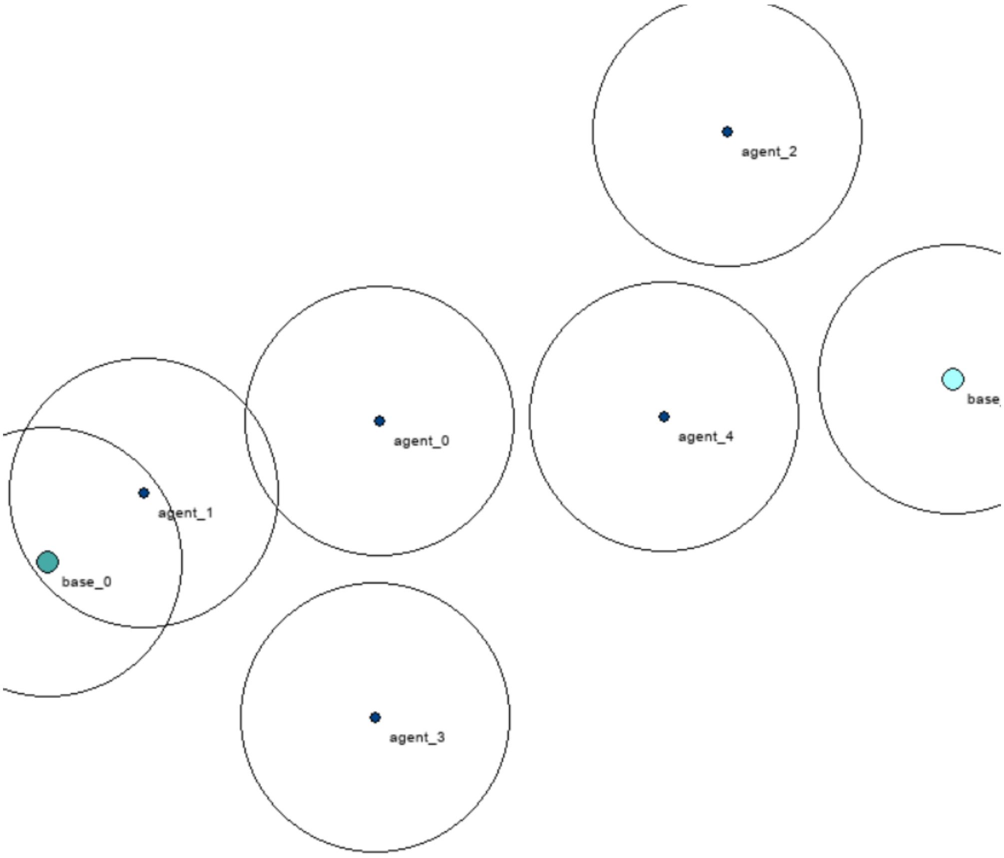
CHALMERS
UNIVERSITY OF TECHNOLOGY



Communication Relaying Networks Using Autonomous Drones

Solving Cooperative Markov Games Using Multi-Agent Reinforcement Learning

Master's thesis in Physics

Jacob Ljungberg

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

Communication Relaying Networks Using Autonomous Drones

Solving Cooperative Markov Games Using Multi-Agent Reinforcement Learning

JACOB LJUNGBERG

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology
Gothenburg, Sweden 2025

Communication Relaying Networks Using Autonomous Drones
Solving Cooperative Markov Games Using Multi-Agent Reinforcement Learning
JACOB LJUNGBERG

Supervisor: Mika Persson, Department of Mathematical Sciences/SAAB AB
Examiner: Axel Ringh, Department of Mathematical Sciences

Master's Thesis 2025
Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

Communication Relaying Networks Using Autonomous Drones
Solving Cooperative Markov Games Using Multi-Agent Reinforcement Learning
JACOB LJUNGBERG
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

