COMMUNICATION EFFICIENT MULTI-AGENT REINFORCEMENT LEARNING IN MOBILE WIRELESS NETWORKS

by

Yuqing Cui

December 2022

A thesis submitted to the
Faculty of the Graduate School of
the University at Buffalo, The State University of New York
in partial fulfilment of the requirements for the
degree of

Master of Science

Department of Electrical Engineering

I grant the State University of New York at Buffalo the non-exclusive right to use this work for the University's own purposes and to make single copies of the work available to the public on a not-for-profit basis if copies are not otherwise available.

Yuqing Cui

Table of Contents

List of Figures

Abstract

We consider a collaborative distributed multi-agent reinforcement learning problem with networked agents, where agents are connected via a time-varying communication network and serve as flying nodes between two endpoints. The agents serve as flying base station to establish the wireless connection between the backhaul nodes and users. The agents aim to maximize the overall throughput.

We first investigate the difference between TensorFlow and PyTorch and proceed our research with TensorFlow. Meanwhile, we test the collaborative multi-agent Advantage Actor-Critic (A2C) on Ground Mobility Vehicles (GMVs) with both OpenAI Gym simple spread scenarios and navigation scenario in UBSim over the indoor autonomy research testbed. Then we conduct simulation experiments with networked agents and compared the proposed algorithm with REINFORCE-based Policy Gradient (REINFORCE PG) over the UB SOAR facility with UAVs. Then to further reduce the communication overhead and the real-time gap related to battery charging issue, we utilize a Lazily Aggregated method to set up a trigger condition to recharge the UAV battery based on the endurance, weight and speed. Our work is provided along with the numerical experiments from the perspectives of two different libraries and two different algorithms.

Chapter 1

Introduction

Unmanned aerial vehicles (UAVs, or "drones") have been utilized in many fields and are capable of executing commands, performing search-and-rescue missions [1], establishing a network [2], and provide support for a network [3, 4, 5]. UAVs are also well known as delivery trucks [6] which can help reduce the traffic in civilian life. They can carry large weight and are able to achieve a much faster delivery to satisfy customers. For example, in 2022 researchers introduced fully autonomous mini drones and tested navigation and coordination as a swarm in a bamboo forest [7]. Swarm UAVs are widely used for large-scale computation and large data processing. Swarm UAVs are often deployed in a large area to complete collaborative or cooperative tasks, such as search-and-rescue, data collections, and swarm UAV networking [8]. Swarm UAVs can be used for speed up the convergence of the network policy, quicker access and communication with other UAVs. Instead of having only one UAV exploring the environment, Swarm UAVs will have multiple UAVs to collborate with each other by each scanning a much smaller area, share their data with others to reduce the time cost, and achieve a faster rate of policy conver-

gence. Swarm UAVs are more efficient in transmission time, information processing, communication and energy harvesting. Power consumption is a key performance metric in wireless UAV networking, and using multiple UAVs to speed up the exploring and training process can help reduce the power usage, which is very important to calculate the maximum flight endurance. There are several challenges to address, such as the model design and optimization, resource allocation, and protocol design. The complexity of minimizing communication overhead, resource allocation, model and protocol design motivate and accelerate advancements in analytical model design and data-driven decision-making based on artificial intelligence and machine learning (AI/ML) technology. Model design, model optimization, algorithm implementation, and protocol design are required to design and deploy network control programs and test the experiment on software radios simultaneously.

1.1 Background

Reinforcement learning (RL) [9] is a type of machine learning, where an agent takes actions in the environment over a sequence of time steps and aims to maximize the long-term cumulative reward or minimize the long-term cumulative loss it receives from the environment. In a typical RL problem, there is at least one agent, exploring or exploiting its surrounding.

The agent takes actions in the environment in exchange for reward for the information of that and next states, then the agent proceeds another action based on these information. The agent aims to maximize the long term reward in the environment. The way that the agent interacts with the environment continually for a certain amount of time and learns based on the feedback from the

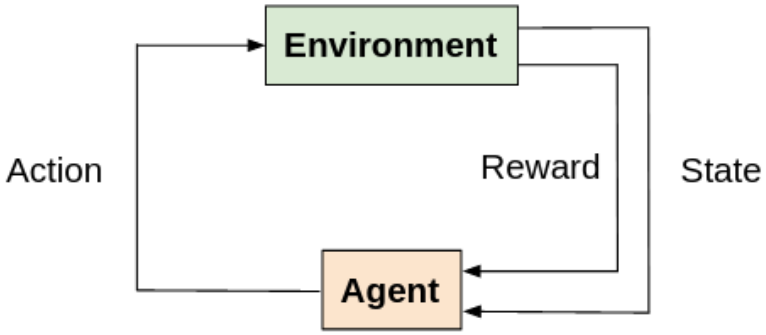


Figure 1.1: Agent-environment interaction in a MDP.

environment is called Markov Decision Process (MDP). The MDP consists a tuple:

$$(S, A, \rho, P, R) \tag{1.1}$$

where s $\in$ S is the state space, a $\in$ A is the action space, ρ is the initial state distribution, P is the state transition kernel, and R is the reward received from interacting with the environment. The agent aims to maximize the long-term cumulative reward or minimize the long-term cumulative loss by the state information received from interacting with the environment.

$$\max_{\pi} \sum_{m \in M} R_m(\pi) \text{ with } R_m(\pi) := \mathbb{E}_{T \sim P(\cdot|\pi)} \sum_{t=0}^{\infty} \gamma^t r_m(s_t, a_t) \tag{1.2}$$

In this research, we consider multiple agents in RL, also called Multi-Agent RL (MARL), where instead of having one agent interacting with the environment, multiple agents can work together. In this case, there are two types of MARL that will affect agents' behaviors, cooperative and collaborative MARL. In cooperative MARL, the agents work together to achieve their own goals. In collaborative MARL, the agents work together, have individual progress,

and aim to achieve the same shared goal. Both learning strategies can be utilized in either a centralized method or a decentralized method. A centralized method involves a central controller and a group of agents, and at each time step, the agents update their policy gradient and receive the next state information and reward from the controller. Unlike the centralized method, decentralized method does not have a controller to keep track of all updated information from the agents. In decentralized method, the agents are linked via a time-varying communication network and share the required information with their neighbors. Both state and reward information is localized, while in centralized method, state and reward information may be globally observable depending on the requirement of the tackle scenario.

We aim to reduce the communication overhead for collaborative MARL. In a multi-agent network, for example, multi-UAV networks, which are also called Swarm UAV networks that often suffer from transmission latency, communication, and data overhead due to the frequent information exchange among UAVs through a time-varying communication network and many unpredictable environmental factors that affect the convergence of the learned policy.

