, as already mentioned, not only competitive or control scenarios can make use of the techniques where agents model other agents. In purely cooperative scenarios, agents can model agents that are in the same "team", and are striving to achieve the same goal by working together. Teammate agents can model each other to anticipate [MZXG19] each other’s actions without necessarily using direct communication channels for coordination.

In single and multi-agent DRL, an agent is learning through the process of trial and error while interacting with the environment. Sometimes, a human, another system, or a process might attempt to interrupt learning. This interruption should not affect the agent’s learning process negatively and is thus referred to as safe interruptibility in literature [GHM+17]. Intentional interruptions of the agent’s actions and learning should be identified by the agent correctly. Agent learning an interruption as a state can lead to a situation when an agent attempts to avoid such interruptions, whether it is possible or not, and learns otherwise incorrect policies. This challenge becomes more complex in multi-agent systems. Such interruptions should be interpreted correctly not only by the agent being interrupted but also by other agents within the same environment, which proves to be challenging, especially in the case when agents are fully independent and don’t communicate explicitly.

The idea of reusing learned policies from one agent in another agent has attracted the attention of researchers as a way to tackle the challenge of the learning process, which is typically being time-consuming and computationally expensive. A recent survey [DSC19] on transfer learning was published outlining a few key points. It is necessary to determine the source of the knowledge received by the agent, as well as the way this source is defined in the first place. Then there is the question of what to transfer, which tasks or parts of the tasks are similar between the source and the target agents. Later the transfer can be enacted in multiple ways, including transferring learned policies, value functions, modes, experiences (e.g., replay buffer in DQN-based agents). Authors also show that in the case of DRL-based approaches, such as DQN, special care for experience replay transfer is required. Experience reuse can negatively impact learning instead of accelerating it, and it is challenging to recover from this. The decay techniques mentioned previously in this work can help tackle the challenges of experience replay transfer.

Hierarchical learning is currently an active area of research. The key idea behind hierarchical learning is to break down complex tasks into simple low-level ones that are later sequenced together to achieve the ultimate high-level goal. The agents must be trained accordingly. General challenges of multi-agent DRL accompany the application of this principle as well. But there are some solutions presented. For example, the challenges associated with the non-stationary environment [THL+18], and the challenge of identifying the global optimal solution taking into account an increasing number of communicating agents [KSHH17].
The communication between agents is essential to achieve coordination, naturally it is an active area of research in MDRL. Learning communication protocols instead of pre-defining them manually has been proposed by a number of papers [SLR19, HYZW18, WZW18, MGN17, SsF16, DGR18]. A somewhat different approach to learning communication is presented by communication through actions [TZW18] instead of explicit protocols, where agents create models of other agents that allow them to reason about these agents’ intentions. Communication hierarchies [KXLW17] can help tackle the challenge of scalability in MDRL. Another subcategory of communications research is the emergence of a language between agents [MA17, HT17, GCK19, CLL18, LCS20], with the language being defined as the means used by the agents to communicate concepts to achieve a joint goal. However, as the research of emergent language gains traction, it has been pointed out by some researchers [LFB19] that certain conclusions drawn in this research direction might be misleading.

Communication between agents can also be a way to tackle the problems arising from the variability in the number of existing agents [WS18], which we believe can be especially useful in a distributed data management system powered by MDRL to achieve elasticity. A specific challenge mentioned by researchers [WS18] is that in partially observable environments, each agent might have its policy affected by noise, which leads to choosing sub-optimal actions. However, a communication layer is provided for the scenarios when an agent decides that its observations are insufficient and requests information from other agents, and additionally share information that is accurate with others. Here the learning process becomes the process of not only learning to interact with the environment but also learning when and which data to share with other agents [KM18].

Researches explore novel ways to facilitate communication between agents in MDRL applications to improve the scalability and ability of dissimilar agents to cooperate. For example, researchers propose [NNN20] agent communication based not on protocols or explicit data sharing but instead on generating visual maps, which are then processed by each agent’s neural network. The idea is rooted in the assumption that DNNs can be efficient at discerning meaning from the visual representation of information.

Macra-actions

Apart from the aforementioned aspects of learning and communication, one of the active areas of research is taking on the concept of macro-actions [KKZ18, AKKH19]. Such actions have a strategic nature and take prolonged periods to execute, in contrast to fine-grained micro-actions that take no more than one time-step within the system to execute. The difficulty of implementing macro-actions in MDRL is that every agent can choose to execute such action, and different actions can take a different amount of time to finish, which in turn results in a challenge of coordination. Hence this constitutes an area of research within MDRL.
Enforcing regulation

Safety, combining robustness and specification requirements, is naturally important in MDRL. It requires rethinking, when compared to single-agent methods, because safety needs to take into account the emergent behavior of the system of agents. In some scenarios, especially in a multi-agent system consisting of independent agents with no overarching coordination system, a set of regulations might be required. For example, if agents are competing for the same pool of resources, it is required to prevent a single agent from consuming the entire pool. Without global coordination, some agents might attempt to act in accordance only with their self-interest. One proposed method of enforcing regulation is a boycott system [SCWL19]. The proposed mechanisms discourage regulation violations by making other agents develop their policies such that they disregard the interests of the agent that is detected to be breaking the regulation, which in turn leads to a decrease in reward for that non-compliant agent. If applied to a distributed database system, such concept can be an alternative or a supporting tool for enforcing operation constraints of the system.

Neural network architecture’s behavior

To better understand the nature of challenges presented by the learning in MDRL-based systems, researchers conducted a study [COSW19] to investigate various neural network architectures behaviors, specifically in regards to representation of action-value functions. One of the key conclusions of this work is that different DNN architectures are suitable for different subsets of problems, there is no all-around best solution. As a result, for every chosen task, the impact of the architecture needs to be considered by researchers. This means that DNN architecture becomes a hyperparameter.

2.2.3 Applications

To date there have been many publications describing applications of MDRL. For illustration, we mention some real world applications.

Communications and networking are a natural field for the application of multi-agent systems due to the inherent existence of multiple actors within the network. One of the most recent publications is a framework for improving the quality of transmission for cellular users with a proposed algorithm is based on the actor-critic principle [LG19]. A DQN based solution was proposed by researchers [ZHW18] to ensure optimal allocation of users to the cell towers, while ensuring the balance between cost efficiency and quality of service.

One of the most popular applications for reinforcement learning is autonomous vehicles. It is not surprising to see that MDRL is finding applications in this area as well. Specifically, in facilitating collaboration between unmanned aerial vehicles [YL18], while basing the work on Q-learning. Another published work is focusing on unmanned surface vehicles [HZW19].

Researchers and practitioners explore the possibilities of employing MDLR in applications bridging multiple areas. For example, trajectory optimization of autonomous vehicles
playing a role of cellular network base stations, where the goal is a quality of service improvement for the network [WZW\textsuperscript{+}20, WZWS20].

Efficient transportation is important for many industries and people so there is always a search for efficiency improvements in this area. This search includes attempts of MDRL application. There are works that try to solve multiple related problems, such as optimization of large transport fleet management [LZXZ18], optimization of online matching for ride-sourcing services [KXYY19], a solution for facilitating ride sharing solution [LJY\textsuperscript{+}19]. All three mentioned solutions utilize Q-Learning and actor-critic based algorithms.

Another area of interest, which is tightly related to the two previously mentioned, is a vehicle to vehicle communication [YLJ18]. The proposed solution is based on Q-learning and facilitates agents to select the optimal communication band locally to satisfy latency constrains, while at the same time making sure there is no interference to other communication systems.

The important problem of the real world is energy efficiency. A few publications proposed application of MDRL to tackle different aspects of this complex problem. Application of IoT for scheduling of power grid load [ZKX\textsuperscript{+}19], facilitation of energy sharing in household communities designed to operate fully on renewable energy [PD18].

Another publication proposes an adaptation of MDRL to support the microservices architecture of software development to facilitate the ability of such architecture to adapt to the changes in operating environment [Mag19].

This is brief and in no way an exhaustive overview of the potential applications of MDRL in real world. As we can see, there is already a range of diverse problems that can potentially be addressed by MDRL. It stands to reason that introduction of MDRL to database systems and specifically to data allocation tasks is a reasonable direction for research.

2.3 Environment

The environment is one of the key components in the reinforcement learning process. On the conceptual level, the environment represents the world which the agents observe and where they act. On a practical level, this environment can be a physical system or some virtual world, such as a video game or a software system. As long as it is possible to facilitate the Markov decision process, where agents are capable of interacting with the environment, there are no technical constraints on the applicability of the environment implementation to the reinforcement learning.

However, it is impractical to build a physical robot or utilize a complex real world system for research and development. In fact even attempting to integrate DRL into a real-world database system would be challenging, because the codebase of these systems is simply not designed for such use. The research process of developing a new algorithm or tuning hyperparameters is iterative, often with many different sets of experiments running in parallel. Even a viability test for a new algorithm requires some environment to run. A real world system would be too expensive for such cases, especially if the
developed solution is concerned with robotics or any physical hardware, such as self-driving cars. Even integration with a real distributed software system might be challenging and impractical for experiments and research. Such systems designed for distributed real world use are often packed with sophisticated heuristics, have a complicated codebase, and generally not designed for research purposes.

In such cases, a synthetic environment is utilized. Depending on the task at hand, a researcher can choose to use a simple generic environment or one that represents the system for which the DRL algorithm is developed and tuned. Such environments provide a number of advantages for the research. They tend to be easy to utilize and usually don not require any special hardware. Open source solutions provide a degree of customization that allows to model actions and observations as simple or as complex as required by the hypothesis being tested.

Synthetic environments based on models can be parameterized to simulate a variety of conditions, such as added noise, variability in the number of agents, limited communication, sudden interruptions, or any other condition designed to evaluate the MDRL research directions and approaches discussed in Subsection 2.2.2. It also allows to abstract from the real world system implementations and quickly model different scenarios. It’s also possible to build a synthetic environment that is based on a real world infrastructure but designed specifically for research. Then the environment is not a mathematical model but a system that is affected by real world conditions instead of simulating them.

In principle, an environment for a reinforcement learning algorithm has to facilitate Markov decision process, as described in Section 2.1. Therefore, three basic functions must be exposed. First, the environment implementation should allow the retrieval of the observation that accurately and sufficiently represents the current state of the environment. Second, an agent must be able to make a step in the environment. In other words, the agent must be able to perform an action changing the state of the environment. Lastly, the agent must be able to evaluate the correctness of the taken step, meaning the environment must return the reward value as feedback. Additionally, a function typically provided by the implementations of such environments is reset, which reverts the environment to its initial state. This initial state can either be statically defined or have some randomization.

There are multiple examples of such environments designed for single-agent scenarios. In this section, we mention a few most well known of such examples. However, we do not provide an exhaustive survey for single-agent DRL environments, as it is not the focus of our work.

Perhaps the most well known is the OpenAI’s Gym [BCP+16]. It is an actively maintained open source toolkit that is aimed at developing environments for single-agent reinforcement learning algorithms. It comes with a few pre-defined environments of varying complexity and nature. Amongst these, there is a Gym wrapper for old Atari arcade video games, as well as control of a virtual robotic manipulator. There are also simpler, more abstract, environments such as ‘classic control’, which includes a car that needs to be pushed up the mountain by an agent, amongst other similar examples. Gym also serves as a basis for a large variety of third-party environments, which are not developed or maintained by OpenAI. These include board, games, flight control simulations, trading market simulations, etc.
Third-party environments and DRL frameworks often use open AI’s Gym data structures for observation and action spaces. It is done to ensure compatibility between the two. Therefore it is important to understand what these data structures are. We provide a brief description of these data structures in Table 2.2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Intention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Box</td>
<td>Observation</td>
<td>Gym.Box is one or more dimensional array with values of the same type lying within a predefined interval.</td>
</tr>
<tr>
<td>MultiBinary</td>
<td>Observation</td>
<td>Gym.MultiBinary is an arbitrary length list of binary values.</td>
</tr>
<tr>
<td>Discrete</td>
<td>Action</td>
<td>Gym.Discrete is a single number. Intended for action spaces. Gym.Discrete = 2 would mean that the action space has two actions. For example, a button that can be either pressed or not pressed, both of these would be discrete actions.</td>
</tr>
<tr>
<td>MultiDiscrete</td>
<td>Action</td>
<td>Gym.Multi-discrete is a list of Gym.Discrete. For example, a controller with multiple buttons.</td>
</tr>
<tr>
<td>Tuple</td>
<td>Container</td>
<td>Gym.Tuple is a tuple structure that can contain other action or observation spaces</td>
</tr>
<tr>
<td>Dict</td>
<td>Container</td>
<td>Gym.Dict is a map structure that can contain other action or observation spaces</td>
</tr>
</tbody>
</table>

Another notable example of a single-agent oriented reinforcement learning environment is Deepmind Lab [BLT+16]. It differs from Gym in the sense that it is not meant to be a foundation for a large variety of environment types. Instead, it focuses strictly on 3D environments. Deepmind Lab is built upon an open source 3D video game engine. An agent observes the 3D image as pixels and can navigate within the game’s 3D world while attempting to learn how to solve simple tasks. Suitable for development and benchmarking of single-agent DRL algorithms, especially the ones designed for processing raw visual input as an observation space. Alternatively, researches can utilize ViZDoom [KWR+16], an environment very similar to Deepmind Lab, however based on a different open source 3D video game. In both cases, environments are configurable, ship with a number of pre-defined scenarios, and support the implementation of custom scenarios.

There is an apparent abundance of environments providing game-based environments to DRL agents, particularly if we consider the ones simulating a 3D world. However, it is worth noting that there are non-game-like 3D environments. An example of such is MINOS [SCD+17]. It is a Gym compatible environment focused on the simulation of complex indoor scenarios. A notable difference from the previously mentioned 3D environments is that unlike these, it is not built with the assumption that the agent observation space is strictly a visual one. MINOS is capable of simulating complex observation spaces, with or without added noise. For example, a simulated GPS assists an agent. Complex sensors can also be added to the environment, each with varying properties, including resolution and orientation.

---

While games are a popular basis in the research-oriented DRL environments, there are also examples of such that instead focus on real-world software systems. Perhaps the most notable example is Park \cite{MNN19}, which is a Gym compatible framework for developing environments simulating real-world system interfaces. Amongst such environments, provided in Park, researchers can find SQL query optimization, database indexing, server load balancing, content delivery network (CDN) memory caching, and others. Park is an important project attempting to breach the gap between DRL as a scientific concept and real-world systems. It’s important to note that Park is capable of working with real world systems and not only simulations of such. For example, by default it contains a scenario based on a real world database engine instance that serves as part of the environment.

CARLA \cite{DRC17} is an interesting environment example. It provides a rich physical simulation for autonomous driving. It contains a library of road layouts, different types of vehicles and road elements, configurable weather conditions, rich visualizations, as well as C++ and Python-based interfaces. CARLA is focused on the single-agent control, DRL isn’t the sole focus, however.

While, the list of the single-agent oriented environments mentioned above is by far not exhaustive, we believe these are representative examples for active projects in the field and can serve as the testing and development ground for a variety of single-agent DRL research.

### 2.3.1 Multi-agent environments

All of the previously discussed environments are explicitly aimed at single-agent reinforcement learning research. That means, these environments do not support scenarios where more than one learning agent observe and act in the environment independently of others. It is important to differentiate between single-agent and multi-agent capable environments in order to sufficiently evaluate the challenges of learning in multi-agent systems mentioned in Subsection 2.2.1, and advance the research directions mentioned in Subsection 2.2.2. In order to support MDRL, at minimum, the environment should provide interfaces where each agent independently observes the environment, performs an action in parallel to other agents, and receives a reward that is returned as a response for the specific action of the specific agent.

Naturally, such environments exist and pose a particular interest for us, within the context of this work. To better understand commonalities and differences between these environments, we conduct a structured literature review based on the query provided in Table 2.3 and use the Cornell University Arxiv library as a source. We search for the publicly available open source implementations of the environments designed for multi-agent reinforcement learning research. We do not consider implementations that are tightly integrated with a particular algorithm and can not be reused for further research without significant alterations. Additionally, we do not consider environments that rely heavily on the components without publicly fully available source code, such as commercial video games (e.g., Starcraft, Minecraft).

We ignore all environments based on commercial video games. The reason behind this decision is the non-reproducibility of the results as games like Starcraft are constantly...
updated, and older versions can not be easily obtained or used. Additionally, as commercial products, these have an upfront cost and often rely on complex copy protection systems that may hinder any attempts to reproduce the results provided in the publications.

Table 2.3: Multi-agent environments search query

<table>
<thead>
<tr>
<th>Query</th>
<th>Returned</th>
<th>Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;multi-agent AND reinforcement learning AND environment&quot;</td>
<td>254</td>
<td>9</td>
</tr>
</tbody>
</table>

Multi-agent particle environment

We put particular emphasis on the OpenAI’s multi-agent particle environment (MPE) [LWT+17]. It is an open source, easily customizable environment available for all researchers. While it is relatively simplistic and not actively developed, it is still commonly used in MDRL research and publications as a generic reference environment. OpenAI has designed the environment specifically for multi-agent research, as opposed to their Gym based single-agent oriented environments.

Conceptually MPE represents an abstract two-dimensional world with basic simulated physics, where a variable number of simple agents (particles) can move and interact with each other. By design, the environment allows the definition of custom scenarios, which in turn define properties of the world and the agents. MPE is capable of simulating both cooperative and competitive scenarios. To this end, agents are categorized as either "good" or "adversary". Only "good" agents are present in a cooperative scenario, while a competitive scenario requires the presence of one or more "adversaries".

The simulated world in MPE is not necessarily just an empty field where agents roam around. Scenarios can define a variable number of so-called landmarks. These are special entities that are placed in the specified coordinates in the world, have an assigned color, and are potentially observable by the agents. Depending on the simulated scenario, landmarks can serve different functions. For example, these can be either an obstacle or a goal for agents to find. Landmarks are immovable.

Each agent, which is also referred to as a particle in the paper that originally introduced MPE [MA17], has a few properties. It can be marked as "movable", which means that it can either be "pushed" by other agents or if it can move on its own by making movement actions. An agent can also be set to be able to collide with other agents or landmarks, instead of passing through or overlapping with them. Similarly to landmarks, each agent also has an assigned color, potentially observable by the agents present in the same environment. Since MPE simulates basic physics, agents have certain basic physical properties such as mass and maximal speed.

MPE features scenarios where agents possess the means of facilitating explicit communication between agents, where agents can exchange messages between each other. In this case, each agent can be a message producer (speaker), a consumer (listener), or both. Speaker agents, in addition to movement actions, also possess communication specific action, which is used to generate a message that would be consumed by the listener agents.
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Good</th>
<th>Adversaries</th>
<th>Landmarks</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cooperative communication</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>✓</td>
</tr>
<tr>
<td>Cooperative navigation</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>✓</td>
</tr>
<tr>
<td>Keep-away</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>✗</td>
</tr>
<tr>
<td>Physical deception</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>✗</td>
</tr>
<tr>
<td>Predator-prey</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>✗</td>
</tr>
<tr>
<td>Covert communication</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>✓</td>
</tr>
</tbody>
</table>

Listener agents consume the messages from speaker agents as part of the observation space.

MPE comes with six scenarios, which are described in Table 2.4. These scenarios contain a variable number of "good" agents, "adversarial" agents, and "landmarks" (immovable objects). Amongst these scenarios, two are fully cooperative (only "good" agents), rest are competitive or mixed (both cooperation and competition), and have at least one adversary defined. Two scenarios focus on learning of communication, one cooperatively another is mixed. Numbers of agents and landmarks are adjustable, and default values are not specifically mentioned in the corresponding publications [LWT+17]. Agent and landmarks quantities mentioned in the Table 2.4 are based on the default values in the OpenAI’s MPE source code\(^6\) published alongside the paper.

Action spaces in the standard MPE scenarios are very simplistic. Cooperative navigation, Keep-away, Physical deception, and Predator-prey scenarios offer agents with simple action spaces consisting out of five actions controlling movement of the agents in a 2D space. Covert communication does not permit any movement, but agents have four actions, each where each action corresponds to a bit in a bit array encoding a color. In Cooperative communication, agents have different action spaces, one has thee actions each of which "communicate" a different color code to the second agent, which in turn has five movement in 2D space actions. In all of these scenarios except Covert communication, reward directly depends on the relative positions of the agents in the world.

We have put a lot of emphasis on MPE because, at the moment, it is the most popular reference multi-agent environment among the researchers. The original publication presenting MPE is the most cited paper among all of the environments we’ve discovered. However, it is not the only multi-agent environment that we have discovered during our survey.

**Game-like environments**

Like in the case of the single-agent oriented environments, games serve as a source of inspiration for multi-agent training environments. However, where single-agent environments might rely on open-source versions of existing video games, it becomes problematic in multi-agent scenarios. Traditionally video games are designed to be played by a player providing input and observing the visual output within the confines of a single game client. Even if the game itself has multiplayer capabilities, such as multiple game clients

connecting from different workstations over a network connection, emulating such complicated setup for an MDRL environment is not practical. As a result, we see games that are specifically created not for humans to play but specifically for reinforcement learning agents to train.

A good example of such an environment is Neural MMO [SDIM19], developed initially as part of a research project within OpenAI. The environment is inspired by MMORPGs (Massively Multiplayer Online Role-Playing Games). This environment presents a world where a large number of independent agents learn to navigate and interact with the world, as well as interacting with the other agents. Agents can have varying actions, and generally, each has only partial world observability around itself. The environment can be used in research studying emerging behaviors of large agent populations in a shared world.

Another example is OpenSpiel [LLL+19] developed by Deepmind. OpenSpiel was designed to serve as a framework and a collection of reference multi-agent environments. OpenAI Gym heavily influenced the design of this framework. The authors stated goal was to create an extensible collection of reference game-based environments all under the same unified interface for the learning agents, and as a result providing creating a multi-agent equivalent to Gym, specifically the Gym’s Atari-based environments. OpenSpiel is radically different from Neural MMO, as it is providing a large number of environments varied in nature, which are mostly focusing on small population multi-agent games (e.g. chess, go, dice, poker, etc.) or more abstract scenarios such as cooperative box-pushing.

Google Research team has published an open source environment simulating a football video game [KRS+19]. While technically not offering a large variety of environments like OpenSpiel, the football environment is suitable for simulation of different learning scenarios for reinforcement learning algorithms. Football is a competitive team game where there is one team cooperates within itself to compete against the other similar team. This already offers an opportunity to simulate a cooperative-competitive scenario. Additionally, authors have supplied the implementation with a rule-based bot, which, for example, can control one of the teams in order to simulate a fully cooperative scenario. The rule-based bot can also control some of the team players if the simulated scenario requires a smaller population of actual learning MDRL agents.

**Simulations**

Simulation and modeling of real world systems are especially crucial in multi-agent scenarios. To physically build a number of agents, programmed to learn to interact with each other and their surroundings, is not practical due to the sheer amount of effort and expenses this would require, especially in the early stages of research. The issue of dealing with physical systems is also becoming apparent in cases where the research process requires the ability to scale the number of agents quickly.

CityFlow [ZFL+19] is an environment simulating car traffic flow. It is suitable for MDRL as it allows learning agents to control the speed of individual vehicles and/or traffic lights. It supports the simulation of traffic flow based on arbitrary synthetic parameters as well as based on datasets containing real world data. The core of the environment is developed
in C++ while providing a Python-based interface facilitating integration with the popular MDRL frameworks.

The idea of air traffic control has also attracted the attention of researchers in the field of MDRL. For this purpose, researchers provided an MDRL extension [BW19] to an air traffic simulator named BlueSky [HE16], which was originally developed in TU Delft. The resulted project is a simulation of air traffic control aimed at multi-agent scenarios such as altitude deconflicting of multiple aircraft. It provides a Python-based interface, which can be utilized by an MDRL framework to control the position and heading of a simulated aircraft. Basic weather condition simulation is also possible.

As we mentioned in Subsection 2.2.3, autonomous driving vehicles is one of the application areas of MDRL attracting the interest of the researchers and the industry practitioners. Naturally, an environment meant specifically for multi-agent scenarios in autonomous driving was published by the researchers. MACAD-Gym [Pal19] was developed by Microsoft AI team based upon the aforementioned single-agent CARLA environment, also used as a single-agent DRL environment. MACAD-Gym extends CARLA in the way such makes it suitable for MDRL scenarios with a variety of adjustable settings. It is capable of operating with the scenarios where all of the learning agents have the same action space and observation spaces, meaning all are the same type of a vehicle. Alternatively, it supports agents that are completely dissimilar in their capabilities and goals. The environment is also suitable for research with the implicit communication between the agents. As the name suggests, MACAD-Gym is compatible with Gym. It uses data structures provided within OpenAI’s Gym for observation spaces and action spaces interfaces, which facilitates integration with the reinforcement learning frameworks.