Training several autonomous actors to solve cooperative tasks allows for systems with decentralized control and without single points of failure. One example is the use of swarms of drones to build quick and adaptable communication networks, allowing for efficient communication in hard-to-reach areas or dynamic settings. To ensure that these drone swarms operate effectively, robust training methods like reinforcement learning are essential. The problem can be modeled as a Markov game, allowing for multi-agent reinforcement learning (MARL). Multi-agent proximal policy optimization (MAPPO) was used as the training algorithm as it can effectively utilize shared data between all actors during training, and still allow for decentralized decision making during execution. It was found that building these communication relaying networks was possible for a small number of drones in fully observable environments. However, this ability decreased as the number of agents or the complexity of the environment increased.

Keywords: MARL, Reinforcement Learning, Markov Games, Centralized Training Decentralized Execution, MAPPO, MPD

Acknowledgements

I would like to express my sincere gratitude to my supervisor Mika Persson for excellent guidance and ideas throughout the project. The thesis would not have been possible without his input, dedication, and especially patience. I would also like to thank my examiner Axel Ringh for excellent feedback and constructive criticism throughout the project. Furthermore, I want to extend my gratitude to Adam Andersson for creating the project and contributing essential ideas and perspectives. Finally, I would like to express my deepest appreciation to Saab for this fantastic opportunity to explore such a fascinating subject.

Jacob Ljungberg, Gothenburg, June 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

CL	Curriculum Learning
DRL	Deep Reinforcement Learning
MADDPG	Multi-Agent Deep Deterministic Policy Gradient
MAPPO	Multi-Agent Proximal Policy Optimization
MARL	Multi-Agent Reinforcement Learning
MATD3	Multi-Agent Twin-Delayed Deep Deterministic Policy Gradient
MDP	Markov Decision Process
MG	Markov Game
MLP	Multi-Layer Perceptron
NN	Neural Network
PG	Policy Gradient
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
TD	Temporal Difference (Learning)
TRPO	Trust Region Policy Optimization

Contents

List of Figures

List of Tables

1
Introduction

In today's world, information and rapid, reliable communication are crucial, especially in dynamic and fast-changing environments, where resilience and adaptability are key. A fast and efficient communication network provides a significant advantage to the user which can deliver accurate status reports promptly. Moreover, in constantly changing environments, the communication network needs to be flexible and capable of quickly adapting to disruptions, including the replacement of damaged or missing links.

One promising solution to solving these problems is to deploy swarms of autonomous drones. The concept of drone swarms is not new and has proven useful in both civilian and military contexts, including disaster relief in hard-to-reach areas and conducting surveillance operations. Using autonomous systems has the advantage of decentralized control with no single point of failure. Losing any particular drone would not inhibit the swarm's ability to achieve its objective. Drones also have the advantage of easily navigating difficult terrain, in addition to a relatively low manufacturing cost. These traits make drone swarms appealing to various companies, including Saab, with which this study was performed in collaboration.

To enable drone swarms to function effectively in these complex environments, it is crucial to use robust training algorithms. As these problems can be modeled as Markov games, one approach that seems promising is reinforcement learning. In reinforcement learning, the agents learn by exploring and earning rewards and penalties based on their actions, allowing correct behavior to develop over time. Extending this concept to swarms of agents leads to multi-agent reinforcement learning (MARL), which allows the drones to optimize not only their individual actions but also how to cooperate and achieve collective goals. However, training multiple agents simultaneously poses certain problems, such as non-stationarity, where the actions of one drone could affect the learning process of another. Approaches such as multi-agent proximal policy optimization (MAPPO) can help address these challenges and have shown strong performance in control tasks [27].

1.1 Problem formulation

The research question can be formulated in the following way. Can one, and if so, how, use reinforcement learning to train a swarm of agents to autonomously build and maintain a communication network between several different nodes with the need to communicate with each other? In particular, these communication networks will be built in environments in which the agents have access to all information.