1.2 Challenges

There are many challenges in MARL, such as communication overhead and signal interference, limited endurance, data overhead, sim-to-real gap, and swarm control. Except for the last one, which is caused by hardware, each MARL challenge can be discussed from three basic categories: environment, policy, and networked control problem.

The environment can cause challenges based on data complexity and decision-

making. When multiple agents interact with the same environment, the state information received by each agent will include the current agent's information and other agents' information. This information depends on the scenario and the network control problem. Each agent will then process and take actions based this information and reward. With multiple agents, we often expect them to cover a large and complex area, this can increase the state data complexity. It can lead to inaccurate simulation results due to many unpredictable factors and changes in other agents' behavior in a real environment.

Regrading the policy, due to large amount of information that require to be processed, individual behavior is hard to predict and will interfere with other agents, such as collision and misinformation. It can be more challenging when having a decentralized system, since the agents will be required to learn information from each other, since information can be misunderstood or failed transmission can make the policy even hard to converge.

In a network control problem, there are many things that need to be considered, such as communication links and scenarios (bandwidth and data rate). Each agent requires lots of communication with its neighbor agents to maintain a controllable system and this will lead to large communication overhead and latency.

More detailed MARL challenges are listed below and discussed based on the three categories mentioned above. We consider a basic navigation scenario as an example to help the discussion of each challenge. The navigation scenario, also called seeking scenario, where N agents seek N landmarks. The agents will receive rewards based on the distance to the landmarks and other agents. In this case, the state for each agent will include its location coordinates, distance to the closet landmark, and distance to the closet agent. To avoid confusion, we

consider agent i as the current agent and agent j as the other agent.

1. Communication Overhead / Signal Interference

   The communication overlap and signal interference can be caused by exploring the surrounding environment, training policy, and communication network. In typical MARL, the agents need to frequently communicate with the central controller or other agents and receive the needed information to take the next action. Two methods can affect the communication differently, centralized method and decentralized method. Next we will discuss the communication overlap and signal interference for each method for each category.

   In a centralized method, a central controller is needed to collect all the information from every agent and send out the observation and reward to help the agents take the next step action. In centralized method, the agents do not need to communicate with other agents. Each agent i chooses an action based on the initial state information received from the central controller. Based on this action, the central controller receives location information and calculates the distance to the closest landmark and closest agent j. As the number of agents increases, the amount of location information received from the environment at each time step will also increase. The required distances for calculating the reward will require more time to process. Since the agents take actions individually, their actions are hard to predict, and the training policy takes time to converge. If considering a network control problem with a centralized method, agent i is linked to users and backhaul stations but not to agent j.

   In a decentralized method, since there is no central controller, the

agents share their information with their neighbors via a time-varying communication network. Each agent receives their state information from the environment and updates through this communication network to their neighbors. In this case, each agent's state information is affected by two factors, how accurate the shared information is and how stable this communication network is. During the information exchange, there is some noise and signal interference that can cause lost transmissions. This will make the policy training even harder to converge.

To reduce the signal interference, communication cost, time cost, and the communication frequency need to be reduced. Otherwise, the state information will be lost and agents' actions will be negatively affected.

2. Data Overhead

In both centralized and decentralized methods, the agents will need sufficient state information in order to take an action. This information increases as the number of agents increases, which will increase the amount of data that need to be processed and hence the cost of time.

3. Swarm Control

Swarm control algorithm is really important, especially in decentralized MARL. Since there is no central controller, a control method is needed to maintain the swarm formation while avoiding collisions with each other or the environment. Swarm control framework includes flight control and cross-layer optimization designs. This is required to coordinate large-scale UAV systems to reduce the execution time.

4. Hardware Challenge: Limit Endurance

UAVs have a limited payload that they can carry: as the weight of UAVs increases, the maximum flight time decreases, due to increased strain on the battery. In order to increase the flight time, UAVs will require a larger battery. Hardware problems can cause all the above challenges, due to a sensor malfunction, mislocation, and wrong configuration. UAVs rely on their sensors to conduct data collection. If there is a sensor malfunction occurs, UAVs might be misled and requires directions or command to fly back to the maintenance station. Important sensor malfunctions, like GPS sensor, can also lead to data inaccuracy, which will be useless to transfer. Transferring this kind of information can increase communication cost.

Both the challenges and utilization of the swarm UAVs networks have motivated our research.

1.3 Contributions

The goal of this research is to study a communication-efficient multi-agent reinforcement learning algorithm that can reach a maximum throughput with less communication overhead. In this work, we consider a basic networked agent scenario, where there is a starting point, a destination point, and 2 agents to set up the connection between two points.

1. In the first step of the research, we want to understand the difference between TensorFlow and PyTorch and conduct experiments to see the code changes. We want to identify a library that has more supporting documents and tutorials for future research. We evaluate the difference be-

tween TensorFlow and PyTorch by testing both libraries with the REINFORCE-based Policy Gradient (REINFORCE PG) method with the OpenAI Gym simple spread scenario.

2. In the second step, instead of using existing scenarios in OpenAI GYM, we design our own scenarios and MDPs. We adapt UBSim, a network simulator with integrated optimization, learning, and experimentation capabilities for digital-twin-enabled wireless networks. We add a set of new node classes, including flying class and landmark class to further compare the convergence of the algorithms in the navigation scenarios in OpenAI Gym and UBSim.

3. We implement a centralized Multi-Agent Advantage Actor-Critic (A2C) method in UBSim with the navigation scenario using the indoor autonomy research testbed. By comparing the results of Multi-Agent A2C and REINFORCE PG, it is found that A2C converges faster than REINFORCE PG, which could be beneficial for future research.

4. We model the UB Structure for Outdoor Autonomy Research (SOAR) in UBSim for outdoor simulation experiments and further test it with all the algorithms. For a swarm UAV system, the agents are often searching in a large area where the indoor autonomy research testbed is small compared to the SOAR facility. The SOAR facility will be utilized for all future simulations and real-world experiments involving multiple agents.

Chapter 2

Related Work

2.1 Overview

There are two regimes of RL that involves multiple agents, Distributed Multi-Agent Reinforcement Learning (Distributed MARL) [10] and Parallel Multi-Agent Reinforcement Learning (Parallel MARL) [11]. In Distributed MARL, all agents conduct training in the same environment and each agent's actions can be influenced by other agents. Distributed MARL is used for a large area investigation and performs tasks that require collaboration or cooperation among agents. In Parallel MARL, all agents work in parallel, which means the agents train in a single-agent environment and complete their single-agent tasks. Parallel MARL can reduce the training time and data complexity.