Previously, in this section, we have mentioned a single-agent oriented environment for indoor 3D navigation named MINOS. There is a multi-agent capable environment with the similar intent and capabilities, it is called HoME: a Household Multimodal Environment [BPA+17]. Similarly to MINOS [SCD+17], HoME is a Gym compatible environment allowing to simulate navigation in a closed space environment such as a house and provide observations to the agents based on the variety of sensorial input. HoME provides basic physics simulations for interaction between agents and the environment itself. There is also an ability to facilitate implicit communication between the agents in cooperative scenarios.

Additional notable examples

Additionally, we would like to mention an SDK that falls outside of our search criteria due to heavy reliance on a proprietary 3D graphics engine but presenting interesting capabilities and design decisions. Additionally, at the moment of writing, it is possible to obtain older versions of the 3D engine in case these are needed to reproduce the results presented in the publications using this environment. Unity ML-Agents [JBV+18] is an SDK allowing the creation of reinforcement learning environments for scenarios, which require modeling of an arbitrary 3D world where arbitrary agents act and interact. ML-agents provide several built-in sample environments and designed to be easily extensible. The latter feature is exploited by another project called Arena [SWL+19], which is based on
ML-Agents and aims to create a generic MDRL evaluation platform, much like OpenAI’s Gym is for single-agent DRL.

The most notable feature of the ML-Agents SDK is its architecture. By design, there is a clear separation of concerns where reinforcement learning algorithms, scenarios, and agent definitions are made with a common interpreted language, Python. At the same time, the logic concerning integration with the 3D engine and parallelization handled within C# code. With this non-monolithic architecture, the developers of the SDK provide researchers within the field of reinforcement learning with the ability to prototype and model their hypothesis using the language prevalent in this field of research. At the same time, using an inherently more suitable platform for handling concurrency and interoperability with Unity 3D engine. While in principle it is not unique for this specific environment, the clean design of the loosely connected via internal, well defined, API components allows for a great flexibility of experimental design and codebase maintainability. Essentially, it is a sound design that provides both: convenience for the researchers, just using the existing SDK capabilities, and flexibility for the engineers extending the range of tasks that can be handled by ML-Agents.

Another example excluded from the main survey because it depends on a commercial physics engine is OpenAI’s Multiagent emergence environments [BKM19], a set of environments simulating hide-and-seek game with two opposing teams of agents. One team is hiding, and another is seeking, while both can interact with certain objects within the environment, such as moving a ramp to get over the wall. Authors have published the environment as part of the research aimed at discovering possible unexpected strategies of solving given tasks by the agents within the environments of variable complexity.

2.3.2 Summary

It is clear that there is a variety of multi-agent scenarios allowing researchers to verify the viability of different algorithms and their parametrization in a range of possible scenarios and applications. We have reviewed the technical implementation of the mentioned environments while paying attention specifically to the program interfaces provided for integration with MDRL frameworks, and there are worth noting observations.

All of the reviewed multi-agent environments ship with pre-defined scenarios and permit the addition of custom scenarios. Scenarios typically define the size and composition of the environment (e.g. size of the area where agents can move), number of agents and their properties, reward functions.

Multi-agent particle environment (MPE) designed to be compatible with Gym and provides easy to use interfaces, which is one of the factors behind its popularity in the role of a reference environment in MDRL research. MPE provides two basic functions required for integration with an MDRL framework: observation retrieval, step function accepting actions from agents and returning the rewards.

All of the mentioned game-based and game-like environments were specifically developed in a way that provides a high level of interoperability with most MDRL frameworks as well by clearly defined exposed functions designed specifically to facilitate the Markov decision
process. The same can be said about MACAD-Gym from the simulation environment. BlueSky simulation logic is controlled with Python-based plugins, which can easily be integrated with an MDRL framework by following the examples provided in the MDRL extension of the project. However, CityFlow and HoME provide all necessary control interfaces for agents within the environment, but a layer interfacing an MDRL framework with the environment, meaning step functions with retrieval of the reward and the state, must be programmed additionally.

It is also worth noting the reset function provided by all of the Gym compatible multi-agent environments and CityFlow. The explicit reset function is missing only in HoME. The function is not part of the Markov decision process but is extremely useful for the learning process. Its purpose is to revert or re-initialize the environment. After execution of the reset function, depending on the implementation of the scenario and parameterization, one of two things is expected to happen, the environment is either reverted to its initial state as it was before any agent’s actions, or it is initialized anew with a random state. It is a convenience function that allows staging repeating learning episodes for the agents. Each episode typically consists out of a pre-defined number of steps or can be terminated early if the simulated scenario can be 'solved', meaning it has a specific terminal state, the goal that must be reached by the agents, after which any further actions become pointless.

Additionally, environments that are designed to be compatible with OpenAI’s Gym data structures and interfaces are convenient to use most of the time. Still, dependency on Gym might also be detrimental in certain situations. The issue with Gym is relatively high volatility of its codebase. Meaning, functions present in one version of Gym might get removed completely or have their signatures changed in a later release. This poses a problem when both the environment and the MDRL framework or an algorithm depend on Gym but two different and incompatible versions. This results in the need to introduce changes into the codebase of either one of these. Such Gym codebase volatility can be observed affecting even OpenAI’s own MPE, where certain functions were backported from older versions of Gym to make the environment compatible with more recent versions of Gym.

### 2.4 Data allocation management

A recent survey presenting a taxonomy of fragmentation and allocation techniques in databases [NA18], amongst others, provides a comprehensive overview of the dynamic allocation techniques. Such methods for allocation of fragments within a database system, define algorithms for transmission and replication, within distributed database systems, according to the workload. These allocation methods are meant to remove the need for manual allocation configuration, which is especially useful in scenarios where the workload is unknown or shifting.

The aforementioned survey presents a variety of proposed approaches based on heuristic and metaheuristic algorithms, such as genetic algorithms. These are designed for consistently shifting workloads with varying target conditions. The authors of the survey clearly distinguish between replicating and non-replicating algorithms. Replicating algorithms control distribution by creating copies of the fragments on different nodes, while
non-replicating only move fragments between nodes. This is an important distinction as replication increases storage costs for the data fragments, which is not always a viable option. In such a case, the algorithm must confidently move the fragment to the node it is most requested or where the overall cost of access from different nodes will be optimal.

The authors of the survey summarize the factors which are taken into account when deciding on whether fragments should re-allocated or replicated. These also determine the applicability of the algorithms for different usage scenarios. Specifically, algorithms can focus on the locality of access and attempt to move or replicate the fragment to the node where it is most requested. Time of access, transmission costs between nodes, as well as query costs, are also taken into account by some of the presented algorithms. In some cases, however, rarely, algorithms can take into account constraints such as site constraints for specific fragments that mandate these to be present within the specific node.

The authors of the aforementioned survey point out that, while the dynamic allocation algorithms strive to reduce the cost or time for accessing data, it is important to remember that re-allocation and replication of data incur certain costs as well.

The transmission cost of moving or replicating fragments between nodes must be considered not only as a target of data access optimization but also during the re-allocation process. Transferring large fragments too often may result in a high network traffic overhead hindering the normal database operations. It is especially the case when the fragment cannot be transferred directly from the source to the destination nodes. In this case, the shortest path algorithm can be employed, which, however, might result in increased computational overhead instead.

Storage cost is also a consideration for the algorithms. The general approach, among many, is to maintain access counters or other structures containing access statistics for the fragments. As a result, the more statics an algorithm maintains per fragment, the higher will be the storage cost. A short comprehensive explanation of the relative difference between different algorithms is provided in the paper for the Performance optimality enhancement algorithm [AAM14], which is mentioned in the survey [NA18].

It is also worth mentioning that the fragmentation of data takes a large portion of the ongoing research, and there is often an evident separation of concerns between fragmentation and allocation algorithms. However, there are also allocation algorithms that perform both dynamic fragmentation and allocation based on access statistics.

A notable work that is not present in the survey above is NashDB [MPSG18] designed for optimization of read-only OLAP access. It supports varying numbers of nodes and query prioritization. It is a particularly interesting concept that is encompassing fragmentation and allocation. The notable difference between this work and most of the previous is the fact that when making decisions for fragmentation and re-allocation or replication, it is driven not by the raw access statistics but attempts to simulate an economic game. Within this game, each fragment has an estimated value. The value is estimated via a heuristic mechanism every time a query is executed, or DBMS performs data re-distribution. This fragment value used directly in the decision making of whether a fragment should be replicated or not. The simple economic principle would drive the value up for the fragments in high demand and make nodes, that do not have the fragment,
"want" to obtain them for "selling" later to a query. Similarly, if a particular fragment is no longer popular but replicated in many nodes, it loses value as a common but rarely sought for commodity. Nodes will attempt to release storage space by removing low-value fragments.

2.5 Selected MDRL algorithms

The goal of this work is an attempt to promote cooperation between agents, each of which manages a data node. Especially in the context of data distribution, we need a mechanism, which would allow agents to account for the actions of other agents efficiently. For example, if we have two agents, both have a replica of the same data record. Both agents might choose to remove this replica at the next step. Such behavior is detrimental for the learning efficiency and unless external constraints are set, dangerous for the safety of the data record, which might be removed completely. Therefore, we hypothesize that the best performance should be expected from an algorithm where an agent explicitly keeps track of the other agents’ policies and attempts to predict their actions.

2.5.1 Multi-agent deep deterministic policy gradient

Multi-agent deep deterministic policy gradient (MADDPG) [LWT+17] is one of the most well known MDRL algorithms. MADDPG is an actor-critic algorithm, which draws its idea from a single-agent DDPG algorithm, which we mention earlier in Subsection 2.1.3.

The primary focus of the algorithm is dealing with one of the main challenges of multi-agent systems, which is environment non-stationarity, previously mentioned in Subsection 2.2.1. The proposed approach to solving the challenge is twofold. The agents’ actors each individually choose an action with their own respective action and observation spaces. At the same time, each actor has a paired critic that has access to experiences of all agents, judges the quality of the action picked and influences the actor’s policy adjustments. This means the critics, unlike actors, have access to observations of all agents, instead only their own potentially limited observation space. Additionally, critics infer other agents’ policies via sampling their past actions from the replay buffer. Consequently, since any given critic of any given agent is aware of all other agents and account for their actions, non-stationarity of the environment is mitigated. This process is called "centralized learning", as critics do not select the actions.

As shown by the authors, the algorithm is suitable for cooperative and competitive scenarios by utilizing multi-particle environment and its various scenarios, which we discussed in Subsection 2.3.1.

However, It was noted in the literature [IS18], that because MADDPG critics essentially each concatenate and have to process all of the observations of all critics, this might lead to low performance in some scenarios. Consequently, this also results in high memory requirements, as critics make used of replay buffer.

It is also worth noting that there are several modifications of MADDPG, most of which concentrate on the learning performance. But there are also researchers attempting
to enable MADDPG functionality for real-world scenarios. For example, R-MADDPG [WEH20] is a recently published modification designed to enable efficient use of the base algorithm in the distributed systems. In such systems, agents must learn not only to perform the primary task but also to be conscious of the technical limitations of the environment. Specifically, R-MADDPG is focusing on limiting network bandwidth by placing a limit on the observation sharing by the critics. The publication clearly demonstrates the direct correlation between the amount of observation data shared by the critics and the learning performance of the agents. Since MDRL scenarios can potentially include a large number of agents in the same environment, algorithms like MADDPG, where critics must exchange observations, might potentially cause a significant strain on the network infrastructure. When implementing real-world systems relying on MDRL, it is important to understand the limitations and keep in mind the possibility of introducing a trade-off between network load and learning performance, such as proposed by R-MADDPG.

### 2.5.2 Multi-Actor-Attention-Critic

Multi-Actor-Attention-Critic (MAAC) [IS18] is an actor-critic MDRL algorithm similar in its core idea to MADDPG. Each agent has an actor, action picking is based on the local observations, and a critic having access to observations of all agents as well as the ability to infer policies of the other agents. And again at training time, critic affects the actor’s learning process. However, there are several crucial conceptual differences.

First is the way critics infer the actions of other agents. While in MADDPG critics rely on the replay buffer, MAAC allows critics to access actions of other agents directly. MAAC authors show that this approach improves the coordination of agents’ policies in comparison with the approach based on inferring actions from the replay buffer.

Another, and the most crucial difference from MADDPG, is the so-called attention mechanism. As mentioned previously, one of the major drawbacks of MADDPG is the fact that each critic for each agent has to concatenate all of the observations from all of the agents. This a fundamental part of the algorithm design, but it leads to the so-called curse of dimensionality problem. The more agents there are, there larger observation space becomes for every given agents’ critic, which affect both the learning and processing performance of the algorithm. MAAC deals with the problem by "paying attention" only to features of the observation space, which the underlying attention mechanism considers important.

Authors compare MAAC with MADDPG using a modified multi-particle environment. Specifically, they introduce two new scenarios with increased complexity, one cooperative, and one competitive. First is a modified cooperative communication scenario with eight agents instead of two. Second is a completely new competitive "treasure hunting" scenario with eight agents, two of which constitute an adversary team opposed to the remaining six. It is demonstrated by the authors that MAAC is outperforming several baselines, including MADDPG, in these scenarios. However, there is no direct comparison of MADDPG and MAAC in the default MPE scenarios listed in Subsection 2.3.1.
2.5.3 Selected algorithms reference implementation validation

We have performed a comparison and the published code viability tests of the selected MADDPG and MAAC implementations against the default MPE scenarios. Additionally, we compare the results with DQN, a single-agent algorithm within a multi-agent environment. The detailed results and description of the experimental setup are provided in Appendix Section A..

We can see that multiple MPE scenarios exhibit similar learning trends irrespective of the algorithm. Specifically, physical deception, cooperative communication, cooperative navigation. Varying per algorithm but relatively stable results for predator-prey and keep-away. Covert communication turned out to be the least stable, and MAAC was not able to solve the environment.

In most cases, DQN, a single-agent algorithm, was able to solve the environment with a comparable level of stability and reward gained to at least one of the tested multi-agent algorithms. We believe this stems from the inherent simplicity of MPE scenarios. Low dimensional observation space, limited action spaces, and a small number of agents which are not fundamentally altering the environment. These conditions do not present much of a challenge even for a single-agent algorithm. Nevertheless, further investigation for the applicability of single-agent algorithms in our own multi-agent environment might be prudent.

However, the computational complexity of the DQN implementation we used was significantly higher when compared to both MADDPG and MAAC, which significantly limits the research potential due to sheer time requirements. Table 2.5 shows the amount of time that every algorithm required to work through the MPE scenarios at default hyperparameter values. Table values based on mean runtime till scenario completion for seven runs of each scenario for MADDPG and MAAC. For DQN each scenario was executed only two times each due to high amount of time required to finish the scenario.

<table>
<thead>
<tr>
<th>MPE Scenario</th>
<th>MADDPG (minutes)</th>
<th>MAAC (minutes)</th>
<th>DQN (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cooperative communication</td>
<td>27</td>
<td>138</td>
<td>612</td>
</tr>
<tr>
<td>Cooperative navigation</td>
<td>51</td>
<td>252</td>
<td>846</td>
</tr>
<tr>
<td>Keep-away</td>
<td>28</td>
<td>171</td>
<td>438</td>
</tr>
<tr>
<td>Physical deception</td>
<td>35</td>
<td>152</td>
<td>680</td>
</tr>
<tr>
<td>Predator-prey</td>
<td>62</td>
<td>356</td>
<td>914</td>
</tr>
<tr>
<td>Covert communication</td>
<td>37</td>
<td>501</td>
<td>795</td>
</tr>
</tbody>
</table>

2.6 Summary

In this chapter, we have presented the fundamental concepts for deep reinforcement learning. We focused then on multi-agent deep reinforcement learning (MDRL) and how it is different from single-agent deep reinforcement learning. In addition, we reviewed the state of MDRL research from the perspective of the directions tied to the
specific features, which revealed that MDRL is an active but relatively new field of research.

We have reviewed and classified the existing open source multi-agent oriented reinforcement learning environments and noted the common integration functions these present for integration with MDRL algorithms and frameworks. We have outlined the importance of such environments for the research of MDRL in domain-specific tasks.

In addition, we have also taken a brief look at a recent survey on the state of distributed data allocation in databases. This survey clearly indicates a consistent research interest in this area.

Finally, we have also taken a more detailed look at two off-the-shelf MDRL algorithms. We performed validation of the selected off-the-shelf algorithms implementations against a reference multi-agent reinforcement learning environment named multi-agent particle environment (MPE). Thus ensuring RLLib’s implementation of MADDPG is functional, as well as verifying the reference implementation of MAAC in the standard MPE scenarios for direct comparison with MADDPG. We use these two algorithms for the evaluation of the multi-agent data distribution environment designed within the confines of this work. In the next chapter we present our proposed design for a multi-agent data distribution environment.
3 Design of the environment

In this chapter, we discuss the general design concepts of our data distribution environment. We begin with clearly outlining research questions in Section 3.1. Then we proceed with the description of the primary components, control flow, and simulated MDRL scenario in Section 3.2.

3.1 Research questions

1. How do the selected off-the-shelf algorithms perform in an existing reference environment with varying hyperparameter values?

2. How does workload estimation perform in comparison to the benchmark performed on a real-world system?

3. What is the performance of the selected off-the-shelf MDRL algorithms in a scenario of direct manipulation data fragments allocation manipulation with a highly nuanced action space?

3.2 Architecture of the multi-agent data distribution environment

Research in multi-agent reinforcement learning is active and, as we show in Subsection 2.2.2, has multiple distinct directions. Validation and verification of the research in every case is done with using some environment, sometimes a paper specific implementation but often publicly available reference environments, such as discussed in Section 2.3. Algorithm research or benchmarking new MDRL concepts can be done within the most generic environments, which explains the popularity of game-like environments for both single-agent and multi-agent deep reinforcement learning research. However, from our review of the research directions of MDRL we can also clearly see instances such as self-driving cars and certain communications related research relies on the environments, which simulate the real-world usage of the algorithms.

Real-world usecase specific environments are important for verifying the validity and performance of an MDRL algorithm for a particular use case. It would be a naive assumption that an algorithm developed to play an arcade game-like environment, will perform equally well for a self-driving car or a database system. However, to the best of our knowledge, neither the database-centric scenarios in general nor dynamic fragment allocation, in particular, exist in the form of a publicly published environment which is ready for MDRL research.
Our goal with this work is to open the way for MDRL research, specifically in dynamic data allocation within distributed database systems, while taking into account the variability of the existing dynamic data allocation techniques. To achieve this, we design and implement Multi-agent Data Distribution Environment (MDDE) suitable for multi-agent reinforcement learning, meant to be easily interoperable with MDRL frameworks and algorithms, and able to operate with a variable number and nature of optimization goals and constraints.

### 3.2.1 Components

As we mentioned in our motivation, Section 1.1, there is ongoing research of single-agent DRL in relation to various DBMS subsystems. A common approach in this research is to replace a built-in heuristics-based partitioning or query optimizer module in an existing database system with a DRL-based [SSD18, HBR19, MP18a, KYG+18]. This is also true for the generic environments of Park [MNN+19], which was mentioned earlier in this work. Park’s query optimization environment and indexing environment for database related scenarios rely on open source real-world databases.

The described above approach is suitable for exploring opportunities for enhancement of existing database systems with DRL algorithms by replacing specific subsystems, such as query optimizer or storage engine. This, however, would make any research in the area to be specific for the chosen database engine. This makes evaluation of the results easier, as it becomes possible to compare the performance of a proposed DRL algorithm against the default built-in database heuristics. Evaluation of the algorithm, in an environment free of the numerous sophisticated database operations optimizations, becomes problematic. In addition, it often requires a considerable amount of effort to introduce fundamentally new concepts in the real-world database codebase, as the complexity and high level of integration within such code often leads to a steep learning curve for a person unfamiliar with the specific project.

In our work, we strive to provide researchers with a generic environment implementation, which is not dependant on any specific database engine. The highest priority of this work is to make this environment flexible and easily extensible while providing fine-grained control over the allocation of data records within a distributed data storage.

To ensure flexibility and ease of use by the researchers, we make the architecture consisting out of loosely coupled modules with a clear separation of concerns in mind. This allows choosing the most suitable tools for the job done by every specific module, instead of relying on a single platform, that might be suboptimal for some of the requirements. We present the concept of such a design in Figure 3.1, which clearly depicts four main semi-independent components of the architecture: stage, registry, benchmark runner, and data nodes. Changes in the internal workings of one component should not affect the other as long as the communication protocols are staying the same. We expect the stage component to be the most volatile one and the central subject to the modifications and extensions, as this is the component that must integrate with the custom MDRL algorithms, while the registry is responsible for data allocation control and metrics collection.
As described above, our architecture consists out of four primary components, out of which three are the modules that are fully or partially developed within the confines of this work. Data nodes component of our architecture is an unaltered, publicly available database engine. In addition, since the data nodes are not directly integrated within one of the components but instead real database instances, these can be easily rolled out in a real-world infrastructure. Such a setup would allow MDDE to capture not only its own properties and metrics affected by the changes in the data allocation but also properties of the infrastructure itself. We believe this to be a valuable feature of our environment that can potentially allow evaluate MDRL algorithms in a data distribution system, that is geographically distributed and functioning in a real-world setting as any other database system.

The MDDE-specific components constitute three modules: stage, registry, benchmark. In this section, we describe the exact composition, the purpose, and the way these custom components interact with each other. The overall structure and interconnections between all of the most important modules and components of MDDE are depicted in Figure 3.2.

Figure 3.1: Concept of the multi-agent data distribution environment
Figure 3.2: MDDE Components
Stage

The central component of the stage module is \textit{environment}. This component defines the flow of the environment initialization and execution. Environment is the component that wires together the third party MDRL frameworks, user-defined scenarios, and the operations executed by the registry components (including data allocation control and benchmark runner control). The environment component contains the internal logic of MDDE, and that logic is not meant for modification by the end-users of the environment.

As we have shown in Subsection 2.2.2, the research directions of MDRL are quite diverse. Therefore there is a need to define scenarios for a variety of possible hypotheses easily. This is achieved by providing a stable, well-defined interface in the environment component for supplying hypothesis-specific environment representation, including action spaces, observation spaces, and reward functions. This interface is then implemented by the \textit{scenario} component, which is meant to be easily modifiable by the user. The user can define the rules by which the agents interact with the environment. The scenario component requires an implementation of the \textit{agents} component to be supplied as a parameter. Agents component clearly defines the number of the agents $A$, as well as the exact action space $\alpha$ and observation space of each. The scenario component then aggregates these functions and supply them to the environment component.

Additionally, for the execution of the scenario, a \textit{fragmenter} component is required. This component defines the logic used for the initial fragmentation of data. The agents do not manipulate the data records directly, instead fragments $F$ are created by the fragmenter. Each fragment $f \in F$, can contain one or more data record. After being grouped together into a fragment, all data records that belong to that fragment must always be colocated on the same data node. If a fragment is subject to an action, such as copy or delete, this means that this action is done to all of the data records that belong to this fragment. A data record can not belong to more than one fragment at a time.

In our current design, the fragmenter supports only horizontal fragmentation. The fragmentation is done sequentially upon initialization of the environment and remain unchanged throughout the experiment. The fragment accepts the total number of fragments that should be formed, number of fragments must be higher or equal the number of records. The default fragmenter logic attempts to create fragments of equal size, smaller size fragments are possible if data records are distributed unevenly across the data nodes or the total number of records can not be evenly divided upon the target number of the fragments. Note, in our experiments we specifically avoid a situation where there are fragments of uneven size.

The MDRL framework or algorithm interacting with the environment (stage), does not have direct access to the data nodes. All of the actions done by the agents are routed through the environment component, according to the logic defined in the scenario and agents components. Then when this internal scenario logic is executed, it boils down to a call sent to the registry. Such calls are divided into three categories: control, read, and write. Each category is implemented by its own individual client component. Such division is required to ensure constraints on the correctness of the logic implemented by the scenario, while ensuring a clear separation of concerns in the design and implementation.
Control registry functions of the registry are always accessed and executed by the environment component. Such functions include wiping the data from the nodes, generating the test data, snapshot creation and restoration, benchmark execution. None of these functions can be a part of the Markov decision process in any capacity. Therefore these are never exposed to the scenario or agents components directly.

Write registry functions implement data manipulation, such as copy, delete, create fragments. These functions are exposed to the agents in order to facilitate the action execution. Read functions allow retrieval of the fragment allocation map, as well as reading the results of the latest benchmark run. Read functionality is exposed to the scenario and the agents when an observation space is formed or observations are retrieved for reward calculation.

The environment itself does not contain the MDRL agents’ logic and must be integrated with such. As discussed in Subsection 2.3.1, most environments provide, with minor variations, three specific functions: step, observation, reset. Within our design, we follow suit.

Observation function accepts no parameters and returns a map $o : w \rightarrow s_{w,t}$ where $w \in W$ (agents) and $s_{w,t}$ is a numeric array representing the state observed by the agent $w$ at time point $t$, such as allocation, read counts, etc. The exact shape and type of contents in $s_t$ is consistent across all time-points and depends on the scenario. Additionally, the observation function optionally returns a map $L : w \rightarrow l_{w,t}$ that contains a numeric array that specifies correct and incorrect actions which agent $w$ can take in the current state. The exact composition of such array depends on the agent and scenario, in the scenario implemented within the confines of this work, it is a binary map array with length equals to the size of the agent’s action space. In this action map, 1 signifies an action that can be taken by the agent in the current state without breaking any hard-coded constraints, while 0 signifies that the action will be rejected by the registry if taken in the current state.