A part of the project involves developing a simulated environment with which the agents can interact. This is referred to as the information relaying environment and is the representation of the Markov game in the simulator. The experiences gained from this interaction are then used to train the agents to perform their tasks better. Their performance is evaluated by how quickly they can successfully relay messages between the transmitting base and the receiving base, and how well they manage to build relaying networks that could be used for multiple messages.

2

Theory

This chapter introduces the theoretical background and concepts necessary for this project. The chapter is split into four parts: Markov decision processes (MDP), reinforcement learning (RL), Markov games (MG), and multi-agent reinforcement learning (MARL).

A Markov decision process is the standard model to describe sequential decision making with probabilistic outcomes. It is a state-based model where all information of importance to the decision-maker at any given time is stored in the current state. The decision maker then chooses an action based on the state and the MDP transitions to the next state. MDPs are described in Section 2.1.

Reinforcement learning is a machine learning approach in which the agent learns to optimize its behavior by trial and error in an environment, where the agent's performance is measured by how big rewards it receives for choosing certain actions in certain states. It is one of the most common approaches to solving MDPs. In the RL section (Section 2.2), Q-learning, deep Q-learning, policy gradient algorithms, actor-critic, and proximal policy optimization, are presented.

Markov games are an extension of MDPs to multi-agent settings. This allows for the modeling of decentralized control and cooperation between several agents and is described in Section 2.3.

Finally, multi-agent reinforcement learning is a powerful approach to solving Markov games and is presented in Section 2.4. Here, the new challenges that arise in the multi-agent setting are presented, as well as the multi-agent version of proximal policy optimization.

2.1 Markov decision processes

The standard model used to describe sequential decision making with uncertain outcomes is the *Markov decision process* (MDP), also known as a *controlled Markov process* (CMP). MDPs are thus a good model for single-player games, where the decision maker chooses certain actions in certain states to reach a goal. The MDP evolves to a new state based on this action and internal transition rules. The decision-maker then receives a reward that indicates how good the chosen action was.

An MDP is defined as a tuple consisting of four elements (X, A, q, r), where:

- X is called the *state space* and each element $x \in X$ is called a *state*. The state space can be either discrete or continuous. The current state is always known to the agent.

- A is the *action space*. The action space may be either discrete or continuous. For each state $x \in X$ there is a subset $A(x)$ of A, whose elements are the actions the system can choose when in state x. The set of all admissible state-action pairs is denoted by

$$XA := \{(x,a) \mid x \in X \text{ and } a \in A(x)\}$$

- q, or $q(x' \mid x, a)$ is called the *transition probability function* or *transition kernel* and is a stochastic kernel on X given the set of all admissible state-action pairs. It can be understood as the probability of ending up in state x' given the action $a \in A(x)$ in state x. For a mathematically rigorous definition of (transition) kernels, see Appendix C in [7].
- $r : XA \to \mathbb{R}$, or $r(x,a)$, called the *one-step reward function* for taking action a in state x. The one-step reward function can also be represented as $XAX \to \mathbb{R}$ or $r(x,a,x')$, as shown in [1, p.39].

An MDP involves a decision maker, also called an agent, and represents a discrete-time controlled stochastic system. The initial state $x_0 \in X$ at time $t = 0$ is given. At time t, the current state and action taken are denoted by x_t and a_t. If the system is in state x_t at time t and the action $a_t \in A(x)$ is selected, then the system evolves into a new state x_{t+1} according to the probability distribution $q(\cdot|x_t, a_t)$ on X, and a reward $r(x_t, a_t, x_{t+1})$ is received. This process repeats itself until a terminal state is reached, or a maximum number of time steps has been completed, called *truncation*. Each run through an entire MDP is referred to as an *episode*. If the MDP is non-terminating, the process will be repeated for an infinite number of steps [7].

The goal of an MDP is to find a good *policy* for the decision-maker. A policy is a function π that determines the action a that will be taken when the system is in state x. In the general case, the policy is a probability distribution of the possible actions in a given state from which the chosen action is sampled, $a \sim \pi(a \mid x)$. A policy can be either stochastic or deterministic. An MDP, combined with such a policy, behaves like a Markov chain as the action chosen from state x is completely determined by $\pi(a \mid x)$, and thus independent of past states and actions.