Additionally, there are two methods that can be considered on top of MARL, centralized method and decentralized method. Th former involves a central controller to keep track of all data and coordinate the information transferred between agents. The latter does not have any controller to coordinate the transition of information, including action, observation, and reward. Instead, in

decentralized method, the agents are connected and they transfer information to fulfill their observation via a time-varying communication network in order to calculate their local reward. Lastly, we also compared two machine learning libraries in this section, TensorFlow and PyTorch.

2.2 Centralized MARL

The centralized MARL [12], similar to the centralized method briefly discussed in Chapter 1, has one central controller with multiple agents. These agents are connected to the central controller by communication links. These communication links are all bidirectional, which means theagents can upload information to central controller, and central controller can broadcast information to the agents.

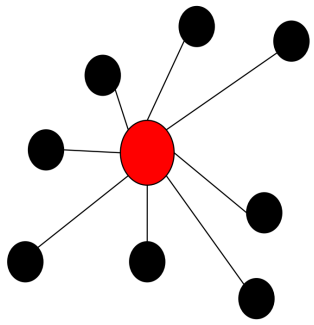


Figure 2.1: A central controller linked with 8 agents.

2.3 Distributed Multi-Agent Reinforcement Learning (Distributed MARL)

In Distributed Multi-Agent Reinforcement Learning (Distributed MARL)

[10], instead of having a central controller to connect to the agents like centralized MARL, the agents are connected to their neighbors via a time-varying communication network. The agents jointly optimize the policy using the joint action of all the agents.

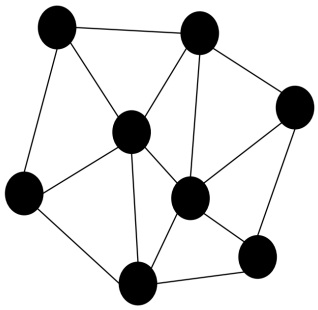


Figure 2.2: Eight agents that are connected to their neighbors by time-varying communication links.

In typical distributed MARL, multiple agents interact with each other within the same environment, which consists of new tuples:

$$(S, \{A_m\}_{m \in M}, \rho, P, R) \tag{2.1}$$

where $\{A_m\}_{m \in M}$ is the joint action of all the agents at each time step, which is required to determine the next state information. This will further affect the long-term cumulative reward function:

$$\max_{\pi} \sum_{m \in M} R_m(\pi) \text{ with } R_m(\pi) := \mathbb{E}_{T \sim P(\cdot|\pi)} \sum_{t=0}^{\infty} \gamma^t r_m(s_t, a_{n,t}) \tag{2.2}$$

where $n = 1, 2, ..., M$ and it is used to calculate the joint reward at each time step. The advantage of Distributed MARL is data efficiency because multiple agents work together to reduce the amount of data that needs to be processed.

In distributed MARL, there are multiple agents interacting with each other

in the same environment. There are different learning theories. For instance, in collaborative learning, the agents take individual actions and aim to maximize the overall reward, but each agent can receive the individual or global reward at each time step. In cooperative learning, unlike collaborative learning, the agents work together. Each agent has an important role and each role plays an important part to achieve the maximum global reward. In competitive learning, on the other hand, the agents compete with others to achieve a maximized result. In summary, they all seek to maximize the reward in that environment. The only difference is, the agents in collaborative learning can have different actions and different rewards; the agents in cooperative learning must have others' contributions and will receive the same reward; and lastly, the agents in competitive learning can have different rewards, actions, and states.

2.4 Parallel Multi-Agent Reinforcement Learning (Parallel MARL)

Unlike the distributed MARL, parallel MARL[11] divides the large-scale complex tasks into multiple single-agent task and has all the single-agent tasks running in parallel to reduce the computational complexity of the original and speed up the training process.

$$(S_m, A_m, \rho_m, P_m, R_m) \tag{2.3}$$

where compare to the distributed MARL, each agent has its own state, action, initial state, state transition kernel, and reward. Thus, a different long-term

cumulative reward function shown as follow:

$$\max_{\pi} \sum_{m \in M} R_m(\pi) \text{ with } R_m(\pi) := \mathbb{E}_{T_m \sim P(\cdot|\pi)} \sum_{t=0}^{\infty} \gamma^t r_m(s_{m,t}, a_{m,t}) \tag{2.4}$$

The advantage of parallel MARL is improved time efficiency and lower computational complexity. Since the complex task is divided into multiple small and less complex tasks, the training time can be reduced [**?**]. [13] also shows that training policy in parallel can speed up the training process and help to process data faster.

2.5 TensorFlow vs. PyTorch

This research consider two different open-source software libraries for machine learning development, TensorFlow [14] and PyTorch [15]. TensorFlow is developed by Google and written in C++, CUDA, and Python. TensorFlow is an end-to-end machine learning platform that has multiple tools to process and load data. TensorFlow can be used for model and algorithm design. It can support model iteration and distributed training. TensorFlow has Model Analysis and TensorBoard to help track training performance. Some pre-trained models can be found in TensorFlow Hub to help beginners. TensorFlow can support deploying the model on different environments, such as CPUs, GPUs, FPGAs, servers, and edge devices. PyTorch is developed by Meta AI, which is under Meta Platform, also known as Facebook. It is based on the Torch framework and can also be written in C++, CUDA, and Python. PyTorch also has many tools that can be used for data processing, model design, and algorithm design. PyTorch can support distributed training and performance optimization by us-

ing the torch.distributed. It also has support on major cloud platforms.

Since TensorFlow is released in Nov. 2015 which is almost a year earlier than PyTorch, there are more existing documentation and coding examples than PyTorch. To code a simple expression, TensorFlow has more libraries and sources to use than PyTorch, which requires more hard coding. This type of data parallelism made TensorFlow hard to debug than PyTorch. Another difference between TensorFlow and PyTorch is in the visualization. TensorFlow has a tool called Tensorboard that can display the summary of collected data in many different ways, plots, images, and audio, whereas the visualization for PyTorch is very limited. A more organized comparison is listed in Table 2.1.

	Comparison	**TensorFlow**	**PyTorch**
1	Developer	Google	Facebook
2	Written In	C++, CUDA, Python	C++, CUDA, Python
3	Release Date	Nov. 2015	Sept. 2016
4	Data parallelism	Support for asynchronous execution	Manually code
5	Debugging	Difficult to conduct debugging	Good debugging capabilities
6	Visualization	Better visualization	Limited

Table 2.1: Comparison table between TensorFlow and PyTorch

Chapter 3

Model and Algorithm Design

3.1 Overview

This chapter discusses two key elements of this research, scenario design, and algorithms design. Scenarios are the environment to conduct the training performance evaluation. Scenarios decide the MDPs and the agents' goals. Then two different algorithms are used for training performance evaluation.