Step function accepts a map $Z : w \rightarrow \alpha_{w,t}$ where $\alpha_{w,i} \in A_w$ is an action taken by the agent $w \in W$ from the pool of actions $A_w$ available to the agent $w$. Step function returns a map $\varsigma : w \rightarrow T_{w,t}$ where $T_{w,t}$ is a tuple $\langle r_{w,t}, s_{w,t}, d_{w,t} \rangle$, in which $r_{w,t}$ is a reward value obtained by agent $w$ after taking step at time point $t$; $s_{w,t}$ is state observed by the agent $w$ after taking step at time point $t$; $d_{w,t} \in \{0, 1\}$ is a binary 'done' flag signifying that the agent should not do any more steps till the end of the episode.

Final interface function, reset, reverts the environment to the original state via restoring the initial snapshot of the data records and allocation.

Action spaces for all of the agents can be retrieved via the action space interface of the stage. Depending on the way agents define action spaces, there might be a need to initialize the entirety of the environment first, for example if the number of actions of each agent depends on the number of fragments. Otherwise, if the number of actions is static, these can be retrieved before the initialization.
Registry

The registry provides an abstraction layer for data access and manipulation, as well as benchmark control. Each and every data record read and write actions from the stage must be processed by the registry. While stage contains read, write and control clients, the registry provides the corresponding server-side query processors. These processor components contain nothing but the implementation of a communication protocol and should be easily replaceable without affecting the rest of the logic, in case a communication protocol between the components changes. After a query processor receives a request, it is unmarshalled and converted into internal, well defined, data structures of the registry, which are then passed to the corresponding handler components.

Read and write handler components, as the names suggest, are responsible for handling read and write operations. However, these operations are not performed on the data records directly. Instead the registry manipulates internal IDs of the records, IDs of the fragments formed out of these data records, and any other meta-information related to the data records and required for the stage module operations. This means, that the stage can not retrieve the actual contents of any data record because such an operation is pointless within our environment design, it can only control the fragmentation and allocation. All data record and fragment related information is stored internally in the registry within the *data records allocation registry* component. Read queries constitute functions such as current allocation map retrieval (observation) for all or specific data nodes, composition of the fragments, number of the fragments. Write queries include fragment manipulation, such as copy or delete. Additionally write queries allows the stage to control the composition of fragments.

As we point in Subsection 2.5.3, training of the agents even in a simple MDRL scenario as MPE is time-consuming. Therefore, we take every opportunity to speed up this process. It is not practical to execute the benchmark after every step taken by the agents. When an agent executes an action that involves replication or deletion of a data record, this action is recorded in the registry allocation map so that agents would be able to retrieve current observations based on this internal representation, but the actual allocation on the data nodes stay unaltered. Each data allocation alteration action that was invoked by the agents is put into the queue for later execution. That queue is maintained and managed by *data records shuffle queue* component. The idea is that this queue is only executed before an actual benchmark run, hence potentially allowing scenarios where there is no need to retrieve measurements too often. It allows executing a series of actions rapidly, and then make data manipulation queries in bulk before the benchmark. Or perhaps not executing a single action on the actual data nodes throughout the episode because of scenario logic not calling for the real-world benchmark run even once.

*Control query handler* manages all of the operations that are not related to data records manipulation done by the MDRL agents. The purpose of this handler is to provide environment management functions. This handler is responsible for creating and restoring snapshots for both the registry and data nodes. It is also responsible for the initialization of the benchmark procedure and returning the collected metrics to the stage module. The control query handler is also responsible for the initialization of the data nodes and the registry module itself. And finally, it provides the implementation of the reset function that reverts both the *data records allocation registry* component state and data nodes to the original state of the environment as it was initialized.
**Benchmark runner** is the component that encapsulates the logic responsible for the initialization of benchmark data records generation, workload execution, and processing the metrics produced by the benchmark. This component is also responsible for providing the exact record allocation for the benchmark module when the benchmark workload is executed. Benchmark module requests allocation information via a communication protocol defined in the *benchmark callback API* component of the registry. Benchmark communication API is separate from the read query processor component because the benchmark runner component provides read-only current data allocation access to the benchmark. This benchmark-specific allocation map is not reading directly from the data records allocation registry component during the benchmark workload execution. Instead, the runner component creates an in-memory data structure designed specifically for high-frequency concurrent reads. Thus attempting to reduce the registry module overhead footprint in the resulted benchmark throughput.

Additionally, the benchmark runner component is responsible for maintaining a data node access counter, keeping track of how many concurrent reads from any specific data node are happening during a benchmark workload execution.

Direct interaction with the benchmark module is performed by the *benchmark controller* component of the registry. It contains a predefined set of data generation parameters and read-oriented workloads. These are used as parameters passed to the benchmark when a corresponding function is called either by the benchmark runner or by the control query handler.

**Benchmark estimator**, is the component encapsulating workload estimation logic, which is discussed in Subsection 3.2.2. Essentially it is an alternative mechanism capable of estimating benchmark metrics without executing the data shuffle queue or executing any real queries to the data nodes. However, it is not capable of generating data records and requires the benchmark (we also refer to it as real-world benchmark) component to establish baselines for the metrics by executing the chosen workload at least once. Only after data records are generated, and baselines are established, the estimation mechanism can be used.

The *snapshot controller* component is responsible for creating and keeping track of the data nodes and the registry state snapshots. In the general case, it should only create one snapshot after the environment is first initialized, and before any actions by the agents are taken. This is the snapshot to which the environment is reverted when the reset function is called. Replacement or modification of this snapshot is not permitted over the lifecycle of MDDE environment.

**Benchmark**

The purpose of the benchmarking module in our architecture is twofold. First, it must be able to generate a specified amount of data records of the given size. Second, is to execute Online-Transaction-Processing (OLTP) reproducible data access queries that constitute a measured benchmark workload.

As can be seen in Figure 3.2, there are four primary components of the benchmark module. First is the *common benchmark functions* component, which is database agnostic and contains all of the common logic of generating the data and workload execution. Second
is the MDDE-specific database client, which implements a specific database access layer while taking into account the existence of the registry module. The database client has to notify the registry about all data manipulations performed, as well as it should query the registry for the exact location of any given data record. Communication between the registry and the benchmark module is done via MDDE Client that implements a specific communication protocol specific for the benchmark operations. The last essential piece of the benchmark is the access logs writer, which collects the exact access information such as which data records were accessed by the benchmark and on which data nodes. After the end of a benchmark run, this data is later aggregated and sent to the registry, where it is further processed and subsequently supplied to the MDDE environment component upon request.

We have decided in favor of data generation over static datasets. This allows us to control precisely the size and composition of the generated records. The initial dataset is generated as part of the environment initialization, as depicted in Figure 3.5. Each generated record must have a unique key, which is stored in the registry in the fashion explained earlier. The generation of the data records is a time-consuming process because these must be placed sequentially to the designated data nodes. For each, the registry must be notified about the newly created record and its location. A data generator must also ensure that a newly generated data record was indeed saved in the data node correctly and recorded by the registry. There is no inherent transactional atomicity between the data nodes and the registry, as these are semi-independent, hence the correctness of insertion must be verified by the data generator.

**Data nodes**

Data nodes in our architecture is a set of two or more distributed data storage locations. There are only three basic requirements for the data nodes. First, these must be compatible with the benchmark implementation. The benchmark must be able to generate data records and perform a workload run where the throughput is measured, and data record access statistics are collected, such as the number of reads per record per node.

Second, the data nodes must be as free of or contain a minimum of heuristic optimizations of their own. The point of our environment is to assess the performance of MDRL algorithms without influence of sophisticated heuristics for data distribution. Mixing and matching MDRL with heuristics has merits in principle but beyond the scope of this work.

Finally, and most importantly, the data nodes must allow fine-grained control over data records allocation by the registry. In essence, it can be either a distributed database, that provides a way to override its default data record distribution, or just a set of entirely disconnected data storage solutions.

### 3.2.2 Measuring performance

The key aspect of the designed environment is the ability to reason the quality of distribution. This is done by executing synthetic benchmark read queries according a spe-
cific workload, while collecting various statistics from data nodes. In our work, we rely on throughput as the primary measurement for the quality of the distribution. Higher throughput is better. Additionally, the number of reads per fragment during the workload execution is collected and used later in the agents’ learning process.

Workload execution

The data access workloads must be reproducible. Such as, unless otherwise specified, the benchmark must exhibit the same access pattern if executed over the same set of data records. The most and least frequently accessed data records must remain such across different runs of the benchmark. This behavior should allow us to evaluate the ability of an MDRL algorithm to adapt to a specific workload adequately.

More precisely, workload execution yields a tuple containing three metrics. First, is the throughput $\tau$, which signifies the capacity of the data storage to retrieve data. Higher throughput means more records can be retrieved within a fixed-length point of time. We believe that throughput is the most important metric within our environment design as it can show the exact differences in system performance under different data record distribution configurations.

Second metric is the set $N = \bigcup_{w \in W} N_w$, where $N_w$ is a set with the number of reads $\nu_{f,w}$ for each fragment $f \in F$, that exists in the environment, per data node (agent) $w$. More precisely $\nu_{f,w}$ is a an exact number of times fragment $f$ was read from node $w$. This metric shows the exact statistics of how many times every fragment was retrieved from each node that contains it. Total number of reads per fragment $f$ can be found by simply summing up all reads of fragment $f$ from all nodes (Equation 3.1).

$$f_{\text{reads}}(f, N) = \sum_{N_w \in N} \left( \sum_{\nu_{f',w} \in N_w, f'=f} \nu_{f',w} \right)$$  (3.1)

The final metric is the set $R \neq \emptyset$, which contains the total number of reads $r_w$ performed from the data node that belongs to each agent $w \in W$, from agents $W$ defined within the current configuration of the environment. This metric is important and shows specifically the level of load on every specific node during the workload execution. Each $r_w$ is calculated as a sum of values $\nu_{f,w}$ from the corresponding $N_w \in N$ (Equation 3.2).

$$r(w) = \sum_{\nu \in N_w} \nu$$  (3.2)

To summarize, the final output of the benchmark run is a tuple $\langle \tau, N, R \rangle$. 
Estimated workload

Running workload execution is a very expensive operation. It requires first moving fragments across different data nodes and then performing a series of data access operations as defined in the selected workload. It might be impractical and too time-consuming, especially for early training stages, where agents would typically explore the action space and attempt to build associations with the observation space. We strive to speed up this stage by providing agents with estimations instead of running an expensive workload frequently early on. In addition, such estimation mechanism is useful for quick hypothesis evaluation or using MDDE in a system with limited amount of resources.

Workload estimation is performed based on the previous workload execution by the real world benchmark, the results of which are used as a baseline. The baseline throughput $\tau$ produced by the workload is associated with the read distribution $R$ baseline imbalance $D(R)$, calculated as shown in Equation 3.3. Imbalance, in this case, means the sum of degrees of difference in the nodes utilization during the workload run. If all of the nodes are utilized equally, meaning the number of reads performed from every node was equal, then $D(R) = 0$. The following equation shows the proposed calculation for the read imbalance. In this equation we compare the reads of each node, against all (including itself), and we normalize each produced value by the total number of reads.

$$D(R) = \frac{1}{2} \sum_{r \in R} \sum_{i \in R} \frac{|r - i|}{\sum R}$$

(3.3)

The maximum imbalance $M$ value depends on the number of nodes $|W|$ and can be calculated via Equation 3.4.

$$M = 2(|W| - 1)$$

(3.4)

Assuming that between the real benchmark run and the requested estimation, agents were executing commands modifying the allocation of data in the registry, we need to estimate the number of reads $N'$ that would be hypothetically executed against every fragment per data node. We achieve this by using a simple heuristic to estimate the number and distribution of reads among the nodes. Since agents were active, we have a current allocation map $G$ which is a set of tuples $(f, f_w, f_{\text{reads}}(f, N))$, containing fragment $f \in F$, nodes where fragment is allocated $f_w = \{w \in W | f \in w\}$ and baseline number of reads for this fragment $f_{\text{reads}}(f, N)$. To be more precise, each tuple $g \in G$, contains three elements: $g_f$ is the fragment, $g_{f_w}$ is the set of nodes where the fragment is allocated, $g_{\text{reads}}$ is the baseline number of reads recorded for fragment $f$.

$G$ is pre-sorted by the number of replicas $|f_w|$ in ascending order and then additionally by the number of baseline reads $f_{\text{reads}}(f, N)$ in descending order. For clarity, a sorted example of $G$ is illustrated in Figure 3.3.

We estimate exact number of reads per fragment $f \in F$, per node $w \in W$ with Algorithm 1. The algorithm receives the current sorted allocation map $G$ and the set of nodes $W$ as input. The purpose of this algorithm is to spread the number reads as equally as possible across all of the nodes, while taking into account the allocation of
the fragments. It iterates over $G$ from the start. The algorithm distributes the reads as follows:

- All of the fragments that have only a single replica, get the number of reads that are specified in the baseline $R$ per fragment added to the read counter associated with the nodes where these replicas are allocated.

- If there a fragment $f$ is replicated more or equal number of times than there are baseline reads $f_{\text{reads}}(f, N)$, then the first $f_{\text{reads}}(f, N)$ (integer: baseline number of reads) nodes where this $f$ replicas are present, get their read counters increased by 1. If there are more replicas than the number $f_{\text{reads}}(f, N)$, then the rest of the allocations receive the 0 reads for the fragment $f$.

- If fragment $f$ baseline has more reads $f_{\text{reads}}(f, N)$ than fragments, then the Algorithm 2 is used to distribute reads across the nodes for fragment $f$, while taking into account the current estimation of the reads allocated to them. We do not simply divide the number of baseline reads $f_{\text{reads}}(f, N)$ for fragment $f$ over the number of replicas but spread the reads in the way that the least used node will get more reads assigned to it than the one that already have a lot of reads from other fragments $f' \neq f$.

This algorithm, is a basic heuristic but in pursuit for least computational overhead we specifically opted out of using a more sophisticated metaheuristic algorithm, such as genetic algorithm, for the purpose of estimating reads. Based on the estimated $N'$ we calculate an estimated read distribution set $R'$, similarly to $R$ in actual benchmark. Baseline $R$ and the estimated $R'$ then supplied to Equation 3.5.

In the estimation of the throughput $\tau$ we follow a naive assumption that if the load distribution among the benchmark run affects the throughput directly. The higher degree of load balance among the data nodes $W$, the higher is throughput. In the ideal case, every node $w \in W$ serves the same percentage of read queries. For example, if $|W| = 4$, then in the best case scenario we have each node serving exactly 25% of reads during the benchmark run. The worst case then would be when a single node $w$ serves 100% of queries, while three others serve 0%. The worst case value is the maximum imbalance value $M$. This assumption is naive as we are not taking into account the temporal aspect of incoming queries, however our goal to keep estimation as computationally inexpensive as possible. We hypothesize that the naive estimation is sufficient for at least early stage training of the agents.

$$\tau' = 10^{\log_{10}(\tau) - \zeta(D(R') - D(R)) \log_{10}(1 + \frac{|D(R) - D(R')|}{M} + \beta)}$$  \hspace{1cm} (3.5)$$

Variable $\zeta$ is the direction of change for the imbalance calculated as shown in the Equation 3.6. We perceive lower imbalance as positive and would want to increase the estimated throughput value $\tau'$ relative to baseline $\tau$. While higher imbalance is neg-
Algorithm 1: Read distribution $N'$ estimation.

Input: $G, W$
Result: $N'$

/* $W_{reads}$ is an array of integer read counters per node with initial values 0. Used for estimation balancing. Array index equals index of nodes $w \in W$. */

1. $W_{reads} \leftarrow \text{Array}[|W|]$; // Create array of integers with length equals number of agents. Initial values are 0.

2. foreach $g \in G$ do

3. $t_{reads} \leftarrow g_{reads}$; // Total number of baseline reads for $f$ (recorded from the YCSB workload execution).

4. $t_{alloc} \leftarrow g_{fw}$; // Nodes where $f$ is allocated.

5. if $|t_{alloc}| = 1$ then

   /* Only one replica of fragment $f$ exists. */

   6. $t_w \leftarrow t_{alloc}.\text{first}$; // Get the first node index out of all allocations for fragment $f$.

   7. $W_{reads}[t_w] \leftarrow W_{reads}[t_w] + t_{reads}$;

   8. $N'.\text{Append}((f, t_w, t_{reads}))$;

else if $t_{reads} \leq |t_{alloc}|$ then

   /* Number of replicas is equal or higher than reads. */

   9. for $i \leftarrow 0$ to $t_{reads}$ by 1 do

      10. $t_w \leftarrow t_{alloc}[i]$;

      11. $W_{reads}[t_w] \leftarrow W_{reads}[t_w] + 1$;

      12. $N'.\text{Append}((f, t_w, 1))$;

   end

else

   /* Distribute reads for fragment $f$ among nodes containing its replica, while taking into account already assigned reads for other fragments on these nodes. */

   13. $N', W, W_{reads} \leftarrow \text{BalanceFragment}(N', W, W_{reads}, t_{reads}, t_{alloc})$;

      // Algorithm 2.

end

14. return $N'$;
Algorithm 2: Balance fragment reads among nodes.

BalanceFragment $N', W, W_{reads}, t_{reads}, t_{alloc}$:

1. $read\_sum \leftarrow \sum W_{reads}$;  // Current estimated total sum of reads on all nodes.
2. $t_{w\_prc} \leftarrow \text{Array}[|W|]$;  // Empty array of Floating point numbers, current percentage of participation per node.
3. foreach $t_w \in t_{alloc}$ do
   4. if $read\_sum > 0$ then
      5. $t_{w\_prc}[t_w] \leftarrow \langle t_w, \frac{W_{reads}[t_w]}{read\_sum} \rangle$;
   6. else
      7. $t_{w\_prc}[t_w] \leftarrow \langle t_w, \frac{1}{|t_{alloc}|} \rangle$;
   8. end
4. $t_{w\_asc} \leftarrow \text{Sort. Ascending}(t_{w\_prc})$;  // Sort by percentage ascending.
5. $t_{w\_desc} \leftarrow \text{Sort. Descending}(t_{w\_prc})$;  // Sort by percentage descending.
6. $reads\_rem \leftarrow t_{reads}$;  // Amount of not yet assigned reads for fragment $f$.
7. for $i \leftarrow 0$ to $|t_{w\_asc}|$ by 1 do
   8. $prc \leftarrow t_{w\_asc}[i]$;  // Tuple $\langle$ Node ID, participation % $\rangle$.
   9. $inv \leftarrow t_{w\_desc}[i]$;  // Tuple $\langle$ Node ID, participation % $\rangle$.
   10. if $i < (|t_{w\_asc}| - 1)$ then
       11. $chunk \leftarrow \lceil t_{reads} \times inv \times prc \rceil$;  // Ceiling of total reads of $f$ times inv percentage.
       12. $reads\_rem \leftarrow reads\_rem - chunk$;
       13. $W_{reads}[t_w] \leftarrow W_{reads}[prc\_t_w] + chunk$;
       14. $N'.\text{Append}((f, prc\_t_w, chunk))$;
       15. else
       16. $W_{reads}[t_w] \leftarrow W_{reads}[prc\_t_w] + reads\_rem$;
       17. $N'.\text{Append}((f, prc\_t_w, reads\_rem))$;
   18. end
19. end
20. return $N', W, W_{reads}$;
ative, hence, instead of increasing $\tau'$ we want to reduce the value. $\beta \in (0, \infty)$ is a modifier that allows us to control the magnitude of the estimated changes in throughput.

$$\zeta(x) = \begin{cases} \frac{|x|}{x} & x \neq 0 \\ 1 & x = 0 \end{cases}$$

(3.6)

Based on the baseline throughput and imbalance, we can estimate the theoretical best-case throughput. The core assumption is that if reads are equally spread across all of the nodes, meaning $D(R') = 0$, we get the highest throughput. It is a naive assumption that does not take into account the latency to the nodes and any other possible conditions that might negatively affect the node’s performance. We assume that all of the nodes are equal and have identical properties.

$$\beta = \left( |D(R) - D(R')| \right) \frac{k - j}{M} + j$$

(3.7)

It is problematic to make a universal model that would reflect the degree of change in throughput that allocation distribution changes would exhibit in real world setting. Therefore, $\beta$ variable should be used to tune estimation with the real system at hand. This can be achieved through retrieving throughput for the theoretical edge cases of allocation. The most useful are two such cases: best case and worst case. Where the best case is an equal distribution of reads for any given workload, worst case can be achieved by simply placing all of the fragments to a single node, forcing all of the workload activity to be executed on that one node, while the rest are idle. For simple estimation $\beta \in [0, 1]$ can work just fine. However, if we want to estimate a situation where throughput increase or decrease trends closer resemble the real world benchmark, we can use a simple function, as shown in Equation 3.7, that maps the degree of imbalance change to the range $[j, k]$, where $j \in [0, 1]$ and $k \in (j, 1]$. Values $j$ and $k$ are hyperparameters in this case.

The output of the estimated benchmark values is similar in composition to the real benchmark workload run, it is a tuple $\langle \tau', N', R' \rangle$.

It is worth noting that this estimation mechanism does not take into account differences in the properties of data nodes. This estimation assumes that the cost of retrieval from all of the nodes is equal, such as the latency and speed of reading the data records are the same. This is suitable for preliminary evaluation and local tests. However, such estimation is not suitable for the system where data nodes are geographically distributed or rolled out on a system with different performance. In such cases, execution of the real-world benchmark is preferable so that the agents could take into account the infrastructure properties.

### 3.2.3 Control flow

The full composition of the environment involves five components: a learner, an instance of the MDDE stage, instances of the MDDE registry, benchmark, and data nodes. The learner is the MDRL algorithm implementation, and it can be an MDRL framework or just
an independent code supplied by a researcher and consuming the observation, step, and reset functions provided by the instance of the MDDE stage. The stage, registry, benchmark, and data nodes are as described in Subsection 3.2.1.

In Figure 3.4 we demonstrate the overall lifecycle of the environment. It always starts with the environment’s initialization, which is done before the learner takes any action. The exact initialization sequence is displayed in Figure 3.5. There the learner code triggers the initialization function in the MDDE stage instance, which in turn executes a parameterized call to the registry instance with the exact number of the data records to be created. This request is verified and processed by the registry, which in turn initialized the benchmark’s data generation function. In order to ensure the absolute consistency of data records allocation on the data nodes and the registry, the process is performed in a single-threaded sequential manner. This way, we ensure that the registry and the
data nodes are consistent from the very start, including in terms of the insertion order. This sequential insertion is, however, has a downside of resulting in $O(n)$ data generation complexity where $n$ is the number of records generated.

After all of the records were successfully generated by the benchmark, and the stage instance acknowledged it, a snapshot of the current registry and data nodes states is created. This snapshot is designated as the initial state of the environment for the entirety of the current MDDE instance lifecycle.

![Environment Initialization Diagram](image)

**Figure 3.5:** Environment initialization

After the initialization, a predefined number of learning steps is executed. Each step is the sequence as displayed in Figure 3.6, executed in a loop, of a predefined maximum number of iterations. A learner, in this case, is an MDRL algorithm or an MDRL framework, where the algorithm for each MDRL agent is implemented. At the start of the step sequence, all agents request an observation of the current allocation state from the registry. Based on the received observation, each agent picks an action that should be executed within this step. We support a situation where agents take steps together or one by one. Not all agents must participate in every step. However, the MDRL algorithms we have considered within this work always assume that all agents take a collective step every time.

After agents supply the list of chosen actions for execution to the stage, these actions are executed one by one. Order of the execution of the action depends on the agent ID (numeric, sorted in ascending order) and the same throughout all steps and episodes. We execute actions sequentially to ensure data constraints consistency. We hypothesize that as long as the order of execution stays the same, sequential execution of actions should not hinder the learning process. Additionally, we hypothesize that because MADDPG and
MAAC achieve coordination of agents by employing a centralized critic. Execution of action in stable order should allow the agents to account for sequential execution of actions, as the critic attempts to infer the policies of other agents. The order action execution is controlled by the scenario in the stage component.

After all agents’ actions are executed or rejected by the registry, the stage instance receives execution feedback, which is processed by the scenario component of the stage. Upon processing this feedback and other internal logic, the scenario component of the stage makes a decision if a benchmark run must be executed now or not. It is an important consideration because running the benchmark with every step is not practical due to it being expensive time-wise. If the benchmark was not invoked by the scenario, rewards for the current step are calculated by taking into account only past benchmark metrics and the fact if the actions taken by agents were accepted or rejected by the registry. Without running a benchmark, no actions taken by the agents are actually reflected in the data nodes but instead pushed into the queue for later execution.