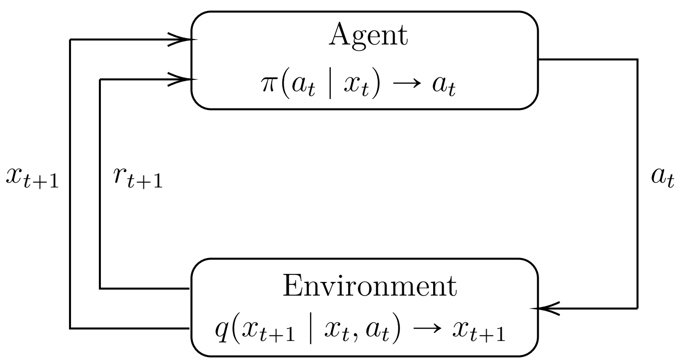


Figure 2.1: An illustration of the agent interacting with the environment in an MDP.

The aim of evaluating the performance of different policies is to find a policy that maximizes some cumulative function of rewards, called *performance criteria*.

In this thesis, the performance criteria used is the *expected total discounted reward*. The expected value of a state $x \in X$ under policy π is given by the *state-value function*

$$V_\pi(x) := \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t r(x_t, a_t, x_{t+1}) \mid x_0 = x\right], \tag{2.1}$$

where π is the policy used, and $r(x_t, a_t, x_{t+1})$ is the reward given at one time step. To guarantee that $V_\pi(x)$ is a finite function, the *discount factor* $\beta \in [0, 1)$, and the one-step reward function is assumed to be bounded. The parameter β can be seen as a scaling factor that controls how much more rewards are worth at the current time step compared to in the future. The function $V_\pi(x)$ represents the total future discounted reward expected by the agent when it is in state $x_0 = x$.

There is always at least one policy that is as good or better than all other policies, called the *optimal policy* π^* [7]. There might be several optimal policies, in that case, they share the same state-value function. The value functions are used to evaluate policies, and the value function connected to the optimal policy is called the *optimal state-value function*, and is defined as

$$V_{\pi^*}(x) := \max_{\pi'} V_{\pi'}(x) \text{ for all } x \in X. \tag{2.2}$$

The action-value function $Q_\pi(x, a)$, which gives the expected total discounted reward from taking action a in state x and then following policy π, is defined as

$$Q_\pi(x, a) := \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t r(x_t, a_t, x_{t+1}) \mid x_0 = x, a_0 = a\right] \tag{2.3}$$

The value function from (2.1) can be rewritten in the following way

$$V_\pi(x) = \sum_{a \in A} \pi(a \mid x) \sum_{x' \in X} q(x' \mid x, a) \left[r(x, a, x') + \beta V_\pi(x')\right] \tag{2.4}$$

where x' is the next state. This recursive equation is known as the *Bellman equation* and holds for all states and policies. From it, together with $V_\pi(x') = \max_{a' \in A} Q(x', a')$, the following relationship between V_π and Q_π can be derived

$$\begin{aligned} Q_\pi(x, a) &= \mathbb{E}\left[r(x, a, x') + \beta V_\pi(x) \mid x_0 = x, a_0 = a\right] && (2.5)\\ &= \sum_{x' \in X} q(x' \mid x, a) \left[r(x, a, x') + \beta \max_{a' \in A} Q(x', a')\right] && (2.6)\\ &= \sum_{x' \in X} q(x' \mid x, a) \left[r(x, a, x') + \beta V_\pi(x')\right], && (2.7) \end{aligned}$$

which is the Bellman equation for the action-value function The optimal policy π^* can be easily derived from the optimal action-value function Q^* by selecting the action that maximizes the expected return in each state

$$\pi^*(x) = \operatorname*{argmax}_{a \in A} Q_{\pi^*}(x, a). \tag{2.8}$$

Thus, the optimal policy can be derived from learning the optimal value function [7, 23, 1].