3.2 Networked Agents

We considered a network system with M agents, each agent serves as a flying base station and connect to the ground backhaul stations and users. We also considered a centralized setting where the agents communicate with the central controller for rewards and state information. The agents are connected through a time-varying communication network, denoted by $x_t = (M, y_t)$, where x_t is a communication network without communication directions and y_t is the sets of communication links at time t. $(i, j) \in y_t$ means agent i and agent j are con-

nected and can share the network information. This communication network is used to establish network connections to serve users and aim to achieve the maximum throughput.

Algorithm 1: The Networked Agents System

Data: Input: $total\_eps, T$
for *each episode eps* **do**
  Initialize connections
  for $i \in M$ **do**
    Send *packets* to its neighbor $j \in M : (i, j) \in y_t$ over the communication network x_t
  end
end

3.3 REINFORCE-Based Policy Gradient

REINFORCE-based Policy Gradient (REINFORCE PG) [9] is a Policy Gradient (PG) method, which is considered a policy-based method. It is used to estimate the optimal policy's weight by stochastic gradient ascent to increase the probabilities of the high-value actions. REINFORCE is based on trajectory, which is for episodic cases. The trajectory corresponds to a full episode, also called a Monte Carlo algorithm. It aims to maximize the expected return. The policy gradient expression for REINFORCE PG is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(s, a) R_t] \tag{3.1}$$

where θ is the policy parameter, π_θ is a parameterized policy, and R_t is the sum of the reward.

Although as a stochastic PG method, the convergence and performance of

Algorithm 2: REINFORCE-Based Policy Gradient (episodic)

Data: Input: learning rate $\alpha > 0, total\_eps, T$

Initialize: θ;

for *each episode eps* **do**

  $(S_0, A_0, R_1, ...S_{T-1}, A_{T-1}, R_T)$;

  for $t = 1$ *to* $T - 1$ **do**

    $\theta \leftarrow \theta + \alpha\gamma^t R_t \nabla_\theta \log \pi_\theta(A_t|S_t, \theta)$

  end

end

REINFORCE PG are promising, it has a high variance and requires longer training time compared to Actor-Critic method.

3.4 Advantage Actor-Critic (A2C)

Advantage Actor-Critic (A2C) [16] is an algorithm that combines both policy-based and value-based reinforcement learning algorithms. A2C is a temporal difference (TD) method that consists of two neural networks, actor network, and critic network. This means A2C has both actor step and critic step as separate memory structures. The actor controls each agent's behavior and the critic evaluate s the action based on a state-value function:

$$A_\pi(s, a) = Q_{\pi_\theta}(s, a) - V_{\pi_\theta}(s) \tag{3.2}$$

where $A_\pi(s, a)$ is the advantage function, $Q_{\pi_\theta}(s, a)$ is the action value function, and $V_{\pi_\theta}(s)$ is the state value function. The actor learns from this subtraction between the state action pair and the average value of that state. This advantage function is the critic evaluation, which is also called the TD error. It can be

rewritten as:

$$A_\pi(s,a) = R + \gamma V_{\pi_\theta}(s_{t+1}) - V_{\pi_\theta}(s) \tag{3.3}$$

This equation can be used in place of the action value function by using only one neural network, so that the variability in predictions can be reduced.

The Agents learns from advantage function which is the actual rewards instead of the average rewards. Then A2C updates the policy using the advantage function $A_\pi(s,a)$. The update expression for A2C is shown below:

$$\nabla_\theta = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) A_\pi(s,a)] \tag{3.4}$$

The advantage of using A2C instead of REINFORCE PG is lower variance with increased stability because of this advantage function.

Algorithm 3: Advantage Actor-Critic

Data: Input: learning rate $\alpha > 0$, learning rate $\beta > 0, total\_eps, T$
Initialize: θ;
for *each episode eps* **do**
  $Initializes_t = s_0$
  for *time step* $t = 1$ *to* $T - 1$ **do**
    Current action: $a_t \sim \pi(\cdot|S,\theta)$, observe s_{t+1}, R_t
    Advantage function: $A_t \leftarrow R_t + \gamma^t V(s_{t+1}, w) - V(s_t, w)$
    State-value parameter $w \leftarrow w + \beta A_t \nabla V(s_t, w)$
    Policy parameter $\theta \leftarrow \theta + \alpha \gamma^t A_t \nabla \log \pi(a_t|s_t,\theta)$
    $s_t \leftarrow s_{t+1}$
  end
end

Similarly, Asynchronous Advantage Actor-Critic (A3C) [17], which based on the A2C, maintains all the details of the A2C, but involves executing in parallel when interacting with the environment. A3C can improve the robustness and

scalability and also improve the convergence of policy.

Chapter 4

Experimental Evaluation

In this chapter, two platforms, UBSim and OpenAI Gym, are evaluated and compared based on the same training parameter for future model and algorithm designs under the same scenario. These experiments are conducted using REINFORCE PG as the learning algorithm. Then, the SOAR facility is configured into UBSim and is compared with the indoor testbed, with an additional class design for scenario design. Another experiment is conducted to see the relationship between changing the number of agents and reward. Finally, the limitations of MARL is investigated to understand the relationship between the agents and training time.

4.1 TensorFlow vs. PyTorch

For MARL, the computing device is really important to help accelerate the convergence of training policy. PyTorch requires an extra line of code to check if the device is available and device conversion. All data needs to use tensor.to(device) for device such as cuda0. For TensorFlow, we can install the GPU

version of TensorFlow and configure the relevant GPU device drivers. Then, TensorFlow will run operations automatically on GPUs, this can also be controlled by coding. GPUs can process more data than CPUs with less power consumption. Using GPUs can accelerate the training policy convergence because of GPUs' parallel processing abilities. Therefore we choose to use TensorFlow for the MARL algorithm design.

4.2 Simulation Testing

In this section, we utilize two simulation software to test our experiment, OpenAI Gym [18] and UBSim [19]. We consider the same scenario (N agents seeking N landmarks) for both software with training algorithms, REINFORCE PG. We consider the same parameters to reduce the gap between the two software. The simple spread scenario in OpenAI Gym is similar to the navigation scenario, where their scenarios contain two agents and two landmarks. Each agent receives the initial state information containing the agent's current location, distance to the closest landmark, and the distance to the closest agent from the environment. In this case, since the two agents interact with the same environment, the distance to the closest agent will be the distance to the other agent. The agents will then choose their actions based on this state information. Based on this action, the environment will pass the new state information after taking the action and reward to critic if this action worths it or not. The agents will receive negative rewards based on two categories, distance between agents and landmarks and existing collision. The agents learn to minimize the loss by minimizing the distance between agents and landmarks and avoiding collisions.