However, if the benchmark run is invoked, the scenario selects between a real-world benchmark and an estimation. If the real-world benchmark is chosen, two things happen. Firstly, a queue of actions is executed so that the allocation of records in the data nodes reflects the actions taken by the agents and can be measured by the benchmark. Secondly, the benchmark is initiated. Benchmark consists of two parallel processes: benchmark workload execution, metrics collection.

Benchmark sequence in Figure 3.7 depicts an example with two parallel simulated client querying data nodes at the same time and measuring throughput. Which node to query is decided by the registry, and in addition, the registry tracks how many clients access each node concurrently. While the benchmark is running, access statistics are collected and processed in parallel. After the end of the benchmark run, throughput value is returned from the benchmark to the registry and consequently to the stage. At this stage, the reward for each agent can be calculated based on the collected metrics, instead of just basing it on the success or rejection of the action.

After the limit for the number of steps in the episode is reached, the environment must reset. This means the current data allocation is flushed from the data nodes and registry. The initial data allocation snapshot is restored. All of the collected metrics are also reset to their initial values in the stage. After the reset, a new episode starts. While the environment was reverted to its initial state, the learner retains all of the experiences. Therefore, the agents can use all of the previously acquired training in an attempt to solve the environment again, while possibly making better steps yielding higher rewards.

### 3.2.4 Scenario

The core idea behind our hypothesis is that we can use a cooperative MDRL algorithm so that agents collectively develop a policy that results in a sequence of actions ensuring the highest possible throughput of the system with read-only OLTP access workloads.

As evident from our review of the current state of MDRL research in Subsection 2.2.2, there is a large range of possibilities when it comes to designing scenarios and MDRL
Figure 3.6: Learner step sequence

algorithms, which potentially opens multiple possibilities for complex scenarios, such as variable number of agents or nodes, hierarchical node control, etc. However, such scenarios
Figure 3.7: Benchmark sequence (example with two threads)

rely on the specific algorithms’ implementations, usually designed for a specific action and observations spaces, while in our work we strive to develop a generic data distribution optimization scenario with simple rules.

**Action space**

We design action space for this scenario with two goals in mind. First, it has to be uniform across all of the agents. Our target algorithms for the scenario are MADDPG and MAAC, and as such, critics in these have access to the experiences of other agents. As we can see in our initial evaluation of the algorithms against MPE in Appendix A., it results in agents that belong to the same group to learn identical policies. In this case, identical action spaces are a clear advantage.

The second goal is the simplicity of interpretation. This means any potential relationships between the action space and the observation space must be easy to establish. In
the pursuit of simplicity, we present a naive action space which is not designed to scale well and not intended to the real-world use, but instead for basic evaluation of the concept.

Our example environment is fully observable, which means every agent can directly observe the allocation of the fragments and, most importantly, aware of the existence of each fragment in the system. Two basic actions can be done with the fragments: copy and remove. An agent can copy a fragment from another agent, but only if the source object has the fragment and the destination does not. An agent can remove a fragment replica from itself, but only if it is not the only exemplar existing in the system. Agents can not remove a unique exemplar of a fragment. The aforementioned constraints are enforced by the registry and can not be altered in the scenario, hence ensuring data safety irrespective of the scenario.

An example of such action space can be seen in Table 3.1. It is a simplistic example where four agents $W = \{w_0, w_1, w_2, w_3\}$ are managing two data fragments $F = \{f_0, f_1\}$. Total number of actions is $|A| = |W| \ast |F| + |F|$. This action space assumes that each agent can only contain a single copy of any given fragment $f \in F$. Delete actions are defined once per fragment and give the agent the ability to remove a fragment from itself if the constraints described above are satisfied. Copy actions are, however, defined for all agents, even making an inherently invalid action allowing the agent to attempt copying from self to self. For example, if agent $w_0$ attempts to execute the action with index 2 it will always fail as it is a constraint hardcoded in the registry. Still, the actions exist for uniformity of the action space shape across all agents. We assume the agent is capable of learning not to execute these actions by associating its own observation of the state the action space.

Additionally, we provide an optional skip or do-nothing action that allows the agents to skip the step and to do nothing, as displayed in Table 3.2. If this action is allowed the total number of actions becomes $|A| = |W| \ast |F| + |F| + 1$.

In addition to the general action correctness that holds at any state, as indicated in Table 3.1, the validity of actions also depends on the current state $s$. If skip action is active, it is always valid and can be executed by any agent in any state. At the same time, the validity of copy and delete actions for each agent depends strictly on the

<table>
<thead>
<tr>
<th>Action index</th>
<th>Fragment</th>
<th>Source</th>
<th>Destination</th>
<th>Action</th>
<th>Valid for</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$f_0$</td>
<td>self</td>
<td>-</td>
<td>Delete</td>
<td>$w_0$ ✓ $w_1$ ✓ $w_2$ ✓ $w_3$ ✓</td>
</tr>
<tr>
<td>2</td>
<td>$f_1$</td>
<td>self</td>
<td>-</td>
<td>Delete</td>
<td>$w_0$ ✓ $w_1$ ✓ $w_2$ ✓ $w_3$ ✓</td>
</tr>
<tr>
<td>3</td>
<td>$f_0$</td>
<td>$w_0$</td>
<td>self</td>
<td>Copy</td>
<td>$w_0$ ✗ $w_1$ ✓ $w_2$ ✓ $w_3$ ✓</td>
</tr>
<tr>
<td>4</td>
<td>$f_1$</td>
<td>$w_0$</td>
<td>self</td>
<td>Copy</td>
<td>$w_0$ ✓ $w_1$ ✗ $w_2$ ✓ $w_3$ ✓</td>
</tr>
<tr>
<td>5</td>
<td>$f_0$</td>
<td>$w_1$</td>
<td>self</td>
<td>Copy</td>
<td>$w_0$ ✓ $w_1$ ✓ $w_2$ ✓ $w_3$ ✓</td>
</tr>
<tr>
<td>6</td>
<td>$f_1$</td>
<td>$w_1$</td>
<td>self</td>
<td>Copy</td>
<td>$w_0$ ✓ $w_1$ ✓ $w_2$ ✓ $w_3$ ✓</td>
</tr>
<tr>
<td>7</td>
<td>$f_0$</td>
<td>$w_2$</td>
<td>self</td>
<td>Copy</td>
<td>$w_0$ ✓ $w_1$ ✗ $w_2$ ✓ $w_3$ ✓</td>
</tr>
<tr>
<td>8</td>
<td>$f_1$</td>
<td>$w_2$</td>
<td>self</td>
<td>Copy</td>
<td>$w_0$ ✓ $w_1$ ✓ $w_2$ ✓ $w_3$ ✓</td>
</tr>
<tr>
<td>9</td>
<td>$f_0$</td>
<td>$w_3$</td>
<td>self</td>
<td>Copy</td>
<td>$w_0$ ✓ $w_1$ ✓ $w_2$ ✗ $w_3$ ✓</td>
</tr>
<tr>
<td>10</td>
<td>$f_1$</td>
<td>$w_3$</td>
<td>self</td>
<td>Copy</td>
<td>$w_0$ ✓ $w_1$ ✓ $w_2$ ✓ $w_3$ ✗</td>
</tr>
</tbody>
</table>
Table 3.2: Scenario Action space: Optional action

<table>
<thead>
<tr>
<th>Action index</th>
<th>Fragment</th>
<th>Source</th>
<th>Destination</th>
<th>Action</th>
<th>$w_0$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Skip</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

fragments allocation at the current state $s$. Agent $w$ can only delete a fragment $f$ that is currently allocated in it $f \in F_w$ where $F_w$ is a set of fragments allocated on agent $w$, and there is at least one other copy of the specific fragment allocated on a different node $\emptyset \neq \{w' \in W | w' \neq w \land f \in F_{w'}\}$.

As opposed to delete, copy action is valid for the agent $w$ if the fragment $f$ is not already allocated on the agent $f \notin F_w$. Additionally, unlike in the case of delete action where there is only one such action associated with each fragment, there can be multiple copy actions for each fragment as any other agent present in the system can have a replica of the specified fragment. Therefore, in the pool of the agent’s actions $A_w$, only actions where action $a$ is associated with the specific fragment $a_f = f$, and the action’s source $a_{src}$ contains the fragment $f$ are valid, $C_{w,f} = \{a \in A_w | a = copy \land f = a_f \land f \in F_{a_{src}} \land a_{src} \neq w\}$. Because MDDE implements a non-overridable constraint that prohibits the distribution algorithm from removing unique fragments, it is always assumed that if the current agent does have a replica $f \notin F_w$ then there will be at least one valid copy action for this fragment $C_{w,f} \neq \emptyset$.

**Observation space**

This scenario is a fully observable scenario. Each agent sees the entire allocation of data fragments, meaning each agent observes all of the fragments which are stored on its own data node as well as all fragments stored on all other data nodes. This allows every agent to know where each fragment is located and how many copies of it exist. Total size of the observation space equals $|W| \times |F| \times 3$.

By default, the observation space is a multidimensional matrix, where rows correspond to agents $w$ and columns to fragments $F$. In intersections, there are three dimensions. First, is a binary flag containing 1 if a specific fragment $f$ is allocated on the agent $f \in F_w$, 0 otherwise.

The second dimension is the popularity of the specific fragment obtained during the latest benchmark run. Popularity values $\kappa$ lie within the range $[0, 1]$. Popularity is calculated as $\kappa = R_{f,w}/R$, where $R$ is a total number of read operations and $R_{f,w}$ is a number of reads of the specific fragment $f$ from the agent $w$.

Final dimension, is a binary flag containing 1 if a specific fragment slot for fragment $f$ belongs on the agent $w$, 0 otherwise. This dimension should not be confused with the first one, which is an allocation map in the current state $s$, while this dimension is static across all of the states. While both of the previous dimensions in the observation space contain the same values for all of the agents in the scenario, this dimension values are unique per agent. Its purpose is to assist the learning process for the MDRL agents by clearly marking the data fragment slots belonging to this specific agent. Values in this dimension do not depend if the specific agent currently holds a replica of the fragment or
not, it merely indicates the 'slot' where the specific agent observing the dimension might place a replica of a specific fragment.

Example visualization of such observation space for agent $w_0$, based on our simplified example with four agents and two fragments, is depicted in Figure 3.8.

![Figure 3.8: Example observation space (for agent $w_0$)](image)

**Reward function**

Reward function determines the feedback agent receives after executing an action, and this function should reflect the agent’s purpose. In the case of an MDRL, the agents each can have their own reward function, distinct from the rest. Such is the case in MPE environment discussed in Subsection 2.3.1, where "good" agents and "adversarial" agents have a different purpose and therefore rewarded for different actions, meaning they have different reward functions.

In our case, we want to simulate a scenario where agents cooperate to achieve a common goal, the optimal distribution of the data fragments among the data nodes. Therefore, we design a common reward function for all agents. We assume that the optimal distribution of data within our environment results in higher maximum throughput than that of a suboptimal distribution. Throughput is used as the base of our reward function. However, throughput is measured for the entirety of the system, while we also want to give higher or lower reward to the agents based on their own respective actions.

We start by determining the quality of actions taken by the agents. This quality is determined by Equation 3.8. This simple function returns $-1$ for the action feedback $a_{w,s}$ taken at step $s$ by the agent $w$, if the value of $a_{w,s}$ is an error code defined in the set of known error codes $E$. Error codes are defined in the set $E$ and cover errors such as an attempt to remove a unique fragment replica or attempt to copy fragment that is already allocated on the agent that tries to copy it to self again. In case, the agent
have chosen to skip the step and do nothing, the result will be value $V \in [0, 1]$, which is a hyperparameter controlling the desirability of idling agents. In case the action was correct, $a_{w,s} \notin E$, and not skip, the function returns 1.

$$P_w(s) = \begin{cases} 1 & a_{w,s} \notin E, a_{w,s} \neq \text{skip} \\ V & a_{w,s} = \text{skip} \\ -1 & a_{w,s} \in E \end{cases}$$ (3.8)

Equation 3.8 can also be used as a reward function for the steps taken by the agents where the benchmark is not executed. Executing benchmark at every step is too expensive and time consuming, even estimation benchmark called at every step might lead to a significant slowing down of the learning process. While the values of such a reward function are very limited, it provides the agent with the critical negative feedback $-1$ if the action taken was not correct. Additionally, we hypothesize that the $\gamma$ hyperparameter (discussed in Section 2.1) of the agents set to value close to 1 should allow the agent to plan for a higher gain later, for after the benchmark is executed and throughput is retrieved. The reward function which is called when the benchmark metrics are available is defined in Equation 3.9.

$$R_w = \tau \sum_{f \in F_w} \sum_{i=1}^{S} \frac{action(P_w(s))}{S} + \tau \frac{|F| - |F_w|}{|F|} \rho$$ (3.9)

Where $\tau$ is throughput, $W$ is a set of agents, $F$ is a set of fragments, $F_w$ is a set of fragments allocated on agent $w$, $S$ is a number of steps in between benchmark runs. $f_w$ is a specific fragment replica allocated on agent $w$ ($f_w \in F_w$), while $f_{w,r}$ is the number of reads of fragment $f$, from a node $w \in W$ performed during the latest benchmark run.

$\rho \in [0, \infty)$ is storage cost, which controls the degree of influence the storage related term has on the reward function. If the intention to prevent agents from simply hoarding all of the fragments, we can set $\rho$ to a value higher than 0 so that the agents get higher reward if they have a smaller amount of currently allocated replicas. This would allow us to introduce an incentive for the agent to weight the gains of getting a fragment replica, as opposed to getting a reduced reward for using more storage. If storage is not important or outside of the research setup, $\rho$ can be set to 0 so that the entire storage cost term $\tau \frac{|F| - |F_w|}{|F|}$ of the reward function is ignored. If the agent contains all of the fragments, it will get no benefit from the storage term at all in all cases, this should discourage excessive fragment hoarding.

Essentially the reward function in Equation 3.9 can be read as follows:

1. Throughput $\tau$ is first scaled by the percentage of total number of reads done from agent $w \in W$. Higher percentage of reads translates into higher reward.
2. Then it is scaled by the percentage of correct actions taken by the agent \( w \) since the last benchmark. Smaller amount of incorrect actions taken translates into higher reward.

3. Finally, the last term adds the throughput scaled to the percentage of total number of fragments \( F \) currently allocated on the agent. Smaller number of allocated fragments translates into higher reward. It can be viewed as storage cost term. As mentioned previously, importance of this last term is controlled by \( \varrho \).

Note, the transmission cost is not considered within our default scenario. Our goal is to keep this scenario computationally and logically simple. The introduction of transmission costs would require keeping track of action types and possibly assigning different costs of copying fragments from different nodes. It increases the complexity of the observation space and requires introduction of an additional term to the reward function that would somehow account for the cost of retrieval of fragments from different nodes. It is technically supported by MDDE design, and the fact that in our scenario agents have the ability to decide not only which fragment to retrieve but also from which source node. However, the addition of such considerations to the reward function and observation space could overcomplicate the current task and potentially make the agent’s learning infeasible time constraints of this work. The focus of our scenario is facilitation of the best read-access throughput.

**Intent of the scenario design**

Action space in our scenario is very nuanced, which means that certain actions are only permitted in specific states. Taking inappropriate action in a wrong state should be clearly indicated to the agent as incorrect with negative reward. Our hypothesis is that an agent should learn when the appropriate state when any specific action is correct and not going be rejected due to a hardcoded constraint.

We hypothesise that such an action space coupled with an observation space that clearly indicates the full allocation and popularity map, should theoretically result in a policy where the agent is capable of evaluating a potential effect on acquiring or getting rid of the specific fragment. Therefore, if the agent observes that a specific fragment is not popular but is allocated on multiple nodes, it is reasonable to delete the fragment from own storage, given that all other agents would not decide to do the same. In the case of a successfully learned policy like that, this scenario can serve as a basis for more complex scenarios, for example, including actions such as splitting or merging fragments.

**3.3 Summary**

In this chapter, we present a detailed design of the multi-agent data distribution environment (MDDE). We explain in detail the composition and purpose of four components of our environment: stage, registry, benchmark, data nodes. We present and explain in detail the interaction of MDDE components. Furthermore, we present a mechanism for the distribution quality measurement estimation, which is aimed at speeding up the
training of MDRL agents in MDDE. Finally, we present a detailed explanation of the scenario where MDRL agents should directly manipulate data fragment allocation in a fully observable environment with the reward depending on the overall throughput of the system.
4 Implementation

For every project, which relies heavily on the program code implementation, the quality of this implementation is as vital as the conceptual design of the architecture. The source code of our project is intended to be released to the public domain for reference and possibly further research. Maintainability and ease of extension are the highest priorities. We rely on well-established programming languages and libraries, commonly used within the domain of machine learning research and databases, at the core of our project to ensure the highest level of accessibility for the researchers. Additionally, we recognize that the reliance on complex software engineering design patterns or specific application frameworks, while generally improving the architecture, can hinder the ease of program code comprehension. Therefore, we rely on the standard programming language features and common libraries as much as reasonably possible, hence, ensuring that the learning curve for using and extending our codebase is minimal.

In Section 4.1 we describe the key dependencies and reasoning behind implementation choices of the key environment components. Within Section 4.2 we describe the way we have integrated the off-the-shelf MDRL algorithms with our environments and any modifications or corrections of these algorithms.

4.1 Dependencies

In this section, we briefly discuss the dependencies of MDDE. Which programming languages, key libraries were used, and some key deployment considerations. We argue that the implementation specifics are as important as the overall design of the environment. While there is always more than one way to program any given architecture, the choice of the technological stack can make a significant difference in the way any given system performs and its general usability.

4.1.1 Scenario and agents

The stage component of our solution is the component that is intended to be the most volatile and where the researchers would implement most of the changes to the agents, scenarios, reward functions, etc. As such, it is logical to implement it using an interpreted language such that researchers could introduce changes quickly, without the need to rebuild the code every time something is changed. For this purpose, we have chosen standard Python (CPython) as it is currently one of the most popular programming languages amongst both the researchers and practitioners in the field of machine learning. We ensure support for two versions of Python 3.7 and 3.8.
Since Python is an interpreted language, it is easy to modify in-place. However, it is not always practical, especially in cases where the work is intended to be shared with others. In-place changes might introduce incompatibilities with the future versions of MDDE. Therefore, we provide a few points of extension that are specifically meant for the researchers that might choose to introduce their custom agents, scenarios, or data records fragmentation logic. This extension code can be sideloaded along with the MDDE core code via the Python-specific mechanism called implicit namespace packages\(^1\), which allow users to create packages with their custom code, which when loaded functions seamlessly as part of the core MDDE. We believe this functionality is crucial for the reproducibility of research and sharing the results.

Additionally, and perhaps the most important point of extension is integration with DRL frameworks. MDDE provides the environment, the scenarios logic, and the data orchestration mechanisms, but all of this is useless without a DRL framework or an algorithm that interacts with our Python-based stage component. Similar to the other extensions, end-users can supply their code by making use of Python’s implicit namespace packages. These integration packages should serve as an intermediary between a DRL framework and MDDE. There all integration specific additional dependencies can be specified, observation and action spaces can be converted to the shapes suitable for the specific DRL framework. Values for the rewards and any additional parametrization must be done within this package. We provide specific integration examples later in this work, in Section 4.2.

### 4.1.2 Data nodes orchestration

For the registry, we have chosen Java implemented in OpenJDK\(^2\), language level 11. The choice is dictated by the need to execute multiple concurrent functions during the normal operations of the registry, specifically during a benchmark run. We have opted out from basing the entire code base on Python due to the limitations presented in by the standard Python implementation (CPython) where the sequence of execution of relies on a global interpreter lock with limited support for multithreading, which is aimed at alleviating some of the I/O overheads. True concurrency in standard Python can be achieved via multiprocessing, which is a suboptimal solution for our use case, where multiple threads can be created and destroyed rapidly. Java allows flexible multithreading and supports a wide range of multithreading patterns and synchronization mechanisms. At the same time Java is a popular high-level programming language, including in the research community.

Networking is one of the core functionalities that must be reliable and scalable in our implementation of the registry. Java provides extensive low-level API for socket-based TCP programming. However, using this API directly is time-consuming. Since we don’t necessarily require such a fine-grained level of control, which is provided by this API, instead, we rely on a wrapper library Netty [MW15]. It allows us to define our own TCP based protocols and non-blocking API for communication between different components of MDDE architecture.

---

\(^1\)https://www.python.org/dev/peps/pep-0420/ [accessed 2020 July 7]
We provide two types of TCP based APIs: control and benchmark. Control API is designed for sequential execution of data records manipulation and used primarily by the Python-based stage for executing actions, initiating benchmark runs, and retrieving allocation statistics. All communication messages are wrapped in JSON for simplification of consumption in Python code. Benchmark TCP API is less verbose and used by the benchmark runner to request the location of any given tuple and notify the registry when a read operation was completed. Benchmark API designed to operate concurrently, hence multiple concurrent benchmark clients are supported. It is based on command codes and sequenced arguments that are marshaled directly into the binary form without being wrapped into a JSON or similar object.

The key function of the registry is to manage data records and distributed data nodes. Database systems often come with built-in logic for data distribution. Depending on the database, it can be highly sophisticated load balancing or as primitive as simply storing specific key hash ranges on the specific nodes. In both cases, however, the existence of such logic is detrimental to our architecture. We require such a database where we can control data record allocation and replication on the fine-grained level. Additionally, we need the database nodes to be as simplistic as possible in order to avoid skewing results of our own evaluation by built-in heuristics in the database. And finally, we need an in-memory database to ensure the lowest possible overhead for data manipulation and access, to speed up the learning of the agents. Redis DB\(^3\) fits all criteria except the fine-grained data record manipulation in the distributed database cluster.

We utilize Redis as independent data storage nodes where data allocation and replication is tracked and controlled directly by the registry. Meaning each individual Redis node is not aware of the existence of any other nodes. This allows us to exert total control over the data allocation and the ability to collect detailed access information during a benchmark run. Non clustered Redis suits perfectly for the role of data storage in our use case. It introduces no highly sophisticated heuristics controlling allocation, indexing, or caching of data records. Redis is a fully in-memory DB with guaranteed complexity of reads \(O(1)\). An additional benefit for our use case is the fact that all of the incoming queries are executed by Redis sequentially, there is no concurrency even for reads within a single Redis node. We believe this lack of concurrency is beneficial for us during the benchmark run stage because if a single node receives significantly more read requests than the others, these queries will be queued, which in turn should drive the final throughput down.

Because the registry provides total control over the allocation and the benchmark has to inquire about the location of any tuple from the registry, it is possible to supply a custom location logic. It is done by providing a custom implementation of the interface \texttt{IReadOnlyTupleLocator} within the Java code of the registry. Our default implementation simulates a primitive load balancing. This load balancing is based on keeping track of the number of concurrent accesses to any data node. If there are multiple replicas of the same data record requested by the benchmark, the locator will return the node id to the benchmark client, which belongs to the node with the least load at the moment of the read request.

The final vital function of the registry is to control the benchmark and collect the data records read statistics. This statistic is then returned to the Python-based stage, where

\(^3\)https://redis.io/ [accessed 2020 July 7]
it can be used in the learning process of the MDRL agents.

### 4.1.3 Benchmark

In Subsection 3.2.1, we have presented the concept of the benchmark as one of the key components in our architecture. The component is responsible for generation of data and execution of read access workload to the generated data. To fulfill this component of our architecture, we utilize Yahoo! Cloud Serving Benchmark (YCSB) [CST+10]. In this subsection we discuss YCSB in detail based on the original publication and the actual state of the YCSB’s codebase.

YCSB is an open source command-line utility meant for measuring the performance of different database systems. Specifically, YCSB is capable of assessing the performance of basic data related operations: insert, update, read, scan, and delete. It operates in two distinct stages, namely data records generation or ’load’ stage, and data records access also called the ’run’ stage. During the execution of each stage, the throughput of the database is measured and returned back to the user.

YCSB is parameterized with command-line arguments. YCSB has two mandatory arguments: database client and the name of the workload configuration file. The client must be selected according to the database in which performance is measured. YCSB has a number of pre-defined clients, covering a number of popular databases, including Redis. Each client implementation might require additional YCSB arguments, in which case these must also be supplied as command-line arguments when YCSB is invoked.

Workload configuration is a file that allows configuring specifics of the benchmark run. Examples of such configurations can be found in Appendix Section C.. Specifically, it determines the exact number of data records that must be generated for the benchmark. Likewise, this configuration determines the number of benchmark operations that

![YCSB core workload main components](image)

**Figure 4.1:** YCSB core workload main components
must be performed during the benchmark run stage, as well as the proportions of reads, writes, scans, and updates within this number of operations. YCSB defines four workload types: core, time series, rest, and constant occupancy. In the context of measuring data distribution for random read access, only the first is relevant. Core workload is used to generate typical CRUD operations. Figure 4.1 depicts YCSB components that are relevant for the core workload within our architecture. Time series is a special case of core workload dealing with time series related scenarios. Rest is a specialized workload meant to measure the performance of the HTTP based RESTful web services, and not usable with any specific database directly. Constant occupancy workload is designed to measure how application performance is affected by data fragmentation on disk [SvI06].