2.2 Reinforcement learning

Reinforcement learning (RL) is an interdisciplinary match between machine learning and optimal control theory. It focuses on which actions an agent in a dynamic environment should take to maximize some reward function. It could be defined as: **RL algorithms learn solutions for sequential decision processes via repeated interaction with an environment.** In RL, an agent makes decisions over multiple time steps within an environment. This is called a *sequential decision process* and a common example of this is a Markov decision process, which is the decision process most commonly used in RL. In each time step, the agent receives the environment state as input and chooses an action based on its policy. Given a certain action at a particular state, the environment may change its state according to some transition kernel and send a reward signal to the agent. A solution to the decision process is called an optimal policy for the agent. The agent uses a *learning objective* to find the solution. Usually, the learning objective is to maximize the agent's expected future reward in each state, which (2.1) shows [1].

RL is used to find solutions to problems described by an underlying MDP, where the transition kernel q and reward function r are unknown to the agent. Furthermore, the MDPs can be large which makes exact methods impractical, thus methods like RL are needed to find approximate solutions. The solution is an optimal policy, or at least as close to an optimal policy as possible. RL algorithms solve MDPs by *exploration* and *exploitation*. Exploration means that the agent tries new actions in different states to potentially discover new choices that lead to higher future rewards. Exploitation means that the agent chooses the best actions according to its policy to maximize its reward. It is important to find a balance between exploration and exploitation to maximize the cumulative rewards. Too little exploration and the global maximum might not be found, but too much exploration and valuable time and computational resources are wasted on suboptimal actions. The trade-off is called the *exploration-exploitation dilemma*, which is a fundamental concept in decision-making that arises in many fields.

RL differs from supervised learning as it does not need labeled input-output pairs. This is because the reward signals do not tell the agents what to do, a low immediate reward might be followed by a large pay-off further down. It also differs from unsupervised learning as the reward signal still contains information the agent uses to learn the optimal policy.

Two important choices have to be made when applying an RL algorithm to a problem. First, decide if the method is *model-based* or *model-free*. Model-based methods learn a model of the MDP, where the agent learns the transition kernel and the reward function of the MDP and uses it to plan optimal future actions. Model-free algorithms do not learn the underlying dynamics of the environment, but instead learn value functions or policies directly by trial-and-error interactions with the environment, which will be explained in more detail further down. This study only covers model-free approaches to RL.

The second choice to make when using an RL method is whether it is *on-policy* or *off-policy*. On-policy algorithms learn from actions taken during the current policy, while off-policy algorithms learn from data generated by other policies than the one currently being improved. In this context of RL, data refers to the agent's collected experiences (the state, action, new state, and reward). This usually means that a separate exploration policy or older versions of the current policy are used during training, with a separate target policy that learns. The off-policy algorithms are more sample efficient as they can reuse data over several policy updates, which on-policy methods can not do due to the need to sample new data after each policy improvement. However, on-policy algorithms are usually more stable as the collected data accurately reflects the policy being updated [23].

2.2.1 Q-learning

Q-learning is a tabular algorithm that learns action-value functions. The tabular part means that it uses look-up tables where each possible state-action pair (x, a) has its own entry, which corresponds to the action-value of (x, a), called the Q-value. The state and action spaces need to be finite due to the look-up table approach. Q-learning uses the Bellman optimality equation, Equation (2.5), to construct an update rule that iteratively updates the Q-table based on the agent's new experiences and the old Q-table. An agent's experience contains the current state x_t, the chosen action a_t, the next state x_{t+1}, and the received reward $r(x_t, a_t, x_{t+1})$.