4.2.1 OpenAI Gym

OpenAI Gym is an open source python library, which is a standard API for testing the reinforcement learning algorithms and often comes with lots of existing scenarios along with the pre-defined MDPs, including action, step function, observation, reward, and evaluation for the agent's achievement. The step function in OpenAI Gym is using for taking actions and it returns the next observation, reward, and end-of-episode information as the output.

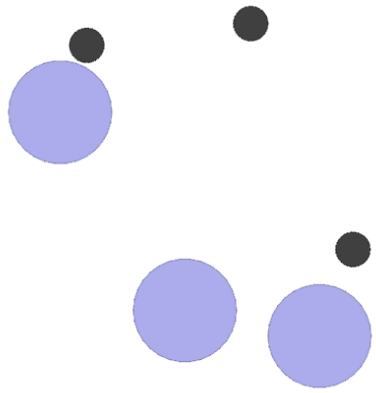


Figure 4.1: Simple spread scenario in OpenAI Gym with three agents (purple) and three landmarks (black).

Figure 4.1 shows three agents aiming to reach three landmarks. The agent in the middle will collide with the agent at the bottom of the figure since the bottom landmark is the closest to both agents. To avoid this situation, redefining the reward function is needed. By increasing the collision penalty, an agent will more likely avoid colliding with each other than reaching the same landmarks

at the same time. The amount of the collision penalty can change agents' behaviors, therefore changing the average reward per iteration and hence changing the speed of policy convergence.

4.2.2 UBSim

UBSim is a discrete event-driven network simulator that can support UAV and ground robot networks with three different frequency bands, microwave, millimeter-wave, and terahertz bands.

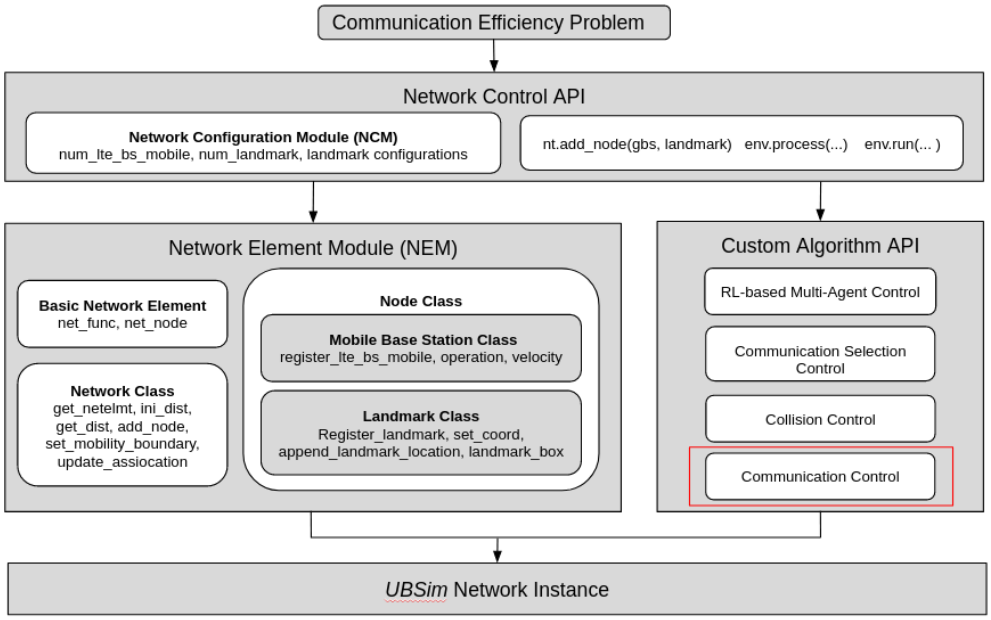


Figure 4.2: Architecture of UBSim simulator for navigation environment.

Figure 4.2 shows the architecture of the UBSim simulator for the navigation scenario, which is N agents seeking N landmarks. The agents are defined in the mobile base station class. This class has a set of basic operations, such as passing agent's current coordinate, velocity, and pause time. Landmarks are defined in the landmark class, which has a similar functionality without velocity since the landmarks are stationary. Landmark class has a unit landmark_box, which

is used to calculate the shape of the landmark to display in the GUI. Custom algorithm API section is used for the user to define their own control algorithms, such as collision control to avoid collision with other irrelevant obstacles, like walls and nets to help prevent great damage.

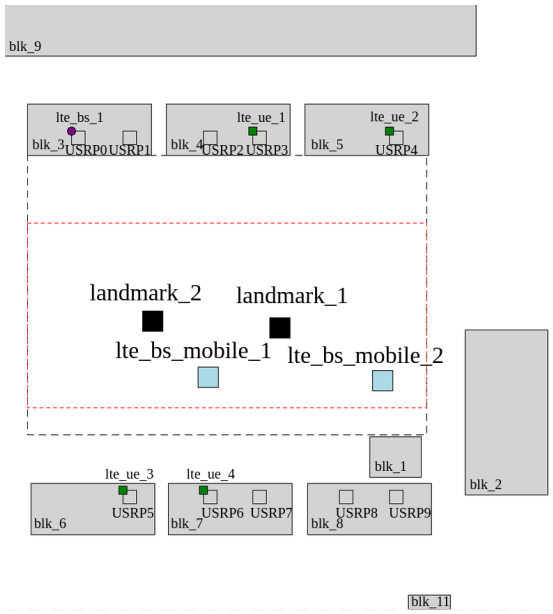


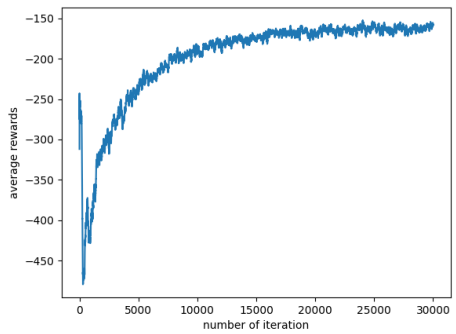
Figure 4.3: Navigation scenario simulation conducted over the indoor testbed.

Figure 4.3 shows a similar navigation scenario as the OpenAI Gym in UBSim. UBSim is based on SimPY, a process-based discrete-event simulation framework. UBSim can synchronize the visualization of the environment for every time step, which cannot be achieved using OpenAI Gym.