The important workload configuration parameter is the data access distribution. YCSB Core workload supports six distributions. Uniform (Figure 4.2a) distribution uniformly distributes the probabilities of any specific generated data record to be selected for access. Zipfian (Figure 4.2b) creates a distribution where a small number of records are going to be selected disproportionately more often than the rest, creating a shorthead-longtail access pattern. Distribution under the name of latest (Figure 4.2c) is similar to the Zipfian distribution, but instead of randomly selected the hot-spot data records, it favors the ones that were inserted the last. Hot-spot distribution allows specifying such access pattern where a specific percentage of the total operations access a specific percentage of the total number of data records (e.g., 80% of requests are performed on 20% of records). Sequential distribution generates a distribution where each data record is accessed only once. Exponential (Figure 4.2d) distribution allows simulating an access pattern where the most accessed data record is exponentially more popular as the second one, and the second is exponentially more popular as the third, and so on. Note, exponential distribution is not mentioned in the original publication [CST+10] but present in the code.

It is possible to configure distributions of the data records field values within the workload. Specifically, for core workload, the following field length distributions are available: uniform, Zipfian, constant, histogram. Uniform and Zipfian distributions accept lower and upper bounds for the field lengths as parameters. As the name suggests, uniform distribution generates a uniform distribution of field lengths across all of the fields for a key. Zipfian distribution, similarly to the one in key access distribution, generates field lengths for a key as the shorthead-longtail distribution within the specified upper and lower bounds. Constant distribution generates all of the keys with the specified constant lengths. Histogram distribution allows generating fields for a key, each with a specific pre-defined length that is retrieved from a file supplied by the user.

YCSB can be configured for read-only benchmark workloads, which is exactly the functionality we need within our experimental setup. Coupled with the built-in ability to generate different data access distributions that make it possible for us to test the adaptability of the trained agents, YCSB becomes a good fit for our architecture. However, we cannot use the default implementation of YCSB as we need it to be able to integrate with the registry. Therefore, we are providing a modified version of the default YCSB Redis client⁴ implementing needed for it to functions effectively within our architecture. Specifically, we provide a custom implementation of the «DB» Specific database client component

⁴https://github.com/akharitonov/YCSB/tree/redis-mdde-client
depicted in Figure 4.1. Our custom database client contains MDDE specific components as depicted in Figure 3.2, Benchmark component.

We are not introducing any changes in the core YCSB functionality to make sure that the custom client we make, can be easily transferred between existing and future versions of YCSB.

### 4.1.4 Deployment

Depending on the scenario, the number of agents, and the volume of test data, MDRL learning outcome might become affected by the way the environment infrastructure is configured and deployed. The Python-based stage, the registry, and the data nodes can all be deployed separately or within the same system. In our default setup, communication between these components is performed via a TCP connection. This allows us to deploy each component on the system most suitable for the task. For example, MDRL algorithms are often computationally expensive and often make use of GPU. Therefore, it makes sense to deploy the Python-based stage on the system with the highest performant CPU or multiple GPUs.

The registry, on the other hand, does not require GPUs or a high-frequency CPU but will benefit from the higher amount of CPU cores during the benchmark run, especially when the benchmark runner is configured with a high number of concurrent clients. Since both the registry and the benchmark runner are multi-threaded applications deployed together, a higher number of CPU threads in the system will reduce the amount of time needed for a benchmark to execute. Running multiple simulated clients in a system with a low number
of CPU threads available will affect the end result, and potentially MDRL learning process because CPU becomes a performance bottleneck.

When it comes to data nodes, within our default setup, these should be deployed on a system with enough RAM to store the dataset generated by YCSB, as we are relying on Redis DB, an in-memory database. It is also might be beneficial to deploy data nodes away from the environment. Even though Redis reads are not a CPU intensive operation and MDRL algorithm is not performing calculations while the benchmark is running, RAM capacity might become an issue. Some algorithms, are memory intensive, especially with a high number of agents, including MADDPG, which we discuss in Subsection 2.5.1.

Database related single-agent environments in Park [MNN19], are relying on real-world databases similarly to our own work, instead of using simulation or a cost model. However, there is no need for the researchers to manually set up and configure the underlying database for Park environments. Instead, internally Park uses containerized with Docker versions of the databases. In addition to alleviating the complexity of configuration, usage of containers also contributes to the reproducibility of research [Boe15]. We do not integrate Docker into our architecture directly, as Park does, because there is a variety of container software solutions besides Docker and we want to retain the ability to deploy the entire infrastructure needed for the environment directly into the system. However, it is perfectly possible to use Docker to deploy MDDE infrastructure, and we provide configurations for Docker, which were used in our own experiments.

### 4.2 Integration

The environment on its own is not very useful and must be designed to be integrated with one or more MDRL algorithms. The Python-based part of our architecture, the stage, is designed to serve as a wrapper for an MDRL algorithm. As explained in Section 3.2.1, it is achieved by exposing actions, observations, and rewards. This architecture decision provides the researchers with all of the functionality needed for integrating any MDRL algorithm that conceptually fits the Markov process.

However, the fact that the algorithm follows the Markov decision process for its agents does not mean that observations, rewards, and actions will be requested and processed identically in every algorithm. In this case, we account for this fact by providing the researchers with the well defined, stable API, which can be adapted to the needs of any of any specific algorithm and its implementation. Researchers need to provide an intermediary class that would convert the data structures and types returned by the environment into the form that can be consumed by the evaluated algorithm. We do not rely on or enforce the use of a specific DRL-related library such as specific version of OpenAI’s Gym.

In our environment, we attempt to achieve reasonably wide compatibility with third party algorithm implementations. This is achieved by the heavy reliance on the data structures and functionality provided in the standard Python. In addition, we used arrays and

---

5https://www.docker.com/ [accessed 2020 July 7]
platform-independent data types provided by NumPy library [WCV11], which provides a stable API and transparent development model.

4.2.1 Ray RLlib

At its core, Ray is a framework aimed at simplification of creating and deploying distributed applications, specifically written in Python. Discussion of the exact mechanisms behind the framework implementation are beyond the scope of this work. However, Ray ships with RLlib [LLM+17] module, which contains implementations of several DRL algorithms. These algorithms are implemented in the way benefiting from Ray’s inherent capability to distribute computations. As noted by the authors [LLM+17], the reinforcement learning process consists of highly irregular computations, during both policy learning and action execution stage. These computations can benefit from parallelization and distribution, Ray’s RLlib facilitates this by alleviating the need for the MDRL researchers to implement complex custom distribution infrastructures for every new algorithm implementation. It is also worth mentioning, in the context of MDRL experiments, that Ray contains a module named Tune [LLN+18]. This module can be used for automatic hyperparameters tuning in DRL algorithms implemented by Ray, in an attempt to achieve the best performing combination of such for the combination of the algorithm and the environment.

At the time of writing, the latest stable version of Ray is 0.8.4. This version or Ray ships with several single-agent DRL algorithm implementations, including DQN, PG, DDPG, and PPO. While these can be considered to be the fundamental algorithms in the modern DRL research, these are not designed for multi-agent scenarios. However, as already mentioned in Subsection 2.2.2, these algorithms can be employed in multi-agent environments. While this practice is riddled with numerous challenges, as previously mentioned in Subsection 2.2.1, it might be prudent, in future work, to evaluate and compare MDRL algorithms and their performance in any given scenario against single-agent DRL algorithms within the same scenario. Ray’s RLlib supports this setup by default by providing a multi-agent interface allowing to execute most of the single-agent DRL algorithms, that it implements, in a multi-agent scenario. We demonstrate the ability to run a single-agent DRL algorithm in multi-agent environment in Subsection 2.5.3, by executing DQN in multi-agent particle environment scenarios. Additionally, RLlib contains an implementation of MADDPG, an algorithm which is multi-agent oriented by design.

The wide range of the provided algorithms and clear suitability for MDRL research makes Ray RLlib a well suited candidate for experimental integration with our environment for evaluation.

Integration of Ray RLlib with MDDE

Ray’s multi-agent oriented interface implements a typical pattern of interaction with an environment by requiring the latter to expose three basic methods: observe, step, and reset. Additionally, the interface requires pre-defined mapping of agents to their actions created before the environment is initialized, these are retrieved from the action space property of the MDDE stage instance.
The composition and shape of the action and observation spaces might be dependent on the algorithm implementation in Ray, but generally, all are Gym compatible, meaning these expect action and observation spaces wrapped into the Gym specific data structures. This does not require any changes in the core codebase of MDDE and instead, done in the interface layer code, a single Python file responsible for the conversion of data types for the correct interaction between MDDE and Ray. This layer can be easily customized by the researchers when needed. For example, MADDPG implementation relies on the Gym.Box for the observation space and Gym.Discrete for action space of every agent. Additionally, Gym.Box can also be used as a container for action space, disregarding Gym.Box’s indented use case.

Invalid actions elimination

As indicated in Section 3.2.4, the validity of actions in the action space of our scenario is highly nuanced and depends heavily on the current state of the environment. Such action space design results in a situation where a large subset of the actions, that belongs to the agent, is invalid in most of the states. Our preliminary experiments with MADDPG, have revealed that MADDPG having difficulties with discerning actions validity based on the state of the environment, which led to extreme sample inefficiency.

To mitigate this issue, we have implemented an approach that allows us to filter out undesirable actions via the embedding of a binary *action mask* to the actor’s neural network output layer while keeping critic unchanged. This approach is present in some of the single-agent algorithms implemented in Ray RLLib\(^7\), but not in MADDPG. Masking out illegal or incorrect actions based on the state was suggested to be a beneficial in some cases by practitioners in the field of DRL [VEB\(^+\)17]. MDDE scenario must form a binary map of valid (1) and invalid (0) actions, then return it together with the observation to Ray RLLib MADDPG. The network which receives this action mask as an embedding then computes a product of the action mask embedding and the output layer of the network, which effectively filters out the invalid actions as the output probabilities for these become zero. The general conceptual idea is presented in Figure 4.3, where input is the current state observation, and output is the probabilities of actions.

The mask formation increases the overhead of the observation computation at every step, which must be performed by the scenario in MDDE. Action mask is calculated via heuristic pre-defined rules that outline the basic constraints of MDDE, such as the impossibility to remove a unique replica or attempts to copy a fragment from a node that does not contain a copy of it. As a trade-off, we expect a significant increase in sample efficiency for MADDPG as the agents no longer select invalid actions intentionally.

However, the mask is calculated based only on the current state and does not take into account the policies of the agents. This fact might still lead to the occasional attempt by the agents to execute conflicting actions. For example, this might happen when two agents have the only two replicas of the same fragment, and both decide to delete it.

in the same step, at which point the action will be declared illegal by MDDE and canceled.

The described above mechanism is achieved through patching Ray RLlib’s implementation of MADDPG directly in the local Python environment\(^8\). This simplifies the execution of our examples and does not require rebuilding Ray fully.

### 4.2.2 MAAC

Unlike MADDPG implementation, which is provided and supported by an established specialized DRL framework, MAAC implementation used in this work is a stand-alone experimental code provided\(^9\) by the authors of the algorithm alongside with the publication [IS18].

Unlike the implementation of MADDPG, that we use in our evaluation, MAAC is a DRL framework independent, stand-alone algorithm implementation, and does not provide a public interface for an environment integration. We circumvent this limitation by providing a Stage module wrapper for the reference MAAC implementation\(^10\), translating the provided implementation algorithm calls step, observation, and rewards into the form

---

\(^8\)https://github.com/akharitonov/mdde/tree/maddpg_act_mask_v2 [accessed 2020 July 7]


compatible with MDDE. We do not modify the core of the reference MAAC implementation of the algorithm provided by the authors and fully rely on the original algorithm’s source code.
5 Evaluation and Results

In this chapter, we present the results of the empirical evaluation and discuss the most notable observations. Three sections of this chapter correspond to the three research questions stated in Section 3.1.

1. In Section 5.1 we provide the results of evaluating the effects of different hyperparameters on one of the chosen off-the-shelf MDLR algorithms. For this purpose, we employ a reference multi-agent environment MPE, discussed in detail in Subsection 2.3.1.

2. Section 5.2 is dedicated to the comparison of the real performance measurement and our estimation mechanism, as presented in Subsection 3.2.2. We discuss the observed shortcomings of relying on the performance measurements based on YCSB. We evaluate the estimation in two carefully designed edge-case scenarios of data distribution. Additionally, the evaluation we perform in this section demonstrates the soundness of the design and the implementation correctness of our own data distribution multi-agent environment, with the pre-defined heuristic rules.

3. Finally, in Section 5.3, we describe the evaluation setup based on our multi-agent data distribution environment, which is designed and implemented within this work. We provide the results of evaluation with the selected off-the-shelf MDRL algorithms, as well as discussion of the observations.

We perform all of the following evaluations on the hardware platform with the specifications described in Table B.5.

5.1 Evaluation of hyperparameters influence

The tuning of hyperparameters is an important step before starting the evaluation of the environment itself with the chosen algorithms. It can only be efficiently performed by running the chosen algorithm within the environment and later analysis of the results. Therefore, the objective is to run multiple runs for each hyperparameter, which is mentioned in Subsection 2.1.4, and then comparing the results. The point of this evaluation is to attain empirical evidence of the hyperparameters influence on MADDPG algorithm. It’s worth noting, however, that multi-agent particle environment is a simplistic environment based on deterministic rules, which is hardly representative of a real system of any kind, database or otherwise. Real system are influenced by a number of outside factors that make their operations non-deterministic. None of this possible outside influences are part of evaluation we perform for our own environment. Therefore, we believe it’s appropriate to investigate the influence of hyperparameters on a simplistic environment such as MPE, where running repeating experiments is cheap, before moving on to our own environment.
Learning rate

We start our evaluation of hyperparameter influence with the learning rate. We perform evaluation of learning rate at the following values: 0.001, 0.01, 0.1, 1, 10, 100. Plots for the results for all standard MPE scenarios are presented in Appendix Subsection B.1.

The results are as expected. The further learning rate values below 1 the more gradual policy updates become. The lower the value the more careful agent’s behavior becomes when it comes to adjusting for the environment changes and to behaviour of other agents. Since MPE scenarios do not offer any sudden and lasting changes in the environment that are unexpected by the agent, lower values are clearly beneficial for the agent’s learning stability. Note, the latter two values, 10 and 100, are hardly reasonable in most cases as these cases the agent to override its policies with any new experiences. This results in unstable learning in both cooperative and competitive scenarios.

Consistent with the original publication [LWT+17], a degree of the reward oscillation is observed in the adversarial scenarios as competing agents adapt to each other’s behavior. This problem is clearly exaggerated by a high learning rate as agents attempt to adapt to each other, but the changes in the policies are too drastic. However, it is also clear that a high learning rate hinders the ability of the agents in the cooperative scenarios to attain higher rewards by working together. This is especially evident in the 'cooperative communication' scenario.

Replay buffer

Reply buffer is essentially a pool of collected past experiences that are randomly accessed by the agent and reused in learning. Basically, past experiences are used to adjust weights in the current agent’s neural network.

The length of the replay buffer should be chosen according to specifics of the scenario. Specifically, it depends on the relevance of old experiences. If the environment and behavior of other agents are volatile, it is likely that old experiences might not reflect the current state of the environment. We evaluate scenarios with three values: $1 \times 10^6$, $1 \times 10^5$, $1 \times 10^4$. The highest value of the replay buffer length value, out of three, is the default length proposed by the authors of MADDPG. We validate only with smaller variations because we believe that increasing replay buffer even further above $1 \times 10^6$ is impractical due to the sheer main memory requirements. The results of this evaluation are presented in Appendix Subsection B.2.

Most of the MPE scenarios offer little variability and no complex behaviors of the agents, which means that most of the possible state conditions are discovered early. Therefore these saved experiences are generally representative of the scenario and the length of the replay buffer is of little consequence. The only exception is 'covert communication' scenario. It is clear that 'covert communication' learning becomes unstable with smaller than default length of the replay buffer. We attribute this behavior to the nature of the scenario where 'encoding' of the messages has a limited pool of possible variations, so the longer buffer allows to store and reuse more past variations of such encoding. Therefore 'covert communication' benefits from a larger pool of collected experience in
the replay buffer but only because eventually the adversary collects observations of all possible encoding variations.

**Discount factor**

We validate discount factor $\gamma$ with following values: 0.95, 0.75, 0.5, 0.35, 0.15. Values closer to 1 in principle should make the agent consider future reward above the immediate reward. Such that an agent might take a suboptimal step in order to achieve a more favorable state where the agent will be able to take a step yielding a higher reward. Values closer to zero should make the agent favor the most optimal immediate step returning the highest reward immediately, disregarding following steps.

Our evaluation indicates that MPE is poorly suited for the evaluation of this specific hyperparameter. None of the scenarios have a potential for strategic planning that can be taken by the agents. The environment is designed for short episodes (default is 25 steps per episode) where the agent’s correctness of action is judged immediately, there is no opportunity for developing complex strategies. This is also clearly reflected in our evaluation, plots for which can be found in Appendix Subsection B.3.

**DNN weights adjustment**

MADDPG partially draws inspiration from DQN, which operates with two deep neural networks. One online and one at rest. Any changes in the network itself are represented in the form of adjusting specific numeric values, called weights. While MADDPG agent makes steps and receives observations and rewards feedback, it adjusts weights in its online DNN. When this DNN is synchronized to the network at rest, the network at rest isn’t necessarily completely overridden. Instead, it’s possible to control the degree to which the saved network is updated in accordance with the online network. It is achieved by using hyperparameter $\tau$ (tau). Note, in the case of MADDPG, the difference between learning rate and $\tau$ is that the former affects the Adam optimizer [KB14] while the latter controls the degree of overriding the stored network in the critic.

We perform evaluations with four different values of $\tau$: 1, 0.1, 0.01, 0.001. $\tau = 1$ overrides the entire stored network’s weights with the values from the online network, while 0.001 is the most gradual update degree that we evaluate. Value $\tau > 1$ is generally impractical, and $\tau = 0$ make no sense. In the first case, the degree of update would be artificially too radical, while the latter case causes the saved network never to update. Plots corresponding to the evaluation of this hyperparameters are displayed in Appendix Subsection B.4.

The results of the test runs are as expected. Value of $\tau = 1$ in every MPE scenario makes learning unstable because the saved network weights are overridden completely, which results in the critic’s stored DNN to adjust too drastically. However, fully cooperative scenarios exhibit less instability in comparison to the adversarial. Cooperative navigation (Figure B.39) and cooperative communication (Figure B.38) scenarios are able to attain a higher final reward regardless of the $\tau$.

Adversarial scenarios are more sensitive to $\tau = 1$. Physical deception (Figure B.37) and Predator-prey (Figure B.41) clearly indicate the detrimental effect of the full network
override in adversarial scenarios. Adversaries display the tendency to fail in adapting to 'good' agents behavior. Opposite of that, in the adversarial scenario named covert communication (Figure B.42), good agents are unable to adjust to each other, instead of the adversary.

There is generally no notable difference in results for $\tau \in \{0.1, 0.01, 0.001\}$. A single notable exception is covert communication. In this scenario, values other than the default 0.01 can lead to highly oscillating rewards or agents not learning the correct behavior as well. It is, however, consistent with the original publication, where authors explicitly mention the volatile nature of this scenario even with the default values. It’s clear that covert communication is the scenario that requires a very precise tuning of the hyperparameters.

**Multi-step learning**

We evaluate the effects of multi-step learning, where DNN weights are not necessarily updated at every step, with the following values: 1, 5, 10, 15. First is the reference default value, where agents update their respective neural networks at every step. The rest of the values force the agent to execute 5, 10, or 15 steps before adjusting the weights in the networks. This means that the agent will no react to the changes in the behavior of the other agents instantly. Instead, agents collect observations and reward for a number of steps before judging the correctness of their steps.

We did not observe any effect on the learning performance for physical deception (Figure B.43), cooperative communication (Figure B.44), and keep-away (Figure B.46) scenarios. Results for cooperative navigation (Figure B.45) and predator-prey (Figure B.47) suggest the slower pace of learning the behavior.

As with the previous hyperparameters, covert communication (Figure B.48) exhibit a high degree of learning instability at values different from the default 1. That is especially evident with values 10, 15, where even in the limited number of experimental runs, the results across different experimental runs are drastically different, to the point where agents are unable to learn the correct behavior over the course of the experimental run.

**Summary**

We have evaluated MADDPG in the set of the reference environment scenarios provided by multi-agent particle environment, with a variety of hyperparameters. The results we have observed suggest that MADDPG is, in general, well suited for cooperative scenarios. We did not observe a critical collapse or instability of learning in cooperative scenarios with all reasonable hyperparameter values.

As a result of this evaluation, we received empirical confirmation of the expected behavior for the hyperparameter changes. More importantly, we have discovered that a lower learning rate 0.001 is preferred for cooperative scenarios. Longer replay buffer can be beneficial but depends strictly on the scenario. The discount factor is largely irrelevant.
within the context of the evaluated scenarios. Q-learning related DNN weights adjustment $\tau$ generally beneficial at lower values 0.001 in both cooperative and adversarial scenarios.

5.2 Benchmark estimation evaluation

We evaluate our benchmark estimation mechanism, which is presented in Subsection 3.2.2. The goal of this evaluation is to see how well it stacks up against the benchmark metric results of YCSB. We make an evaluation with two edge-case scenarios functioning according to pre-defined rules, without the involvement of MDRL agents.

First evaluation edge-case is, which we name "starving" case, is a simple scenario where four agents $W = \{w_0, w_1, w_2, w_3\}$ each start with five unique fragments replicas, 20 fragments $|F| = 20$ in total. We use YCSB workload configuration with uniform read distribution, 10000 records and 100000 reads. Agents perform 16 steps. At every step, agent $w_3$ iterates over the list of fragments replicas of which it does not yet have $\{f \in F | f \notin F_{w_3}\}$, where $F_{w_3}$ is a set of fragments allocated on agent $w_3$. Then, when such replica is discovered, $w_3$ copies a fragment from an agent $\{w \in W | w \neq w_3\}$ that posses this replica. At the same time, each agent, other than $w_3 \{w \in W | w \neq w_3\}$, iterates over the list of fragments that are currently allocated on it $f \in F_w$, determines if $w_3$ already has a replica of this fragment and if so, deletes this fragment from self. In the final state, $w_3$ has all of the fragments, while all other agents are empty. The final state of the "starving" edge-case is the worst case state which can possibly happen in our scenario, because all the read workload queries will be concentrated on a single node, while all of the other nodes would not contribute at all.

The second edge-case is the scenario where all agents are "greedy". Similarly to the first edge-case, four agents each start with five unique fragments replicas, 10000 records, and 100000 reads. However, this time the read distribution is latest, such that at the start, there is one agent $w_3$, which holds all of the hottest fragments. Unlike the "starving" edge-case, the agents, in this case, execute 15 steps. At every step, each agent $w \in W$ gets a copy of the fragment it does not yet possess $\{f \in F | f \notin F_w\}$. Fragments are copied in the order corresponding to their index, from lowest to highest. In the final state, each agent has copy of each fragment. Greedy case is clearly the best from the read efficiency standpoint if all nodes are equally fast and accessible for the agents, because all of the fragments are allocated on all of the nodes. In such case read queries can be easily distributed across all of the nodes. This, however, comes at a cost of storage.

In both of the cases we configure YCSB to read data records in their entirety. Only read queries are performed, no record modifications, additions or deletions are allowed during the workload execution. YCSB emulates 50 concurrent clients (threads). Total number of fragments formed is 20 across all of the nodes. All of the evaluation runs are executed on a single machine with the specifications outlined in Appendix Table B.5. Components of the MDDE infrastructure are executing in a Docker container, as described in Subsection 4.1.4.

We believe these two edge-cases are representative of the two expected throughput behavior patterns. In the case of the "starving" agents, we expect throughput to diminish
gradually as more fragments are concentrated on the single node, therefore increasing the load to that node. Higher load concentrated on a single node should increase the amount of time it needs to process incoming queries because more of these would come to the same node at any given moment of time. Following the same logic, the "greedy" agents should see a rapid rise in throughput values when agents get copies of the hot fragments, thus distributing the load among themselves.

5.2.1 Preliminary observations

Initial execution of the estimation edge-case scenarios indicated a significant hardware dependence of our evaluation mechanism. This dependence reveals itself in two specific challenges: shared resources problem, and MDDE registry overhead. These two challenges are especially evident when the entirety of the MDDE infrastructure runs within the same machine, instead of being distributed in a cluster.

MDDE registry overhead

During the benchmark run, each YCSB client fist has to access MDDE registry to find the fragment’s location and then retrieve the fragment from the data node. YCSB client access the registry via a TCP connection, then the registry executes internal logic deciding which data node YCSB client must access to retrieve the fragment and the returns it to YCSB. Only after this, YCSB can access data node, Redis in our case, and read the data. It is clear that such interaction sequence results in a significant overhead from the registry. When the cost of accessing the registry is higher than reading the data, the results of the benchmark are skewed.