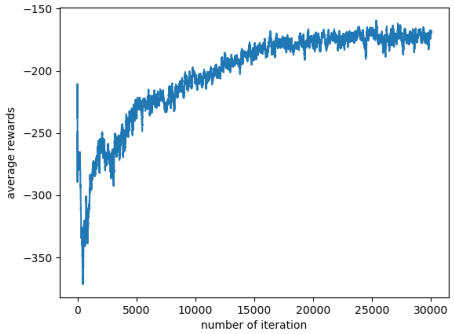
4.2.3 Result Comparison

Since the number of agents needs to match the number of landmarks, in this case, two agents that are seeking to reach two landmarks without collision. The training policies for both simulation software use the same neural size for both hidden layers, in which the first layer contains 128 neurons and the second layer

contains 64 neurons. The discount factor γ is 0.95 and the learning rate α and β are set to 0.00045. The total number of episodes is 30000 with 25 time steps per episode.



(a) Averaged reward vs. iteration plot for the OpenAI Gym simple spread scenario.



(b) Averaged reward vs. iteration plot for the UBSim navigation scenario (similar to simple spread).

Figure 4.4: Comparison between two software.

Based on the result shown in Figure 4.4 (a) and (b), both experiments using exactly the same training parameters but different software platforms can reach the same amount of reward. Both training policies converge between 10000 and 15000 iterations. If the training parameters are the same, the difference between this two software is minor. Therefore, using UBSim is efficient and can be more helpful for debugging purposes, since UBSim is based on SimPy.

4.3 More Experiments

After we evaluate the simulation platform, we further configure the SOAR facility into the UBSim. The space of the indoor testbed [20] is limited and cannot support multi-UAV real-world experiments. The trained policy based on the indoor testbed will be invalid and will need to be retrained. The SOAR facility

is much more suitable for the MARL model and algorithm design.

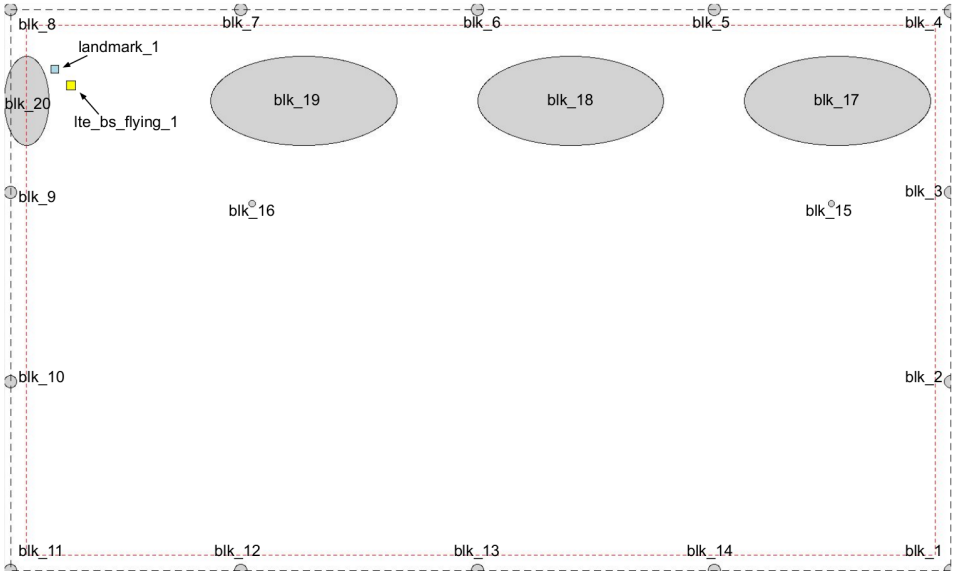


Figure 4.5: Navigation scenario simulation conducted over the SOAR facility.

Figure 4.5 illustrates SOAR facility in the UBSim. The two small squares on the top left corner are the agent node and landmark node. They are the same size as shown in Figure 4.3. This indicates that the SOAR facility is a much larger facility and can be used for large-scale outdoor testing.

Figure 4.6 is the architecture of the UBSim simulator based on the SOAR facility. Agents are now considered UAVs with almost the same functionality as mobile base station nodes in 2-D, without considering the height. There are many reasons can result in a height shifting during flight, such as environmental issue and collision avoidance, thus, recalculation is needed. The distance can be calculated using the get_dist API in the network class.

These figures above are a comparison between the different numbers of agents with the same learning rate $\alpha = 0.0007$ under the same navigation environment. In this scenario, the reward is the loss and the agents aim to minimize the loss. In Figure 4.7 (a), only one agent interacts with the environment, and the train-

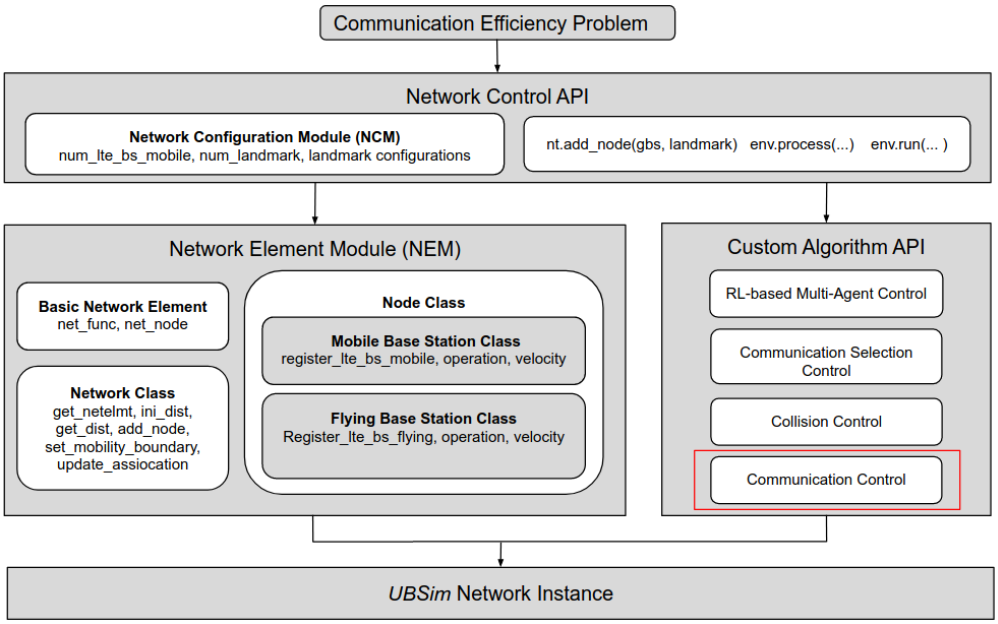
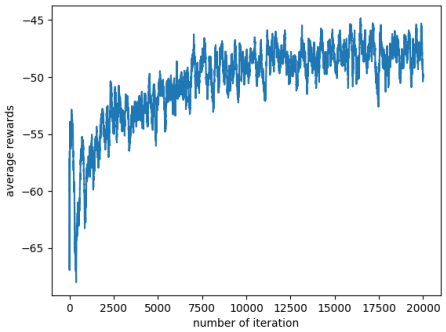


Figure 4.6: Architecture of UBSim Simulator for SOAR with UAVs.