Additionally, to receive meaningful metrics from YCSB we require a sufficient number of emulated clients, which in our setup is set to 50. This means that when YCSB executes a workload, it operates with a pool of 50 threads. To respond to the YCSB client requests, 50 (one per YCSB client thread) corresponding threats are created in the registry, as it coordinates the routing of YCSB reads. Each Redis node is processing requests in a single-threaded mode and requires additional threats to handle incoming network connections and queuing them up. Without a learner, MDDE stage instance takes an additional single thread. When all of the MDDE components are deployed locally, or more than one instance of MDDE is running, it can easily result in high CPU thread contention.

Both of the aforementioned challenges are evident in the situation where the cost of data retrieval from data nodes is low. With our basic workloads (Listings C.1 and C.2), we have observed both of the issues. The result of this situation was that the YCSB throughput metric did not change regardless of the distribution of the fragments. Record size in these workload configurations is small. Each record contains only ten fields, 100 characters long each. This means that the cost of retrieving a record from a Redis node is negligibly low in comparison to the MDDE registry overhead and the cost of switching a CPU context.

In order to mitigate this issue, we have modified our initial workloads by increasing the number of fields per data record and dramatically increasing the field length. We have
increased the number of fields from the default 10 to 100 per data record. Each field length was increased from the default 100 characters to 3000 characters. Using these new workloads (Listings C.3 and C.4) has allowed us to mitigate the thread contention by dramatically increasing the cost of reading data records, essentially shifting focus from CPU to I/O bound operations.

This change in workloads configuration resulted in the expected behavior where data nodes are queuing up incoming read queries, which now took a sufficient amount of time to execute. However, this workload modification comes at the cost of increased RAM consumption by each Redis data node, as well as a significant increase in the time needed to execute the workload.

It is worth noting that the presence or absence of an MDRL framework is inconsequential in the context of the discussed CPU contention challenge. Benchmark is executed when MDDE stage processes step signal received from the MDRL agent. While this happens, the agent is dormant as it awaits the reward and observation feedback from MDDE. There is no scenario in MDDE when MDRL agents perform calculations in parallel to running the benchmark.

**Shared resources**

MDDE infrastructure running within a single machine already requires a significant amount of CPU time and threads. Any additional process running within the same OS is capable of causing YCSB throughput measurement to become misleading and potentially might negatively affect the learning process of MDRL agent. It’s possible that the quality of distribution is increasing with the latest steps taken by the agents but throughput returned by YCSB is instead lower because of there is a process in the system that competes for CPU and main memory.

For example, agent actions should logically lead to increased throughput and, by proxy, reward. However, the opposite might happen simply because, during the execution of YCSB benchmark workload, a process in the OS reserved a portion of CPU threads. This leads to the reduced performance of the registry, Redis, and YCSB.

It is not possible to mitigate this issue without fully reserving the machine for the running experiment and disabling all scheduled internal OS processes that might potentially take up resources during the YCSB workload execution.

It is important to note. The challenge of the shared resources stems not from the design flaw of our environment but from its intention. Our multi-agent data distribution environment is designed in a way that leverages real-world infrastructure. Therefore, the ideal deployment scheme and the intention is where the stage with an MDRL framework is deployed on a specialized server. The registry is deployed separately on an exclusive server. At the same time, data nodes are distributed over different types of server and geographical locations. In this case, the MDRL algorithm that is capable of solving the environment should be able to balance data distribution in a way that accounts for differences in the properties of data nodes. However, when the entirety of the infrastructure is deployed on the same machine, even in container form, there can be no significant differences in properties among the data nodes.
5.2.2 Metrics estimation evaluation results

In this subsection we present the results of executing our 'greedy' and 'starving' edge cases. Each evaluation case is executed 11 times. The host machine at the time of execution was fully reserved for this specific task allowing to avoid the aforementioned problem of sharing the resources. Hyperparameters for Equation 3.7 were chosen as follows: \( j = 0, k = 0.7 \).

For each of the edge-case evaluation we provide the following observations:

1. Heatmaps depicting the number of reads per fragment, per node. Figure 5.1 for starving and Figure 5.3 for greedy. These heatmaps contain common starting state generated by YCSB and final states for YCSB and our estimation. In both of the evaluated edge-cases, the final read count distributions are similar for both YCSB and estimation. This means that the final read distribution prediction for worst and best cases are comparable for both real-world YCSB benchmark in our evaluation setup and the estimation mechanism.

2. Tables with clear distribution changes progression per step. We provide one table for the reads generated by YCSB and for comparison table displaying the results of our workload estimation. First line in each table is the initial distribution generated by YCSB workload. Zero values signify the absence of the fragment on the corresponding node at the step.

   - Starving
     YCSB: Table 5.1
     Estimation: Table 5.2

   - Greedy
     YCSB: Table 5.3
     Estimation: Table 5.4

We observe a clear dissimilarity of the read distribution estimation and YCSB in the greedy edge-case. This dissimilarity decreases the closer the scenario gets to the final state. However, in the starving edge-case scenario, we do not observe a similar crucial difference in read distribution estimation and YCSB.

3. Plots comparing the mean throughput across 11 runs. Figure 5.2 depicts the 'starving' edge-case throughput, while Figure 5.4 is dedicated to the 'greedy' one. Thin lines signify the degree of throughput measurements deviations from mean within the individual executions. We observe comparable trends between YCSB and the estimation, while the absolute throughput values have a degree of difference. Our estimation mechanism is not predicting the exact YCSB throughput values but an approximation with focus on the direction of change accuracy.

Note, the exact numbers of fragments reads will be different in each real-world benchmark run because YCSB is not deterministic in generating the number of reads per data record. Heatmaps and tables in this section are a sample representing only a single, first run from 11 that were executed for this evaluation. We observe similar results throughout all of the executions with no outliers.
Figure 5.1: Allocation change in the "starving" edge-case
Table 5.1: "Starving": Progression of reads per node (YCSB)

<table>
<thead>
<tr>
<th>Step</th>
<th>w0</th>
<th>w1</th>
<th>w2</th>
<th>w3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24913</td>
<td>25059</td>
<td>24948</td>
<td>25080</td>
</tr>
<tr>
<td>0</td>
<td>23370</td>
<td>24993</td>
<td>24855</td>
<td>26782</td>
</tr>
<tr>
<td>1</td>
<td>19833</td>
<td>24910</td>
<td>24942</td>
<td>30315</td>
</tr>
<tr>
<td>2</td>
<td>15176</td>
<td>25012</td>
<td>24949</td>
<td>34863</td>
</tr>
<tr>
<td>3</td>
<td>10063</td>
<td>25028</td>
<td>24995</td>
<td>39914</td>
</tr>
<tr>
<td>4</td>
<td>5045</td>
<td>25229</td>
<td>24781</td>
<td>44945</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>25253</td>
<td>24749</td>
<td>49998</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>24883</td>
<td>19964</td>
<td>55153</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>24944</td>
<td>15195</td>
<td>59861</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>24834</td>
<td>10177</td>
<td>64989</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>25143</td>
<td>5011</td>
<td>69846</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>25164</td>
<td>0</td>
<td>74836</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>20034</td>
<td>0</td>
<td>79966</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>15117</td>
<td>0</td>
<td>84883</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>9825</td>
<td>0</td>
<td>90175</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>5015</td>
<td>0</td>
<td>94985</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100000</td>
</tr>
</tbody>
</table>

Table 5.2: "Starving": Progression of reads per node (Estimation)

<table>
<thead>
<tr>
<th>Step</th>
<th>w0</th>
<th>w1</th>
<th>w2</th>
<th>w3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24913</td>
<td>25059</td>
<td>24948</td>
<td>25080</td>
</tr>
<tr>
<td>0</td>
<td>22695</td>
<td>25059</td>
<td>24948</td>
<td>27298</td>
</tr>
<tr>
<td>1</td>
<td>18248</td>
<td>25059</td>
<td>24948</td>
<td>31745</td>
</tr>
<tr>
<td>2</td>
<td>13804</td>
<td>25059</td>
<td>24948</td>
<td>36189</td>
</tr>
<tr>
<td>3</td>
<td>9311</td>
<td>25059</td>
<td>24948</td>
<td>40682</td>
</tr>
<tr>
<td>4</td>
<td>4812</td>
<td>25059</td>
<td>24948</td>
<td>45181</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>25059</td>
<td>23510</td>
<td>51431</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>25059</td>
<td>18828</td>
<td>56113</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>25059</td>
<td>14168</td>
<td>60773</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>25059</td>
<td>9589</td>
<td>65352</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>25059</td>
<td>4989</td>
<td>69952</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>23990</td>
<td>0</td>
<td>76010</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>19190</td>
<td>0</td>
<td>80810</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>14422</td>
<td>0</td>
<td>85578</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>9646</td>
<td>0</td>
<td>90354</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>4957</td>
<td>0</td>
<td>95043</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100000</td>
</tr>
</tbody>
</table>

Figure 5.2: Throughput dynamic comparison in the "starving" test case

5.2.3 Summary

A clear disadvantage of our benchmark estimation is its deterministic nature. This is especially evident in the starving scenario where estimation reads remain the same per fragment while YCSB introduces a degree of variability. Additionally, the load balancing of our estimator does not take into account the temporal factor of the incoming reads. That shortcoming is resulting in the read distribution that is significantly different from YCSB in the 'greedy' case in steps 5-11 (Tables 5.3 and 5.4).

Decreasing throughput estimation in the "starving" scenario closely resembles the trend of
Figure 5.3: Allocation change in the "greedy" edge-case
Table 5.3: "Greedy": Progression of reads per node (YCSB)

<table>
<thead>
<tr>
<th>Step</th>
<th>$w_0$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2837</td>
<td>4264</td>
<td>7193</td>
<td>85706</td>
</tr>
<tr>
<td>1</td>
<td>2749</td>
<td>4428</td>
<td>7122</td>
<td>85701</td>
</tr>
<tr>
<td>2</td>
<td>2660</td>
<td>4620</td>
<td>7161</td>
<td>85559</td>
</tr>
<tr>
<td>3</td>
<td>2502</td>
<td>4760</td>
<td>6985</td>
<td>85753</td>
</tr>
<tr>
<td>4</td>
<td>2353</td>
<td>5012</td>
<td>6978</td>
<td>85657</td>
</tr>
<tr>
<td>5</td>
<td>10242</td>
<td>8763</td>
<td>15754</td>
<td>67444</td>
</tr>
<tr>
<td>6</td>
<td>24364</td>
<td>8763</td>
<td>15754</td>
<td>67444</td>
</tr>
<tr>
<td>7</td>
<td>29699</td>
<td>11310</td>
<td>29423</td>
<td>29568</td>
</tr>
<tr>
<td>8</td>
<td>28847</td>
<td>12804</td>
<td>29314</td>
<td>29035</td>
</tr>
<tr>
<td>9</td>
<td>27969</td>
<td>13954</td>
<td>29150</td>
<td>28927</td>
</tr>
<tr>
<td>10</td>
<td>25844</td>
<td>22906</td>
<td>25333</td>
<td>25917</td>
</tr>
<tr>
<td>11</td>
<td>25079</td>
<td>24838</td>
<td>24982</td>
<td>25101</td>
</tr>
<tr>
<td>12</td>
<td>25426</td>
<td>25263</td>
<td>24556</td>
<td>24755</td>
</tr>
<tr>
<td>13</td>
<td>24830</td>
<td>24960</td>
<td>24906</td>
<td>25304</td>
</tr>
<tr>
<td>14</td>
<td>24855</td>
<td>24912</td>
<td>24917</td>
<td>25316</td>
</tr>
</tbody>
</table>

Table 5.4: "Greedy": Progression of reads per node (Estimation)

<table>
<thead>
<tr>
<th>Step</th>
<th>$w_0$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3855</td>
<td>4301</td>
<td>6189</td>
<td>85655</td>
</tr>
<tr>
<td>1</td>
<td>4675</td>
<td>4326</td>
<td>5317</td>
<td>85682</td>
</tr>
<tr>
<td>2</td>
<td>4895</td>
<td>4820</td>
<td>4575</td>
<td>85710</td>
</tr>
<tr>
<td>3</td>
<td>4912</td>
<td>4839</td>
<td>4515</td>
<td>85734</td>
</tr>
<tr>
<td>4</td>
<td>4692</td>
<td>4857</td>
<td>4695</td>
<td>85756</td>
</tr>
<tr>
<td>5</td>
<td>19704</td>
<td>6004</td>
<td>5765</td>
<td>68527</td>
</tr>
<tr>
<td>6</td>
<td>35301</td>
<td>6753</td>
<td>15874</td>
<td>42072</td>
</tr>
<tr>
<td>7</td>
<td>34533</td>
<td>7064</td>
<td>20918</td>
<td>37485</td>
</tr>
<tr>
<td>8</td>
<td>33331</td>
<td>7243</td>
<td>26134</td>
<td>33292</td>
</tr>
<tr>
<td>9</td>
<td>30842</td>
<td>7426</td>
<td>30889</td>
<td>30843</td>
</tr>
<tr>
<td>10</td>
<td>30330</td>
<td>12536</td>
<td>30328</td>
<td>26806</td>
</tr>
<tr>
<td>11</td>
<td>28121</td>
<td>20390</td>
<td>27665</td>
<td>23824</td>
</tr>
<tr>
<td>12</td>
<td>26585</td>
<td>23608</td>
<td>26584</td>
<td>23223</td>
</tr>
<tr>
<td>13</td>
<td>25945</td>
<td>25518</td>
<td>25944</td>
<td>22593</td>
</tr>
<tr>
<td>14</td>
<td>24967</td>
<td>25011</td>
<td>25011</td>
<td>25011</td>
</tr>
</tbody>
</table>

Figure 5.4: Throughput dynamic comparison in the 'greedy' test case

the real measurements taken by YCSB. However, the absolute estimated value of throughput is lower than the real YCSB results by 19.32% on average. We do not consider this to be a significant disadvantage within our further data distribution evaluation, which will be oriented on attempting to maximize the throughput with a scenario resembling the "greedy" case discussed in this section.

In the 'greedy' case the average value difference between estimation and YCSB measurements is 4.54%, with estimated values on average 7.82% lower between steps 6 and 12. However, the 'greedy' case estimation across 11 runs has a lower standard deviation of 787.12 in comparison to YCSB 1086.39 between steps 6 and 12. After step ten, when all agents have most of the hot fragments, the YCSB does not register stable increase
in throughput while estimation indicates the room for improvement. When comparing the read distribution between YCSB and an estimation, it’s clear that our estimation mechanism shows lower degree of reads distribution balance. This is the result of the simplistic read estimation algorithm not taking into account the temporal aspect of the incoming reads. However, the stable upwards direction of the trend is holding and the end values for both the real-world YCSB and the estimation do not have a critical difference.

The most prominent advantage of the estimation over the real YCSB benchmark is the amount of time needed to get the results. We have executed benchmark runs side by side with the estimation 22 times. The average time needed from the initialization of benchmark in MDDE stage to receiving the results is 19.8 seconds for YCSB based benchmark and only 0.0034 seconds needed for estimation. This is majorly due to the fact that the estimation does not require to perform any real data shuffling or requests. Instead, a series of cheap calculations are done in the MDDE registry and returned to the stage. Additionally, the estimation is not affected by the outside factors such as shared resources or CPU contention.

Finally, the results of this evaluation section clearly demonstrate the soundness of the general MDDE architecture in the setting without the non-deterministic factor of an MDRL agent acting potentially irrationally.

5.3 Environment with cooperative scenario evaluation

We conduct four sets of evaluation experiments. The first is based on RLLib’s implementation of MADDPG. The second is based on the modified version of RLLib’s MADDPG, including the action mask mechanism eliminating invalid actions, as described in Subsection 4.2.1. The third is an experimental run where non-learning agents take random actions according to the action mask. The final set of experiments is based on MAAC.

In our experiments, we rely on the throughput estimation within our setup to mitigate the challenges and time costs of using a real YCSB benchmark. When the environment is initialized, YCSB workload is executed to generate the data and baseline read counts with the throughput. All consecutive throughput measurements are performed using our estimation. We present the base configuration of MADDPG and MAAC agents in Table 5.5. Neural network contains two fully connected hidden layers where the number of neurons depends on the number of fragments. We base agents’ hyperparameter values on the agents behavior observed in Section 5.1. Estimation configuration is identical to the one used in Subsection 5.2.2. Number of neurons in each layer depends on the size of the action and observation spaces. Specifically number_of_neurons =\| (size_of_observation_space+size_of_the_action_space)\*0.7 \|.

In the performance evaluation of the selected off-the-shelf algorithms in the data distribution scenario designed in this work, we report the performance in terms of the percentage throughput difference denoted as Thr. $\Delta \%$. This metric signifies how drastically throughput value changed as the consequence of the actions, taken by the agents, when compared with the baseline throughput at the first step. If the change is negative, it’s denoted
as a negative percentage. We select this specific metric because the exact reward magnitude in our scenario depends on real throughput measured by YCSB at the start of the experiment. This value depends on the amount of resources (CPU) available on the server when the experiment starts, which can vary, which results in drastically different baselines. However, for the purpose of our experimental setup, we’re interested in the direction and quality of distribution, not raw reward or throughput values. The initial state of the distribution is similar to the initial state of the "greedy" edge case discussed in the previous section, and depicted in Figure 5.3a.

The total length of our experiment run is 1000 episodes per 1001 step, which means 1001000 steps in total and the learning starts after episode 25. We believe it is a sufficient amount of steps to discover any positive emerging behavior of the agents.

### 5.3.1 MADDPG

For MADDPG we execute eight experimental configurations. Deviations of these configurations from the base configuration, described in Table 5.5, are as follows:

0. No differences from the base configuration.

1. The do-nothing action is disabled.

2. Storage cost is 0 (allow hoarding of data fragments).
   Benchmark frequency is set to 1, estimating measurements at every step.

3. The do-nothing action is disabled.
   Storage importance is 0.
   Benchmark frequency is set to 1.

4. Storage cost is 0.

5. The do-nothing action is disabled.
   Storage importance is 0.

---

**Table 5.5: Algorithms base configuration**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Episodes</td>
<td>1000</td>
<td>-</td>
</tr>
<tr>
<td>Episode length</td>
<td>1001</td>
<td>-</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.001</td>
<td>-</td>
</tr>
<tr>
<td>( \tau )</td>
<td>0.01</td>
<td>-</td>
</tr>
<tr>
<td>Learning starts</td>
<td>25025</td>
<td>Policies are learned after the specified step.</td>
</tr>
<tr>
<td>( \gamma )</td>
<td>0.95</td>
<td>-</td>
</tr>
<tr>
<td>Storage importance</td>
<td>0.5</td>
<td>( \varrho ) in Equation 3.9.</td>
</tr>
<tr>
<td>Do-nothing worth</td>
<td>1.0</td>
<td>( V ) in Equation 3.8.</td>
</tr>
<tr>
<td>Fragments</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>YCSB Clients</td>
<td>50</td>
<td>-</td>
</tr>
<tr>
<td>YCSB Workload</td>
<td>C.4</td>
<td>Listing C.4: Latest &quot;large&quot; YCSB workload.</td>
</tr>
<tr>
<td>Benchmark frequency</td>
<td>25</td>
<td>Frequency of throughput measurement, in between reward function is Equation 3.8.</td>
</tr>
<tr>
<td>Multi-step</td>
<td>1</td>
<td>DNN weights updated at every step.</td>
</tr>
</tbody>
</table>
6. The do-nothing action is disabled.
   Storage importance is 0.
   Benchmark frequency is set to 1.
   Replay buffer length is increased to $1 \times 10^7$.

7. The do-nothing action is disabled.
   Benchmark frequency is set to 1.
   Number of fragments increased to 80.

**Without action-mask**

We have observed the inability of MADDPG to solve the environment successfully in our experimental setup. Regardless of the hyperparameters and environment configuration, the agents were unable to discern the correct actions at any state. In the initial exploration stage, the agent performs actions at random while observing the reward feedback and learning the policies. However, at any given state, for any agent, most of the actions are not correct, such as an attempt to remove a fragment that is not in the agent’s possession or attempt to get a copy of a fragment from an agent that does not have it. In our scenario, such actions result in negative reward $-1$. This negative reward should discourage the agent from taking such actions, but the agent that sees no correct actions at all or very rarely is not able to learn a reliable policy.

MADDPG agents behavior that follows after the initial exploration phase, in the case described above, is non-desirable. The agents, in such situation, keep repeating the same action that returns negative reward or just learn to always do nothing that returns a reward with value $V \geq 0$. The outcome of such behavior is the lack of any significant data allocation optimization. Summary of the action correctness is presented in Table 5.6. It’s easy to see that the absolute majority of actions taken by the agents were incorrect, in some cases agent would learn to do nothing, if configuration allowed it. Throughout training episodes, the most throughput increase is displayed in Table 5.10, worst Table 5.11, and average Table 5.9. The average throughput increase is low. Therefore the episodes displaying the best throughput increase can be attributed to agents taking actions leading to high throughput increase simply by chance and not a stable learning trend.

In all test cases we observed, the exhibited behavior is similar. After a number of episodes, the agents fall into repeating the same action or repeating the do-nothing action in the configurations where it’s allowed. Description of the agents’ behaviour per configuration is available in Appendix Subsection D.1.
### Table 5.6: MADDPG: Action correctness

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incorrect</td>
<td>Do-nothing</td>
<td>Incorrect</td>
<td>Do-nothing</td>
</tr>
<tr>
<td>0</td>
<td>88.316%</td>
<td>4.366%</td>
<td>14.705%</td>
<td>82.712%</td>
</tr>
<tr>
<td>1</td>
<td>97.355%</td>
<td>-</td>
<td>97.512%</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>93.775%</td>
<td>3.676%</td>
<td>96.335%</td>
<td>0.999%</td>
</tr>
<tr>
<td>3</td>
<td>97.462%</td>
<td>-</td>
<td>97.400%</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>2.217%</td>
<td>97.195%</td>
<td>17.275%</td>
<td>81.466%</td>
</tr>
<tr>
<td>5</td>
<td>97.517%</td>
<td>-</td>
<td>97.475%</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>97.442%</td>
<td>-</td>
<td>97.438%</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>97.431%</td>
<td>-</td>
<td>97.417%</td>
<td>-</td>
</tr>
</tbody>
</table>

### Table 5.7: MADDPG with action mask: Action correctness

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incorrect</td>
<td>Do-nothing</td>
<td>Incorrect</td>
<td>Do-nothing</td>
</tr>
<tr>
<td>0</td>
<td>1.485%</td>
<td>1.856%</td>
<td>1.883%</td>
<td>4.588%</td>
</tr>
<tr>
<td>1</td>
<td>1.714%</td>
<td>-</td>
<td>2.098%</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>0.000%</td>
<td>6.823%</td>
<td>0.031%</td>
<td>4.594%</td>
</tr>
<tr>
<td>3</td>
<td>0.000%</td>
<td>-</td>
<td>0.033%</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>1.483%</td>
<td>2.080%</td>
<td>1.939%</td>
<td>2.613%</td>
</tr>
<tr>
<td>5</td>
<td>1.722%</td>
<td>-</td>
<td>1.998%</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>0.000%</td>
<td>-</td>
<td>0.033%</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0.000%</td>
<td>-</td>
<td>0.002%</td>
<td>-</td>
</tr>
</tbody>
</table>

### Table 5.8: Agent taken correct actions at random: Action correctness

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incorrect</td>
<td>Do-nothing</td>
<td>Incorrect</td>
<td>Do-nothing</td>
</tr>
<tr>
<td></td>
<td>0.000%</td>
<td>3.930%</td>
<td>0.018%</td>
<td>3.927%</td>
</tr>
<tr>
<td></td>
<td>0.040%</td>
<td>3.974%</td>
<td>0.045%</td>
<td>3.823%</td>
</tr>
</tbody>
</table>
With action-mask

Applying action-mask, which assists the agent with eliminating incorrect actions from consideration, results in the desired behavior of MADDPG. As we summarize in Table 5.7, the number of incorrect steps taken by the agents is reduced significantly and no agent is falling into do-nothing loop as it could happen without an action mask.

The presence of the small percentage of incorrect actions, even with the action mask, we attribute to the fact that the agents take a collective step. Within our architecture, at every step, the MDRL framework sends a collection containing an action for every agent that needs to be taken within this step. However, the major difference of our environment from the reference multi-particle environment is that agents in our environment are not independent of each other. In our environment, each agent is a data node manipulating data. Therefore, to preserve data integrity, data manipulation actions must be synchronized. We achieve this by simply executing one action at a time, sequentially. This sometimes leads to a situation when two or more of the MDRL agents select actions within the same step that are conflicting with each other. In such a case, action for the first agent in the processing queue will be a success, while other agents’ conflicting actions will fail. A simple example of such a situation is when two agents possess the only two copies of the data fragment, and both decide to remove their exemplar at the same time.