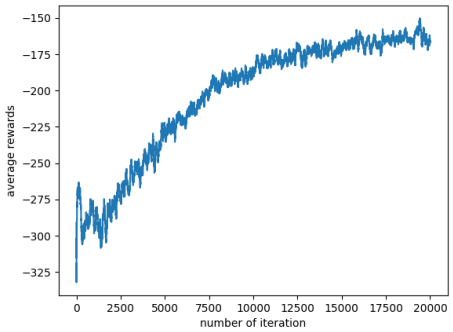
ing converges in around 10000 iterations. The average reward for a single-agent scenario is -50. Figure 4.7 (b) considers two agents interacting with the environment. This means both the collision reward and the reward reflect on how close an agent is to a landmark is considered. For a two-agent scenario, it converges in a little over 10000 iterations with a reward of -175. In Figure 4.7 (c), there are three agents and it converges around 25000 iterations with a reward of -400. As the number of agents increases, the reward decreases exponentially and requires more iterations to train the policy.

4.4 Limitations

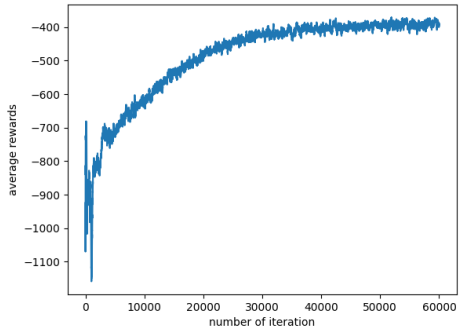
There are many challenges to address in MARL, such as the time cost to train a policy, how fast can a policy converged and how to reach the maximum reward.



(a) One Agent

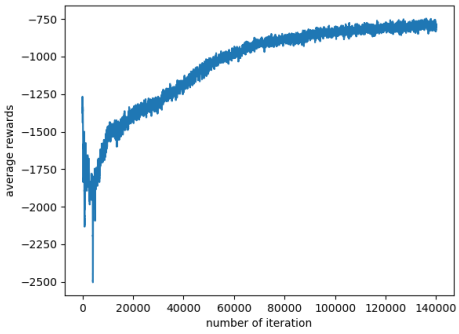


(b) Two Agents

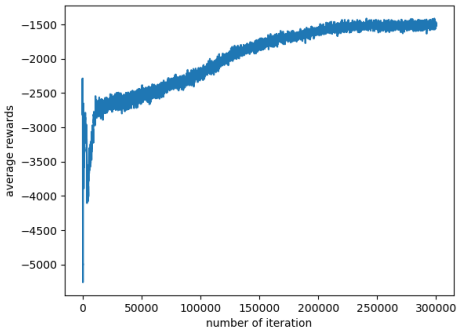


(c) Three Agents

Figure 4.7: Comparison between different number of agents



(a) 4 agents scenario takes 10 hours to train.



(b) 5 agents scenario takes 24 hours to train.

Figure 4.8: Time cost for 4 agents and 5 agents scenarios

In MARL, time cost is also a challenge. As the number of agent increases, the training time for a policy to converge will also increase. In Figure 4.8 (a), a scenario has 4 agents takes 16 hours to complete 140000 iterations. It converges at around 8000 iterations which takes approximately 10 hours to train. In Figure 4.8 (b), this scenario has 5 agents. It takes 34 hours to complete 300000 iterations and converges at around 170000 iterations taking around 24 hours. The training time can take much longer in real scenario. Therefore working in parallel will be more important to reduce the sim-to-real gap.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

We considered a collaborative distributed Multi-Agent Reinforcement Learning problem with networked agents. We considered a wireless network with two base station as the two endpoints and N agents deployed in between to maximize the throughput. We have tested the collaborative multi-agent Advantage Actor-Critic (A2C) on Ground Mobility Vehicles (GMVs) with both OpenAI Gym simple spread scenario and navigation scenario in UBSim over the indoor autonomy research testbed. Then we configured the SOAR into the UBSim and compared the benefits with the indoor autonomy research testbed.

5.2 Future Work

There are many works that can be added on top of this thesis. We proposed a centralized collaborative Multi-Agent Reinforcement Learning algorithm with networked agents, which can be further considered in a fully decentralized set-

ting, where the agents are not only connected by the time-varying communication network to establish connections but also communicate during this connection to exchange information between the agents.

- **Lazily Aggregated (LA) Method**

  Lazily Aggregated method [21] is used to compare the previous policy to the current policy and evaluate the weight of difference, then update the current policy when the difference exceeds a situational threshold:

$$||\delta\hat{\nabla}_n^k||^2 \geq \frac{c}{\alpha^2 N^2}\sum_{d=1}^{D}||\theta^{k+1-d}-\theta^{k-d}||^2 + 6\sigma^2_{n,total\_eps} \tag{5.1}$$

  where $\delta\hat{\nabla}_n^k$ represents the current non-time-critical information and use to compare with the updated boundary, c is a coefficient that is used to evaluate how important this information will be and will set for the lower bound to trigger the update. The LA method can be used to update non-time-critical information, for example, sensor functionality and battery health.

  The LA method can reduce the policy update frequency based on this trigger condition, therefore the communication frequency can be reduced. More problems that need to be addressed before more general implementation, for example how to maintain fairness while trying to maximize the throughput. A selection algorithm will be designed in order to maintain fairness between the agents, similar to the LA method, in which data updated by the agents will be evaluated based on importance and urgency. For example, a UAV that runs with a low battery or sensor malfunction is much more urgent than other healthy agents.

- **Partially Observable Markov Decision Process (POMDP)**

  A Partially Observable Markov Decision Process (POMDP) [22] can be used to reduce the communication overhead. Instead of traditional MDP, which assumes full knowledge of state, action, and transition probability, POMDP is instead defined by conditional transition probability and a set of conditional observation probabilities, where the agents take actions based on a conditional observation, and the policy is trained based on a probability distribution of the underlying state from the environment.

$$(S, A, O, T, P, R, \gamma) \tag{5.2}$$

  where O is the observation that uses to take actions, T is a set of conditional transition probabilities between states, and P is a set of conditional observation probabilities. Since the agents takes actions based on less information than the full state, the computational complexity of POMDP is less than MDP.