Additionally, a small percentage of incorrect actions can also be attributed to the imperfection of the incorrect action elimination mechanism we have added to the Ray RLlib’s implementation of MADDPG. We observe rare instances when a MADDPG agent can still select an incorrect action despite the mask. This occurrence is evident in our experimental results for configurations 0,1,4,5 (Table 5.7). At the same time experimental runs for configurations 2,3,6,7 did not exhibit a behavior where agents would suggest to execute masked actions, only selections of the conflicting actions was observed. Difference in our configurations can not be the source of agents selecting or not selecting incorrect actions despite the mask. We performed a series of shorter experimental runs with our configurations of MADDPG with action mask but were unable to establish a stable pattern of conditions that lead MADDPG agents to suggest incorrect actions despite the action mask.

With the application of the action mask to MADDPG, we observe that the reads are getting distributed across the nodes from the very first episode. The results are similar to the 'greedy' scenario described in Section 5.2.

Our hypothesis behind the application of the action-mask to MADDPG is to provide assistance to the algorithms in discerning the correct actions. This assistance should have allowed for the quick construction of the policies that would be able to identify correct and incorrect actions on their own. While the failure rate and the behavior of the agents appear to achieve this goal, these results are misleading. Through training episodes, the most throughput increase is displayed in Table 5.16, worst Table 5.17, and average Table 5.15. The average throughput increase is almost identical to the best. This hints to the conclusion that eliminating incorrect actions within our scenario design leads to the behavior where the learned policies of the agents become irrelevant. In other words, it doesn’t matter which action agent takes as long as the action is not violating basic action space constraints discussed in Subsection 3.2.4 and not simply a 'skip’ action executed
repeatedly, the resulting distribution of fragments will always lead to the increase of throughput within our evaluation setup.

Furthermore, MADDPG relies on the centralized critic that observes all of the agents. We hypothesized that such a feature would allow the cooperating agents to synchronize their actions. Such that agents would not take conflicting actions within the same step. However, this is not happening within our experimental setup. The number of failed actions is not diminishing with the progression of agents’ learning. As we demonstrate on plots in Appendix D.3, failed actions are evenly distributed throughout the episodes. This indicates that agents have not been able to learn to synchronize their actions.

To empirically prove that action mask application to MADDPG does not actually result in agent forming a meaningful policy within our range of experiments, we compare it with agents picking correct actions at random. At every step, these agents take the list of actions that are correct for this step, pick one at random, and execute it. These agents do not synchronize with each other in any way when choosing an action. The correctness of picked actions is comparable to MADDPG with an action mask applied to the neural network, as evident from Table 5.8. At the same time, the quality of distribution is comparable to that of MADDPG-based agents with an action mask, as can be seen from Table 5.12, Table 5.13, and Table 5.14 when compared to the corresponding tables of MADDPG with action mask (Table 5.15, Table 5.16, Table 5.17). This indicates that the quality of distribution observed in MADDPG with action mask stems from the simple fact that if the agents take only the correct actions at every step, agents achieve the optimal distribution within our evaluation setup by sheer chance. A correct action for agent in regards to any given fragment is to either acquire it, if it is not in the agent’s possession already, or remove if it is. Therefore even by taking actions at random agents quickly acquire all of the fragments by chance, which, in our scenario design results in higher throughput.

We argue that action mask, in the case of MADDPG applied to our scenario, mostly eliminates incorrect actions by adding a bias towards the correct actions but not really contributing to the learning process. It is likely is due to the fact that we apply an action mask only to the actor but not to the centralized critic in MADDPG. Therefore, the critic does not have the mechanism of incorrect actions eliminations and can still consider incorrect actions a better choice for the actor. Application of an action mask to a critic in MADDPG is problematic because it takes observations and policies of all actors present and action mask can not be simply injected into the critic’s DNN without a significant rework of the algorithm implementation, which is beyond the scope of this work.

We have performed two additional, shorter evaluations for the configurations 0 and 1 for MADDPG without an action mask. First, we have increased learning rate to 0.1. Second we have decreased the length of each episode to 101 steps, while increasing the number of episodes to 10000. In both cases we have not observed results significantly different from the ones presented in Table 5.6.
Table 5.9: MADDPG performance: average

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Thr. Δ %</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.613%</td>
<td>3.206%</td>
<td>4.300%</td>
<td>7.113%</td>
<td>85.381%</td>
</tr>
<tr>
<td>1</td>
<td>0.000%</td>
<td>3.005%</td>
<td>4.119%</td>
<td>7.235%</td>
<td>85.641%</td>
</tr>
<tr>
<td>2</td>
<td>4.325%</td>
<td>3.781%</td>
<td>4.831%</td>
<td>7.641%</td>
<td>83.747%</td>
</tr>
<tr>
<td>3</td>
<td>14.860%</td>
<td>6.372%</td>
<td>4.687%</td>
<td>9.962%</td>
<td>78.980%</td>
</tr>
<tr>
<td>4</td>
<td>22.908%</td>
<td>2.897%</td>
<td>4.290%</td>
<td>20.803%</td>
<td>72.010%</td>
</tr>
<tr>
<td>5</td>
<td>0.000%</td>
<td>3.120%</td>
<td>4.277%</td>
<td>7.005%</td>
<td>85.598%</td>
</tr>
<tr>
<td>6</td>
<td>3.654%</td>
<td>3.541%</td>
<td>4.778%</td>
<td>7.633%</td>
<td>84.048%</td>
</tr>
<tr>
<td>7</td>
<td>3.513%</td>
<td>3.471%</td>
<td>4.698%</td>
<td>7.535%</td>
<td>84.297%</td>
</tr>
</tbody>
</table>

Table 5.10: MADDPG performance: best

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Thr. Δ %</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>87.484%</td>
<td>29.803%</td>
<td>3.353%</td>
<td>32.121%</td>
<td>34.723%</td>
</tr>
<tr>
<td>1</td>
<td>0.000%</td>
<td>3.005%</td>
<td>4.119%</td>
<td>7.235%</td>
<td>85.641%</td>
</tr>
<tr>
<td>2</td>
<td>141.676%</td>
<td>25.077%</td>
<td>24.905%</td>
<td>24.941%</td>
<td>25.077%</td>
</tr>
<tr>
<td>3</td>
<td>141.836%</td>
<td>25.086%</td>
<td>24.772%</td>
<td>25.106%</td>
<td>25.036%</td>
</tr>
<tr>
<td>4</td>
<td>50.470%</td>
<td>43.012%</td>
<td>4.085%</td>
<td>6.629%</td>
<td>46.274%</td>
</tr>
<tr>
<td>5</td>
<td>0.000%</td>
<td>3.120%</td>
<td>4.277%</td>
<td>7.005%</td>
<td>85.598%</td>
</tr>
<tr>
<td>6</td>
<td>141.506%</td>
<td>25.097%</td>
<td>25.084%</td>
<td>25.050%</td>
<td>24.769%</td>
</tr>
<tr>
<td>7</td>
<td>141.569%</td>
<td>25.023%</td>
<td>24.850%</td>
<td>24.928%</td>
<td>25.199%</td>
</tr>
</tbody>
</table>

Table 5.11: MADDPG performance: worst

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Thr. Δ %</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000%</td>
<td>2.973%</td>
<td>4.303%</td>
<td>7.069%</td>
<td>85.655%</td>
</tr>
<tr>
<td>1</td>
<td>0.000%</td>
<td>3.005%</td>
<td>4.119%</td>
<td>7.235%</td>
<td>85.641%</td>
</tr>
<tr>
<td>2</td>
<td>0.000%</td>
<td>3.013%</td>
<td>4.240%</td>
<td>7.147%</td>
<td>85.600%</td>
</tr>
<tr>
<td>3</td>
<td>0.000%</td>
<td>2.978%</td>
<td>4.185%</td>
<td>7.021%</td>
<td>85.816%</td>
</tr>
<tr>
<td>4</td>
<td>0.000%</td>
<td>2.933%</td>
<td>4.144%</td>
<td>6.906%</td>
<td>86.017%</td>
</tr>
<tr>
<td>5</td>
<td>0.000%</td>
<td>3.120%</td>
<td>4.277%</td>
<td>7.005%</td>
<td>85.598%</td>
</tr>
<tr>
<td>6</td>
<td>0.000%</td>
<td>2.977%</td>
<td>4.251%</td>
<td>7.200%</td>
<td>85.572%</td>
</tr>
<tr>
<td>7</td>
<td>0.000%</td>
<td>2.969%</td>
<td>4.236%</td>
<td>7.136%</td>
<td>85.659%</td>
</tr>
</tbody>
</table>
### Table 5.12: Random agents performance: average

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Thr. Δ %</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>141.750%</td>
<td>24.940%</td>
<td>24.846%</td>
<td>24.931%</td>
<td>25.283%</td>
</tr>
</tbody>
</table>

### Table 5.13: Random agents performance: best

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Thr. Δ %</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>142.323%</td>
<td>25.015%</td>
<td>24.981%</td>
<td>25.000%</td>
<td>25.004%</td>
</tr>
</tbody>
</table>

### Table 5.14: Random agents performance: worst

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Thr. Δ %</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>140.632%</td>
<td>24.596%</td>
<td>24.804%</td>
<td>25.059%</td>
<td>25.541%</td>
</tr>
</tbody>
</table>
Table 5.15: MADDPG with action mask performance: average

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Thr. Δ %</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>138.522%</td>
<td>23.094%</td>
<td>25.092%</td>
<td>24.792%</td>
<td>27.023%</td>
</tr>
<tr>
<td>1</td>
<td>139.562%</td>
<td>23.776%</td>
<td>24.421%</td>
<td>25.063%</td>
<td>26.739%</td>
</tr>
<tr>
<td>2</td>
<td>141.194%</td>
<td>24.444%</td>
<td>25.063%</td>
<td>24.857%</td>
<td>25.635%</td>
</tr>
<tr>
<td>3</td>
<td>140.929%</td>
<td>24.905%</td>
<td>24.560%</td>
<td>24.575%</td>
<td>25.960%</td>
</tr>
<tr>
<td>4</td>
<td>138.825%</td>
<td>24.431%</td>
<td>23.984%</td>
<td>24.843%</td>
<td>26.742%</td>
</tr>
<tr>
<td>5</td>
<td>138.861%</td>
<td>24.722%</td>
<td>23.742%</td>
<td>25.008%</td>
<td>26.528%</td>
</tr>
<tr>
<td>6</td>
<td>141.081%</td>
<td>24.983%</td>
<td>25.051%</td>
<td>24.104%</td>
<td>25.862%</td>
</tr>
<tr>
<td>7</td>
<td>141.487%</td>
<td>24.443%</td>
<td>24.069%</td>
<td>24.090%</td>
<td>27.398%</td>
</tr>
</tbody>
</table>

Table 5.16: MADDPG with action mask performance: best

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Thr. Δ %</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>141.919%</td>
<td>24.978%</td>
<td>25.007%</td>
<td>24.997%</td>
<td>25.018%</td>
</tr>
<tr>
<td>1</td>
<td>142.139%</td>
<td>25.009%</td>
<td>25.011%</td>
<td>24.984%</td>
<td>24.996%</td>
</tr>
<tr>
<td>2</td>
<td>141.647%</td>
<td>25.002%</td>
<td>25.001%</td>
<td>24.990%</td>
<td>25.007%</td>
</tr>
<tr>
<td>3</td>
<td>141.647%</td>
<td>25.002%</td>
<td>25.001%</td>
<td>24.990%</td>
<td>25.007%</td>
</tr>
<tr>
<td>4</td>
<td>141.846%</td>
<td>25.010%</td>
<td>25.010%</td>
<td>24.982%</td>
<td>24.998%</td>
</tr>
<tr>
<td>5</td>
<td>141.812%</td>
<td>25.001%</td>
<td>25.003%</td>
<td>25.022%</td>
<td>24.974%</td>
</tr>
<tr>
<td>6</td>
<td>141.947%</td>
<td>25.002%</td>
<td>24.976%</td>
<td>25.014%</td>
<td>25.008%</td>
</tr>
<tr>
<td>7</td>
<td>142.160%</td>
<td>24.948%</td>
<td>25.006%</td>
<td>25.056%</td>
<td>24.990%</td>
</tr>
</tbody>
</table>

Table 5.17: MADDPG with action mask performance: worst

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Thr. Δ %</th>
<th>Agent 0</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>131.407%</td>
<td>21.937%</td>
<td>26.958%</td>
<td>23.945%</td>
<td>27.160%</td>
</tr>
<tr>
<td>1</td>
<td>132.279%</td>
<td>28.945%</td>
<td>23.977%</td>
<td>23.958%</td>
<td>23.120%</td>
</tr>
<tr>
<td>2</td>
<td>137.720%</td>
<td>24.038%</td>
<td>24.351%</td>
<td>25.918%</td>
<td>25.693%</td>
</tr>
<tr>
<td>3</td>
<td>137.720%</td>
<td>24.038%</td>
<td>24.351%</td>
<td>25.918%</td>
<td>25.693%</td>
</tr>
<tr>
<td>4</td>
<td>131.426%</td>
<td>27.722%</td>
<td>25.048%</td>
<td>25.514%</td>
<td>21.716%</td>
</tr>
<tr>
<td>5</td>
<td>133.493%</td>
<td>25.195%</td>
<td>21.989%</td>
<td>26.230%</td>
<td>26.586%</td>
</tr>
<tr>
<td>6</td>
<td>138.609%</td>
<td>24.391%</td>
<td>24.896%</td>
<td>24.463%</td>
<td>26.250%</td>
</tr>
<tr>
<td>7</td>
<td>140.087%</td>
<td>25.421%</td>
<td>24.896%</td>
<td>24.263%</td>
<td>25.420%</td>
</tr>
</tbody>
</table>
5.3.2 MAAC

For MAAC we execute six experimental configurations. Deviations of these configurations from the base configuration, described in Table 5.5, are as follows:

0. No differences from the base configuration.
1. The do-nothing action is disabled.
2. Storage cost is 0 (allow hoarding of data fragments).
   - Benchmark frequency is set to 1, estimating measurements at every step.
3. The do-nothing action is disabled.
   - Storage cost is 0.
   - Benchmark frequency is set to 1.
4. The do-nothing action is disabled.
   - Storage cost is 0.
   - Benchmark frequency is set to 1.
   - Replay buffer length is increased to $1 \times 10^7$.
5. The do-nothing action is disabled.
   - Benchmark frequency is set to 1.
   - Number of fragments increased to 80.

We observe the MAAC agent’s behavior, which is similar to the one of MADDPG without an action mask. We summarize the percentage of failed actions executed by MAAC in Table 5.18. We clearly observe the pattern of agents’ behavior similar to MADDPG, where agents either perform a high number of incorrect actions or opt-out to skipping actions by executing the do-nothing action in the configurations where it’s available.

The experimental set for MAAC is more narrow and limited in scope in comparison to MADDPG. Also, we limit the number of episodes for MAAC configurations. When all of the agents either select do-nothing action or any other action repeatedly for ten consecutive episodes, we interrupt the experiment. We believe it is acceptable, because our observations in MADDPG (without action mask) indicate that as soon as agents fall into either skipping actions consistently or repeating the same action, they never recover. The exact number of episodes per configuration and the reason for interruption is described in Appendix Subsection D.2.

The reference MAAC implementation is computationally expensive. Cutting experimental runs short, allows us to sample multiple different hyperparameter configurations, and observe the results, within a limited time frame. However, we recognize that this is a potential threat to the validity of the results.
Table 5.18: MAAC: Action correctness

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Agent 0 Incorrect</th>
<th>Agent 0 Do-nothing</th>
<th>Agent 1 Incorrect</th>
<th>Agent 1 Do-nothing</th>
<th>Agent 2 Incorrect</th>
<th>Agent 2 Do-nothing</th>
<th>Agent 3 Incorrect</th>
<th>Agent 3 Do-nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>41.396%</td>
<td>57.045%</td>
<td>30.549%</td>
<td>67.612%</td>
<td>96.712%</td>
<td>0.410%</td>
<td>32.134%</td>
<td>46.413%</td>
</tr>
<tr>
<td>1</td>
<td>94.765%</td>
<td>-</td>
<td>69.594%</td>
<td>-</td>
<td>93.977%</td>
<td>-</td>
<td>82.587%</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>15.984%</td>
<td>82.581%</td>
<td>16.154%</td>
<td>81.679%</td>
<td>15.510%</td>
<td>81.760%</td>
<td>14.108%</td>
<td>4.983%</td>
</tr>
<tr>
<td>3</td>
<td>94.124%</td>
<td>-</td>
<td>96.152%</td>
<td>-</td>
<td>94.671%</td>
<td>-</td>
<td>87.792%</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>96.378%</td>
<td>-</td>
<td>96.675%</td>
<td>-</td>
<td>95.711%</td>
<td>-</td>
<td>93.094%</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>97.906%</td>
<td>-</td>
<td>97.640%</td>
<td>-</td>
<td>97.436%</td>
<td>-</td>
<td>89.692%</td>
<td>-</td>
</tr>
</tbody>
</table>
5.3.3 Summary

Our observations clearly indicate that the selected off-the-shelf MDRL algorithms were unable to solve the environment within the scope of our experimental setup. Both MADDPG and MAAC exhibit a similar behavior where the agents are unable to discern the correct conditions when any of the actions should be invoked.

We have observed the agents’ behavior, when consistently unable to construct a meaningful policy for the correct actions. In such cases, agents either fall back to do-nothing, which is technically a correct action in our scenario, or simply start taking the same action in repeated succession till the end of the episode or during the span of multiple episodes. In the latter case, agents cease any exploration and disregard any attempts to discover higher, non-negative, rewards. Both of the cases clearly indicate the inability of the agents to construct meaningful connections between the state of the environment and the actions, withing our experimental setup.

Usage of action mask with MADDPG resulted in the desired result in terms of fragment distribution and throughput increase. However, the agents were unable to predict each others’ actions, which consistently led to steps where agents would take conflicting actions. Additionally, the fact that the agents that were taking only correct actions at random were able to achieve the comparable performance, clearly suggests that using an action mask simply reduced a DRL-based agent to basic heuristics within our scenario. Therefore, the use of an action mask does not solve the core problem of the agents’ inability to discern conditions for the valid execution of the actions within our experimental setup.

Additionally, the demonstrated efficiency of agents that take only correct actions but at random, clearly hints on the fact that the nuanced and restrictive nature of the scenario that we use for evaluation of our designed environment is capable of being reduced to basic heuristics when all of the data nodes are equal. We believe that the introduction of differences between data nodes, such that some are faster than others, would potentially change this outcome because simply putting all of the fragments on every node would no longer be optimal. However, testing of this hypothesis is outside the scope of this evaluation, which is focused on MDRL specifically.
6 Conclusion and Future Work

In this chapter, we summarize the main contributions of this research, discuss some possible improvements to the presented evaluation setup and outline some promising directions for future research.

6.1 Conclusion

To the best of our knowledge, this work is the first attempt to create a multi-agent data distribution optimization environment for use with multi-agent deep reinforcement learning algorithms.

- Within this work, we have designed a multi-agent reinforcement learning oriented environment explicitly designed for simulation of various data allocation scenarios within a distributed infrastructure. The design of our environment is inspired by the features observed in the review of existing reference multi-agent environments. We provide a detailed architecture design as well as the reasoning behind the specifics of implementation. The created environment is designed to be versatile and easy to use by the researchers in the field. The modular architecture and clear separation of concerns in the codebase of our environment are specifically meant for ease of extension and modification for further research.

- The proposed multi-agent data distribution environment is designed to be deployed in a real-world infrastructure. This way, it is capable of capturing the effects of the infrastructure on the performance of data retrieval.

- We provide an estimation mechanism that allows us to avoid the need for a complex infrastructure to use our environment. Instead it is possible to use our environment in a system with limited resources. We empirically demonstrate that our estimation mechanism, is considerably faster than real-world benchmarking. The proposed estimation mechanism reflects the trend of changes in throughput directly corresponding to changes in allocation.

A degree of difference between the real-world and estimated metrics is observed. The estimation mechanism is not taking into account the temporal aspect of the incoming read queries. These differences can negatively affect the learning of the agents if real-world and the estimated benchmarks are used interchangeably. However considering the comparable direction of change in throughput and end values, we believe the estimation can be successfully used for speeding up model training and hypothesis evaluation. If the hypothesis holds with the estimation, then it can reasonably be tested with real-world metrics in a longer experiment run.
• We have designed a cooperative scenario for our data distribution environment that allows MDRL agents to control the placement of each fragment directly. Agents have the capability to select a specific fragment and acquire it from a specific source. Agents also are capable of removing a copy of a specific fragment which they own, if there’s at least one more copy present. Evaluation under different configurations of our environment did not yield results that suggest a formation of reasonable policies by the agents. The default implementations of the selected off-the-shelf MDRL algorithms, within our experimental configuration, were unable to discover a stable policy consistently resulting in agent exhibiting learned behavior oriented towards throughput maximization.

The evaluation of two actor-critic based off-the-shelf MDRL algorithms, MADDPG and MAAC, with our multi-agent data distribution environment and direct fragment allocation manipulation scenario revealed several challenges and shortcomings of the designed scenario. It is hard to evaluate the exact reasons behind the developed positive or negative policies in reinforcement learning based on deep neural networks. However, we believe it is appropriate to observe the behavior of the agents and attempt to derive logical conclusions.

Our scenario has an extremely nuanced action space. This means that most of the actions are valid only when the state fulfills a very specific condition. Specifically, the copy action is legal only when two specific bits have very specific values, that is 0 in the specific fragment slot in the destination agent and 1 in the slot for the same fragment in the source. These would signify that the destination does not already have the fragment, and the source has a copy. It’s a very simple concept for a human, but our evaluation shows that it is challenging for the MDRL agents.

In reinforcement learning, agents do not have any knowledge about the purpose of the actions and what any bit in the observation space represents. Agents can only learn policies based on the reward feedback, after executing an action. An agent takes action and if it returned high reward, the agent would attempt to take this action again in the similar state. If the action returns a high reward again, such behavior is reinforced and more likely to occur again. Such logic works well in synthetic environments such as multi-agent particle environment, where there are no illegal actions.

However, in our data allocation optimization scenario, an agent can not repeat the same action. For example, an agent copied a fragment to itself, and throughput has increased, it is viewed as positive. The issue appears from the fact that the agent is unable to repeat the action until it itself deletes this fragment from its own storage, and the source node again gets this fragment. It is however, important to understand that the action was valid because two bits in the observation space held specific values. The agents have no explicit knowledge which bits influence the action validity, and the entirety of the observation space does not stay static either. It makes it difficult for the agents to establish the correlations between the validity of the specific action and observations.

We have also observed the behavior in both MADDPG and MAAC where agents, unable to discover the correct actions consistently, apparently consider the negative reward for incorrect actions the highest. Eventually halting the exploration. If all agents consistently execute incorrect actions, nothing changes in the state because all of the actions
are rejected, and agents are unable to form policies. This essentially halts any possible progress agents might have.

We hypothesize that the problem above can potentially be solved with a longer initial randomized action execution, such that way, agents could generate more experience prior to acting according to their policies, which means increasing the number of steps before learning starts and the overall number of the episodes. However, it’s problematic due to the computational expensiveness of the environment. Because the environment has to operate with real data and a real distributed architecture, there is a considerable overhead associated with the communication costs between the components of the environment and data records manipulation in the data nodes. We were able to considerably speed up the learning process by employing estimation instead of the real benchmark but not enough to make the environment fast enough to stage longer experiments within a reasonable amount of time within our experimental setup.

However, due to the nature of the scenario, where agents manipulate data fragments directly, the complexity and consequently, training time will increase with the number of fragments and nodes as the action space grows. Our evaluation of the incorrect actions elimination with an action mask, demonstrates the lack of learned coordination between agents. We also show that usage of an action mask has an effect comparable with simply executing only correct actions randomly. This demonstrates the general flaw of the scenario design. The action space should be re-evaluated in future work, actions should be more generalized.

6.2 Threats to validity

In this section, we outline the noteworthy problems we have encountered during the evaluation and how they influence the conclusions of this work.

- **Extreme hardware setup dependence.**
  Results of evaluation based on the benchmark runs performed against the real data nodes are heavily dependant on multiple conditions outside of the environment control. Network performance, parallel CPU utilization, RAM availability, etc. All of these conditions affect throughput measurements and depend heavily on the setup where the environment is run. We have observed a high degree of variability when retrieving throughput measurements based on the real data nodes. This degree of instability has the potential of affecting the learning performance of the agents and the establishment of the baseline values for the benchmark estimation.

- **Lack of MAAC hyperparameter influence evaluation.**
  The reference implementation of MAAC [IS18], which is one of two reference state-of-the-art MDRL algorithms we have used for evaluation of our environment, is computationally expensive to execute. Due to this fact, we were unable to execute the full set of hyperparameter tuning evaluations for this algorithm. MAAC is based on MADDPG [LWT+17], for which we provide a comprehensive hyperparameter influence evaluation against a reference environment. However, we recognize that the two algorithms are not identical, and there can be differences in the effects of the same hyperparameters values between these algorithms.
• **Limited scope of MDDE environment evaluation.**
  Repeated experiments are computationally expensive and take a considerable amount of time in our evaluation setup. Therefore, we execute only a limited number of possible configurations for the evaluation of the environment and the performance of the selected MDRL algorithms. More experiments are required to make a complete and conclusive evaluation of feasibility for the direct fragment allocation manipulation with MDRL algorithms. Specifically, attention should be paid to more complex agents’ deep neural network configurations.

• **YCSB is a synthetic workload generator.**
  We rely on YCSB for measuring the overall quality of the data distribution, and to produce the baseline metrics for the throughput estimator. However, a recent publication [zCDVD20] outlines the fact that YCSB is a synthetic test that might produce misleading results, while proposing an approach that’s more suitable for generating workloads that better represent real world. We suggest that future work in the domain of MDRL for data distribution would benefit from taking a closer look at workload emulation.

### 6.3 Future Work

Reinforcement learning environment like the one developed within the confines of this work can serve as a basis for further research in general-purpose MDRL and algorithms and concepts specific for data distribution optimization. In our work, we have provided limited insight into the application of MDRL in the data distribution, while laying the foundation for future research. Taking into account the results and observations of our work, we recommend a number of directions for further investigation.

**A comprehensive model of the system**

Our environment relies heavily on real-world database nodes and a distributed architecture of the components. The intention behind this design was to bring MDRL as close as possible to real world use case. However, such design has proven to be impractical and heavily dependant on the system setup, introducing unintended effects in the training process (e.g. CPU sharing affecting the throughput). Operating with real data records and measuring the quality of distribution based on throughput dramatically increases the time needed for training the agents.

We have provided a naive estimation of the benchmark to alleviate the costs of running one during the training process. However, this estimation is simplistic in nature and does not alleviate communication costs between different components of our environment, only the costs of data manipulation are eliminated.

A carefully constructed model of the system can serve as the basis for the training of MDRL agents before they are deployed to the real system. Recent research [KKHK19] in DRL shows the possibility of moving agents from a simulation to a real-world system, even with not fully matching action spaces. Such a model should not require a distributed
architecture, reducing communication costs, and no need to operate with real data, reducing the time needed for training. A model would provide a more stable, hardware-independent, training and evaluation setup. Modeling distributed infrastructure instead of actually deploying it also gives the researcher an ability to run multiple instances of the environment within the same MDRL algorithm instance. It is also possible to provide simulations of real world conditions such as communication noise, latency or faulty data nodes. These additional conditions can serve as the basis for the evaluation of agent behavior in non-ideal scenarios.

**Generic actions scenario**

Taking into consideration the observations from the evaluation Section 5.3, we believe that a simpler, more generic action space might produce better results. Specifically, introduce simpler actions for agents, instead of attempting to make them manipulate each individual fragment, while at the same time attempting to make a policy with meaningful connections between the popularity of the fragment and the action to copy or remove it. We see that that multi-agent particle environment default scenarios, described in Subsection 2.3.1, have rather small and simplistic actions and no notion of illegal actions.

We propose a simple action space consisting out of three simple actions.

1. Copy to self most popular fragment which is not yet allocated on the agent.
2. Remove lest popular fragment which is allocated on the self and has more than one other copy allocated on another agent.
3. Do-nothing (skip step).

Observation space can consist of a simple array displaying the number of reads performed from each node during the benchmark run or estimation. Reward function should take into account throughput of the system and the cost of storage. The desired behavior of agents is a learned balance between requesting more fragments and getting rid of not very popular fragments.

Such action space is very simplistic and does not make much sense from a practical standpoint as it can as easily be solved with most primitive heuristics. However, we believe its simplicity is comparable to the default MPE scenarios and should be solvable with MADDPG. Such simple data distribution scenario can be used as a basic reference evaluation scenario for algorithms, as well as a base for extension and trying out more custom actions.

**Economic scenario**

The default cooperative scenario, provided in our environment and used for evaluation, is designed for testing the basic ability of MDRL agents to operate within the chosen domain. It directly observes and manipulates individual exemplars of data fragments. A particular shortcoming of such an approach is scalability. With more data fragments, the action and observation spaces must increase proportionally, which also leads to the increased complexity of agents’ training.
However, we believe that a possible solution to this challenge might be the application of the approach for data migration and queries based on the simulated economic-like bidding system for the Mariposa DBMS\cite{SD+94}. Mariposa DBMS, is the heuristic-based database concept based on game theory to facilitate optimal data distribution. The proposed approach for data migration and queries is based on the simulated economic-like bidding system. Queries, when executed, have a specific budget that can be used to produce results. To produce the results, the query processing system, called "the broker", creates "contracts" with different storage nodes (agents) in an attempt to execute various parts of the query at a minimal cost. At the same time, storage nodes attempt to increase their "revenue" from such "contracts" by "buying" or "selling" data objects from or to other data storage nodes. However, this approach relies solely on heuristics and static rules defined and tuned in advance.

Additionally, considering the architecture of the database, specifically the broker component that "contracts" different local nodes to execute various tasks, we believe that \textit{M}$^3$RL \cite{ST18} is a suitable candidate for implementation in the Mariposa DBMS. Conceptually, \textit{M}$^3$RL introduces a manager agent corresponding to Mariposa's broker that also "contracts" other agents to perform tasks for which these agents are most suitable.

Mariposa DB is similar in concept to previously mentioned in Section 2.4 NashDB. Due to its nature, NashDB raises the question of efficient training of agents in the environment where the number of agents can vary. However, during our survey, we have identified a publication\cite{WS18} attempting to solve this particular issue by creating a communication network between agents.

**Wider range of configurations and algorithms**

Within this work, we relied on two off-the-shelf algorithm implementations, MADDPG and, based on it, MAAC. We tested both of the algorithms in the environment, we have developed within the scope of this work, with only a limited range of parameters. Expanding the experiments, especially the design of DNN used in the agents, might yield more promising results. Additionally, we believe it’s prudent to evaluate the designed direct fragment manipulation scenario, we have developed in this work, with a wider range of algorithms, specifically not based on MADDPG.

**Agent actions synchronisation**

The safety of the data records allocated within a distributed database system is a priority. At no point of time, an agent should perform an action that results in data being lost. Since the intention of an MDRL system is to make agents act independently, a mechanism that prevents the loss of data must be implemented.

Within the design of our environment, such a mechanism is provided by the registry, which in this case, serves as the central control node. The registry allows executing actions only sequentially. Therefore, there is no possibility for multiple agents to take conflicting actions, that break any constraints, simultaneously. However, this is only
feasible within a system based on agents making collective synchronous steps. In addition, the existence of such central control node is conceptually questionable from the point of view of an MDRL-based system, where each agent should have a high degree of independence.

Such a centralized design is a clear potential bottleneck and the point of failure. We believe that a more robust, decentralized mechanism that ensures the safety of the data records must be researched and implemented.

**Single-agent DRL in multi-agent setting**

We only provide an evaluation of the multi-agent data distribution scenario for direct fragment allocation manipulation for MDRL algorithms. We also observe the difficulties the evaluated algorithms have with deciphering a highly state-dependent, nuanced action space. We’ve observed instances when agents would converge on performing an incorrect action repeatedly, even when a do-nothing action is permitted. In these cases, agents consistently receiving a negative reward, even though previously they’ve observed a do-nothing reward, that returns higher reward than incorrect action. We believe this to be the result of centralized critic influence on the agent’s policies.

In light of this observation, we suggest the application of single-agent based agents in the multi-agent scenario designed within this work. Our integration mechanism with Ray RLLib permits the use of multiple single-agent algorithms in a multi-agent environment. We clearly demonstrate it with a reference multi-agent environment in Sub-section 2.5.3. However, as we clearly demonstrate in Table 2.5, such evaluation is computationally expensive and time-consuming even with a simpler reference multi-agent environment.

**Advanced fragmentation and partial record reads**

As noted in Section 3.2.1, current the fragmented supports only vertical fragmentation. We recognize that the choice of the fragmentation scheme can make a significant difference in data retrieval performance in a distributed system. We believe that our environment can benefit from the addition of support for more fragmentation schemes (e.g., vertical, hash) as well as improvement in the current horizontal fragmentation logic.

In addition, currently, the fragments of equal size are formed uniformly across the data nodes and can never be changed within the experimental run. We believe it variable size fragments can present an interesting challenge for the data distribution problem. In order to implement such fragmentation logic, additions to all MDDE components are required. The registry must include new APIs for the stage that would facilitate control upon the fragmentation scheme from the stage. Benchmark runner (YCSB) and estimation must be extended to support the desired fragmentation schemes.
Bibliography


A Appendices

A. Verification of the selected off-the-shelf algorithms implementation.

In order to ensure the viability of use for publicly available MDRL algorithms implementations, we perform a series of experiments. We run these algorithms against Multi-Agent Particle Environment to verify the general soundness of the implementation. The purpose is to verify that the implementations of the algorithms are provided fully, functional, and exhibit the expected behavior. We rely on the mean episode reward per agent as a metric. While reinforcement learning, due to its nature, is prone to a certain degree of variability in returned reward values, general learning behavior is expected to be the same. Meaning the agents learn to behave in the given environment and maximize their rewards.

In these test runs, we rely entirely on the published and openly available code. Any changes are introduced only if absolutely (e.g., a code error) necessary and if those do not affect the logic and behavior of the algorithm. Tweaking of existing or invention of new MDRL algorithms is beyond the scope of this work. Additionally, all of the hyper-parameter values presented in this section are the default values from the original corresponding publications.

There is a difference in the reward scale that is returned as the metric for MAAC and the algorithms implemented by Ray RLib (MADDPG, DQN). This is due to the differences in the internal processing of the rewards between these implementations. We do not perform any re-scaling of the output and provide the results as-is. We are not interested in the specific numbers but rather the learning trends in this verification.

We perform all of the experiments mentioned in this section using a single CPU workstation PC. We provide PC specifications in table A.1.

Implementations of the algorithms, which we mention in this section, provide the capability of performing DRL related calculations using either GPU or CPU. All of the results provided below were achieved with the calculations performed by the CPU.

A.1 MADDPG

OpenAI originally employed Multi-Agent Particle Environment alongside MADDPG, for evaluation of MADDPG [LWT+17]. However, we are not relying on the original implementation of the algorithm. Instead, we use the one provided with Ray RLib[LLM+17] and reproduce the experiments outlined in the original publication.
Table A.1: PC Specifications

<table>
<thead>
<tr>
<th>Component</th>
<th>Manufacturer</th>
<th>Model</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel</td>
<td>Core i7-3820</td>
<td>Base Frequency: 3.60 GHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cores: 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Threads: 8</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia</td>
<td>Geforce GTX 970</td>
<td>VRAM: 4 Gb</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CUDA: 5.2</td>
</tr>
<tr>
<td>RAM</td>
<td>Kingston</td>
<td>KHX1600C9D3K4/16GX</td>
<td>Volume: 16 Gb (4 Gb × 4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Standard: DDR3-1600</td>
</tr>
<tr>
<td>Disk</td>
<td>Intel</td>
<td>SSD 530 Series</td>
<td>Volume: 120 GB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: 2.5in SATA 6Gb/s</td>
</tr>
<tr>
<td>Op. system</td>
<td>Canonical</td>
<td>Ubuntu 18.04.4 LTS</td>
<td>Architecture: 64-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Shared access (Exclusive use of 100% of the system’s capacity is not guaranteed)</td>
</tr>
</tbody>
</table>

For convenience, we make public the exact configurations used to run these experiments in a publicly available repository\(^1\), code base of which is directly forked from the publicly available code\(^2\) published by the author of MADDPG implementation in Ray RLlib version 0.8.0.dev2. We do not introduce any changes in the provided experimental code, but we used custom Docker-based experimental setup that we make public in the Internet for convenience and simplification of reproduction if required. We have additionally verified each experiment once on version 0.8.4 to ensure no errors were introduced with a newer Ray RLlib version, which we used in the evaluation of our own environment. We have observed no notable differences in MADDPG performance in default MPE scenarios between versions 0.8.0.dev2 and 0.8.4 of Ray RLlib. Hyperparameters, used for Ray RLlib MADDPG experiments, are presented in Table A.2.

\(^1\)https://github.com/akharitonov/maddpg-rllib [accessed 2020 July 7]


![Figure A.1: MADDPG: MPE - Physical deception](image-url)
Table A.2: MADDPG hyperparameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>max-episode-len</td>
<td>25</td>
</tr>
<tr>
<td>num-episodes</td>
<td>60000</td>
</tr>
<tr>
<td>good-policy</td>
<td>'maddpg'</td>
</tr>
<tr>
<td>adv-policy</td>
<td>'maddpg'</td>
</tr>
<tr>
<td>lr</td>
<td>1e-2</td>
</tr>
<tr>
<td>gamma</td>
<td>0.95</td>
</tr>
<tr>
<td>sample-batch-size</td>
<td>25</td>
</tr>
<tr>
<td>train-batch-size</td>
<td>1024</td>
</tr>
<tr>
<td>n-step</td>
<td>1</td>
</tr>
<tr>
<td>num-units</td>
<td>64</td>
</tr>
<tr>
<td>replay-buffer</td>
<td>1000000</td>
</tr>
<tr>
<td>checkpoint-freq</td>
<td>7500</td>
</tr>
<tr>
<td>num-workers</td>
<td>1</td>
</tr>
<tr>
<td>num-envs-per-worker</td>
<td>4</td>
</tr>
<tr>
<td>num-gpus</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure A.2: MADDPG: MPE - Covert communication

Figure A.3: MADDPG: MPE - Keep-away
For convenience, we make public the exact configurations used to run these experiments in a publicly available repository\(^3\), code base of which is directly forked from the original code\(^4\) published by the authors of the Multi-Actor-Attention-Critic algorithm [IS18]. We do not introduce any changes in the core algorithm logic, but employ custom Docker-based

---

\(^3\)https://github.com/akharitonov/MAAC [accessed 2020 July 7]

experimental setup and MPE-based experiments, which we make public for convenience and simplification of reproduction. Hyperparameters, used for the MAAC experiments, are presented in Table A.3.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer_length</td>
<td>1000000</td>
</tr>
<tr>
<td>n_episodes</td>
<td>50000</td>
</tr>
<tr>
<td>episode_length</td>
<td>25</td>
</tr>
<tr>
<td>steps_per_update</td>
<td>100</td>
</tr>
<tr>
<td>num_updates</td>
<td>4</td>
</tr>
<tr>
<td>batch_size</td>
<td>1024</td>
</tr>
<tr>
<td>save_interval</td>
<td>1000</td>
</tr>
<tr>
<td>pol_hidden_dim</td>
<td>128</td>
</tr>
<tr>
<td>critic_hidden_dim</td>
<td>128</td>
</tr>
<tr>
<td>attend_heads</td>
<td>4</td>
</tr>
<tr>
<td>pi_lr</td>
<td>0.001</td>
</tr>
<tr>
<td>q_lr</td>
<td>0.001</td>
</tr>
<tr>
<td>tau</td>
<td>0.001</td>
</tr>
<tr>
<td>gamma</td>
<td>0.99</td>
</tr>
<tr>
<td>reward_scale</td>
<td>100.</td>
</tr>
</tbody>
</table>

Figure A.7: MAAC: MPE - Physical deception
Figure A.8: MAAC: MPE - Covert communication

Figure A.9: MAAC: MPE - Keep-away

Figure A.10: MAAC: MPE - Cooperative communication
Figure A.11: MAAC: MPE - Cooperative navigation

Figure A.12: MAAC: MPE - Predator-prey
A.3 DQN

Similarly to MADDPG, we utilize the implementation of DQN provided in Ray RLlib. While DQN is not a multi-agent algorithm, Ray RLlib provides a convenient way of running single-agent DRL algorithms in multi-agent environments. Used Ray RLlib version 0.8.0.dev2. Hyperparameters, used for Ray RLlib DQN experiments, are presented in Table A.4.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>max-episode-len</td>
<td>25</td>
</tr>
<tr>
<td>num-episodes</td>
<td>60000</td>
</tr>
<tr>
<td>num-adversaries</td>
<td>0</td>
</tr>
<tr>
<td>trainer</td>
<td>'dqn'</td>
</tr>
<tr>
<td>lr</td>
<td>1e-2</td>
</tr>
<tr>
<td>gamma</td>
<td>0.95</td>
</tr>
<tr>
<td>sample-batch-size</td>
<td>25</td>
</tr>
<tr>
<td>train-batch-size</td>
<td>1024</td>
</tr>
<tr>
<td>n-step</td>
<td>1</td>
</tr>
<tr>
<td>num-units</td>
<td>64</td>
</tr>
<tr>
<td>replay-buffer</td>
<td>1000000</td>
</tr>
</tbody>
</table>

Figure A.13: DQN: MPE - Physical deception
Figure A.14: DQN: MPE - Covert communication

Figure A.15: DQN: MPE - Keep-away

Figure A.16: DQN: MPE - Cooperative communication
Figure A.17: DQN: MPE - Cooperative navigation

Figure A.18: DQN: MPE - Predator-prey
B. Plots: MADDPG hyper-parameter tuning for MPE.

In this section, we provide plots for testing the hyperparameter influence of MADDPG [LWT+17] on its performance in solving Multi-agent particle environment. In this evaluation we rely on MADDPG implementation provided in Ray RLlib version 0.8.4. Specifications of the hardware platform where test were performed are outlined in Table B.5.

<table>
<thead>
<tr>
<th>Component</th>
<th>Manufacturer</th>
<th>Model</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor (1)</td>
<td>Intel</td>
<td>Xeon(R) Gold 6150</td>
<td>Base Frequency: 2.70 GHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cores: 18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Threads: 36</td>
</tr>
<tr>
<td>Processor (2)</td>
<td>Intel</td>
<td>Xeon(R) Gold 6150</td>
<td>Base Frequency: 2.70 GHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cores: 18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Threads: 36</td>
</tr>
<tr>
<td>RAM</td>
<td>Samsung</td>
<td>M393A4K40CB2-CTD</td>
<td>Volume: 384 Gb (32 Gb × 12)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Standard: DDR4-2666</td>
</tr>
<tr>
<td>Disk</td>
<td>-</td>
<td>-</td>
<td>Volume: 24 TB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: RAID</td>
</tr>
<tr>
<td>Op. system</td>
<td>Canonical</td>
<td>Ubuntu 18.04.4 LTS</td>
<td>Architecture: 64-bit</td>
</tr>
</tbody>
</table>

B.1 Learning rate
Figure B.19: MADDPG - MPE Learning rate tuning: Physical deception
Figure B.20: MADDPG - MPE Learning rate tuning: Cooperative communication
Figure B.21: MADDPG - MPE Learning rate tuning: Cooperative navigation
Figure B.22: MADDPG - MPE Learning rate tuning: Keep-away
Figure B.23: MADDPG - MPE Learning rate tuning: Predator-prey

(a) 0.01 (default)  
(b) 0.001  
(c) 0.1  
(d) 1  
(e) 10  
(f) 100
Figure B.24: MADDPG - MPE Learning rate tuning: Covert communication
B.2 Replay buffer

Figure B.25: MADDPG - MPE Replay buffer tuning: Physical deception
Figure B.26: MADDPG - MPE Replay buffer tuning: Cooperative communication
Figure B.27: MADDPG - MPE Replay buffer tuning: Cooperative navigation

(a) $1 \times 10^6$ (default)

(b) $1 \times 10^5$

(c) $1 \times 10^4$
Figure B.28: MADDPG - MPE Replay buffer tuning: Keep-away
Figure B.29: MADDPG - MPE Replay buffer tuning: Predator-prey
Figure B.30: MADDPG - MPE Replay buffer tuning: Covert communication
B.3 Discount factor

Figure B.31: MADDPG - MPE discount factor ($\gamma$) tuning: Physical deception
Figure B.32: MADDPG - MPE discount factor ($\gamma$) tuning: Cooperative communication
Figure B.33: MADDPG - MPE discount factor ($\gamma$) tuning: Cooperative navigation
Figure B.34: MADDPG - MPE discount factor (γ) tuning: Keep-away
Figure B.35: MADDPG - MPE discount factor ($\gamma$) tuning: Predator-prey
Figure B.36: MADDPPG - MPE discount factor ($\gamma$) tuning: Covert communication

(a) 0.95 (default)

(b) 0.75

(c) 0.5

(d) 0.35

(e) 0.15
B.4 DNN weights adjustment

Figure B.37: MADDPG - MPE DNN weights adjustment ($\tau$) tuning: Physical deception
Figure B.38: MADDPG - MPE DNN weights adjustment ($\tau$) tuning: Cooperative communication
Figure B.39: MADDPG - MPE DNN weights adjustment ($\tau$) tuning: Cooperative navigation
Figure B.40: MADDPG - MPE DNN weights adjustment ($\tau$) tuning: Keep-away
Figure B.41: MADDPG - MPE DNN weights adjustment ($\tau$) tuning: Predator-prey
Figure B.42: MADDPG - MPE DNN weights adjustment ($\tau$) tuning: Covert communication
B.5 Number of steps between updates

Figure B.43: MADDPG - MPE frequency of DNN updates tuning: Physical deception
Figure B.44: MADDPG - MPE frequency of DNN updates tuning: Cooperative communication
Figure B.45: MADDPG - MPE frequency of DNN updates tuning: Cooperative navigation
Figure B.46: MADDPG - MPE frequency of DNN updates tuning: Keep-away
Figure B.47: MADDPG - MPE frequency of DNN updates tuning: Predator-prey
Figure B.48: MADDPG - MPE frequency of DNN updates tuning: Covert communication
C. YCSB workloads.

Listing C.1: Latest YCSB workload

Listing C.2: Uniform YCSB workload

Listing C.3: Uniform "large" YCSB workload

Listing C.4: Uniform "large" YCSB workload
D. Environment with cooperative scenario evaluation.

In this section we provide details on agents’ behavior in our custom designed data distribution multi-agent environment scenario. In this evaluation we rely on MADDPG [LWT+17] implementation provided in Ray RLlib version 0.8.4 and reference implementation of MAAC [IS18] provided by the authors of the algorithm.

D.1 MADDPG (without action mask) agents behavior in MDDE

The observed behavior of agents per configuration is described below.

0. Agents 0, 1, and 2 selected only do-nothing actions from episode 277 and never recovered. Agent 3 started to repeat the same incorrect action at every step on episode 144. The repeated action would alter every 5-40 episodes.

1. All agents fell into repeating the same incorrect action at episode 63. The repeated incorrect action changed per agent every 20 episodes on average. After episode 605, all agents executed the same incorrect action without alterations.

2. At step 103, all agents fell into repeating the same action with alterations every 40 episodes on average. It’s worth noting, agent 3 at selected do-nothing, which resulted in a positive reward from episode 102 till 201. However later agent 3 started to select an incorrect action repeatedly, which caused it to receive a negative reward but never reverted to doing nothing, which would warrant it a positive reward.

3. All agents fell into repeating the same incorrect action at episode 198. The repeated action would change irregularly with average intervals of 15 episodes for agents 0, 1, 3 and 45 for agent 2. Frequency of the selected action alterations decreases after episode 543, to 150 on average.

4. Agents 2 select an incorrect action repeatedly from episode 48 to 87. After that, agent 2 defaults at doing nothing till episode 509, then seemingly-randomly selects incorrect actions, but not repeating them. Agents 0, 1, and 3 select do-nothing since episodes 53, 49, and 56 respectively, till the end of the run.

5. Agents 0, 1, 2 repeat the same incorrect action starting episode 101. Agent 3 repeats the same incorrect action after episode 103. The repeated action alters every 30 episode on average.
6. All agents repeat the same incorrect action since episode 104. Similarly to previous examples, periodic alteration of the repeated action happens but less frequently with the progress of episodes.

7. All agents repeat the same incorrect action since episode 104. Alterations of the incorrect action repeating happen every 40 episodes on average.

**D.2 MAAC agents behavior in MDDE**

Number of episodes executed for MAAC configurations is as follows:

0. Runtime: 72 episodes.
   Interruption reason: Agents 0, 1, and 3 defaulted to doing nothing. Agent 2 repeatedly executed an incorrect action.

1. Runtime: 70 episodes.
   Interruption reason: All agents repeat the same incorrect action.

2. Runtime: 73 episodes.
   Interruption reason: All agents do nothing.

   Interruption reason: All agents repeat the same incorrect action.

   Interruption reason: All agents repeat the same incorrect action.

5. Runtime: 68 episodes.
   Interruption reason: All agents repeat the same incorrect action.

**D.3 Plots: MADDPG with Action-mask: failed actions**

![Figure D.49: MADDPG with Action-mask: failed actions with configuration 0](image)

Figure D.49: MADDPG with Action-mask: failed actions with configuration 0
Figure D.50: MADDPG with Action-mask: failed actions with configuration 1

Figure D.51: MADDPG with Action-mask: failed actions with configuration 2

Figure D.52: MADDPG with Action-mask: failed actions with configuration 3
Figure D.53: MADDPG with Action-mask: failed actions with configuration 4

Figure D.54: MADDPG with Action-mask: failed actions with configuration 5

Figure D.55: MADDPG with Action-mask: failed actions with configuration 6
Figure D.56: MADDPG with Action-mask: failed actions with configuration 7