- **Learning Fairness**

  For a decentralized MARL, the agents are all required to communicate with their neighbors through a time-varying communication network. The agents cannot transmit at the same time over the same channel, otherwise this will cause interference and possible communication loss. To determine which agent communicates first it will require a separate fairness algorithm. This algorithm will prioritize those agents who are exchanging either very valuable information or urgent information such as a low battery, then further rank them to reduce the traffic. The fairness algorithm

can make sure the agents will not lose important messages due to communication loss or interference since those messages are always considered the top priority.

- **Sim-to-Real Gap**

  All the above future work intends to either reduce or help the communication network for information exchange, but neither are considered if the simulation data are inaccurate. Inaccurate training data can lead to a large sim-to-real gap. To further reduce the sim-to-real gap, we plan to adapt digital construction [19] with a hybrid simulation parameter, which will use the real-time data as input data of the simulation to reduce the sim-to-real gap.

- **Time efficiency**

  Training in simulation and using this trained policy to train in the real-world can take an infinite amount of time. To reduce the sim-to-real gap it needs to receive real-time data as input data for simulation. Since this process requires lots of time, we plan to learn in parallel. In this case, we can train the same untrained policy at the same time. Then, configure the trainable parameters in simulation and send the trained policy to test in the real scenarios. We can use the real data to continue training the policy in simulation. In this case, we can generate an iteratively improving training result in both cases.

Bibliography

[1] Mario Silvagni, Andrea Tonoli, Enrico Zenerino, and Marcello Chiaberge. Multipurpose uav for search and rescue operations in mountain avalanche events. *Geomatics, Natural Hazards and Risk*, 8(1):18–33, October 2016.

[2] Xiao Liu, Yuanwei Liu, and Yue Chen. Reinforcement learning in multiple-uav networks: Deployment and movement design. *IEEE Transactions on Vehicular Technology*, 68(8):8036–8049, June 2019.

[3] S. Krishna Moorthy and Z. Guan. FlyTera: Echo State Learning for Joint Access and Flight Control in THz-enabled Drone Networks. In *Proc. of IEEE International Conference on Sensing, Communication and Networking (SECON)*, Como, Italy, June 2020.

[4] S. Krishna Moorthy and Z. Guan. LeTera: Stochastic Beam Control Through ESN Learning in Terahertz-Band Wireless UAV Networks. In *Proc. of IEEE INFOCOM Workshop on Wireless Communications and Networking in Extreme Environments (WCNEE)*, Toronto, Canada, July 2020.

[5] S. Krishna Moorthy, M. McManus, and Z. Guan. ESN Reinforcement Learning for Spectrum and Flight Control in THz-Enabled Drone Networks. *IEEE/ACM Transactions on Networking*, October 2021.

[6] Jennifer Shang Wen-Chyuan Chiang, Yuyu Li and Timothy L. Urban. Impact of drone delivery on sustainability and cost: Realizing the uav potential through vehicle routing optimization. *Applied Energy*, 242:1164–1175, May 2019.

[7] Xin Zhou, Xiangyong Wen, Zhepei Wang, Yuman Gao, Haojia Li, Qianhao Wang, Tiankai Yang, Haojian Lu, Yanjun Cao, Chao Xu, and Fei Gao. Swarm of micro flying robots in the wild. *Science Robotics*, 7(66):eabm5954, May 2022.

[8] J. Hu, S. Krishna Moorthy, A. Harindranath, Z. Guan, N. Mastronarde, E. S. Bentley, and S. Pudlewski. SwarmShare: Mobility-Resilient Spectrum Sharing for Swarm UAV Networking in the 6 GHz Band,". In *Proc. of IEEE International Conference on Sensing, Communication and Networking (SECON)*, Virtual Conference, July 2021.

[9] RS Sutton and AG Barto. *Reinforcement Learning: An Introduction. Cambridge, MA, USA: A Bradford Book*. The MIT Press, November 2017.

[10] Matthew W. Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Nikola Momchev, Danila Sinopalnikov, Piotr Stańczyk, Sabela Ramos, Anton Raichuk, Damien Vincent, Léonard Hussenot, Robert Dadashi, Gabriel Dulac-Arnold, Manu Orsini, Alexis Jacq, Johan Ferret, Nino Vieillard, Seyed Kamyar Seyed Ghasemipour, Sertan Girgin, Olivier Pietquin, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Abe Friesen, Ruba Haroun, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning, June 2020.

[11] Teng Liu, Bin Tian, Yunfeng Ai, Li Li, Dongpu Cao, and Fei-Yue Wang. Parallel reinforcement learning: a framework and case study. *IEEE/CAA Journal of Automatica Sinica*, 5(4):827–835, May 2018.

[12] Xueguang Lyu, Yuchen Xiao, Brett Daley, and Christopher Amato. Contrasting centralized and decentralized critics in multi-agent reinforcement learning. December 2021.

[13] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, November 2017.

[14] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, November 2015. Software available from tensorflow.org.

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., September 2016.

[16] Xiao Han, Jing Wang, Qinyu Zhang, Xue Qin, and Meng Sun. Multi-uav automatic dynamic obstacle avoidance with experience-shared a2c. In *2019 International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 330–335, December 2019.

[17] Lei Liu, Jie Feng, Qingqi Pei, Chen Chen, Yang Ming, Bodong Shang, and Mianxiong Dong. Blockchain-enabled secure data sharing scheme in mobile-edge computing: An asynchronous advantage actor–critic learning approach. *IEEE Internet of Things Journal*, 8(4):2342–2353, December 2020.

[18] Ryan Lowe, Igor Mordatch, Pieter Abbeel, Yi Wu, Aviv Tamar, and Jean Harb . Learning to cooperate, compete, and communicate, December 2015.

[19] M. McManus, Z. Guan, N. Mastronarde, and S. Zou. On the Source-to-Target Gap of Robust Double Deep Q-Learning in Digital Twin-Enabled Wireless Networks. In *Proc. of SPIE Conference Big Data IV: Learning, Analytics, and Applications*, Orlando, Florida, April 2022.

[20] J. Hu, M. McManus, S. K. Moorthy, Y. Cui, Z. Guan, N. Mastronarde, E. S. Bentley, and M. Medley. NeXT: A Software-Defined Testbed with Integrated Optimization, Simulation and Experimentation. In *Proc. of IEEE Future Networks World Forum (FNWF): WS5: Federated Testbed as a Service for Future Networks: Challenges the State of the Art*, Montreal, Canada, October 2022.

[21] Tianyi Chen, Kaiqing Zhang, Georgios B. Giannakis, and Tamer Başar. Communication-efficient policy gradient methods for distributed reinforcement learning. *IEEE Transactions on Control of Network Systems*, 9(2):917–929, April 2021.

[22] Mikko Lauri, David Hsu, and Joni Pajarinen. Partially observable markov decision processes in robotics: A survey. *IEEE Transactions on Robotics*, pages 1–20, September 2022.