The Q-learning update rule is defined as

$$Q(x_t, a_t) \leftarrow Q(x_t, a_t) + \alpha \left[r(x_t, a_t, x_{t+1}) + \beta \max_{a' \in A} Q_\pi(x_{t+1}, a') - Q(x_t, a_t) \right], \quad (2.9)$$

where α is the learning rate [26]. Using older values of the same action-value function to update itself in this way is referred to as *bootstrapping*. It is important to note that $\max_{a' \in A} Q_\pi(x_{t+1}, a')$ is an estimated return and not guaranteed to be the true value of state x_{t+1}, and as such, algorithms that use bootstrapping introduce bias in the learning procedure.

Q-learning is guaranteed to converge to the optimal policy π^* under the condition that all state-action pairs are tried infinitely many times and the learning rate α must be reduced over time such that

$$\sum_{k=0}^{\infty} \alpha_k \to \infty, \qquad \sum_{k=0}^{\infty} \alpha_k^2 < \infty,$$

where α_k is the learning rate at the k:th update with (2.9). However, in practice it is common to use a constant α as the convergence rate otherwise could be too slow. Furthermore, Q-learning is guaranteed to converge using any policy to interact with the environment, not just one policy that continually converges to π^*, i.e, Q-learning is an off-policy algorithm. Additionally, Q-learning, like all RL algorithms that learn value functions, suffers from non-stationarity as the agent's policy is changing throughout training, changing the distribution of states the agent visits (which are used in training), called the *state visitation distribution*.

As Q-learning is a tabular method, it becomes infeasible when the state or action spaces become too large. It can not describe continuous state or action spaces, for example, as these would require an infinite table size. To solve this and similar problems, Q-learning can be combined with the function approximation capabilities of neural networks [1].

2.2.2 Deep Q-learning

As described in the previous section, tabular RL algorithms like Q-learning have limitations as they can not represent arbitrarily large state spaces, as it is highly unlikely for the agent to encounter the same state more than once, making generalization essential for efficient learning. This problem can be solved by using a neural networks (NN) to parameterize the value function instead of a look-up table. When substituting the table with a neural network in Q-learning, the result is deep Q-learning. These NNs are multi-layer feed-forward neural networks with the output vector of layer ℓ being given by

$$V^{[\ell]} = g^{[\ell]}(W^{[\ell]}V^{[\ell-1]} - b^{[\ell]}) \tag{2.10}$$

where $V^{[0]}$ is the input vector, $g^{[\ell]}$ is the activation vector of layer ℓ, $W^{[\ell]}$ is the weight matrix of layer ℓ, and $b^{[\ell]}$ is the bias vector of layer ℓ [11, Ch.5].

In deep Q-learning, the neural network Q receives a state x as input, and in one forward pass of the network, outputs a scalar value $Q(x, a; \phi)$ for each possible action a, where ϕ represents the network parameters. The scalar output value represents how good a particular action a is in the current state according to the current action-value function. However, as each possible action has its own output neuron, deep Q-learning is limited to discrete action spaces like tabular Q-learning. The loss function for deep Q-learning is the estimated squared error between the target value y_t and the output of Q at time step t:

$$\mathcal{L}(\phi) = (y_t - Q(x_t, a_t; \phi))^2 . \tag{2.11}$$

The target value is the expected discounted future reward added to the one-step reward at the current time step. The value of terminal states, which are states where the game ends and agents can not receive any future reward, must be zero. To ensure this, the target value is computed differently if the next state is terminal. The target values are calculated using the following rule

$$y_t = \begin{cases} r_t, & \text{if } x_{t+1} \text{ is terminal} \\ r_t + \beta \max\limits_{a' \in A} Q_\pi(x_{t+1}, a'; \phi), & \text{otherwise.} \end{cases} \tag{2.12}$$

The Q-network is optimized by changing its parameters θ until the loss function is minimized. However, deep Q-learning suffers from two big problems. These are *the moving target problem*, and overfitting to the most recent experiences due to strong correlation in successive samples used in updating Q.

The moving target problem describes the non-stationarity that is caused by the changing of the bootstrapped value estimates as the value function is trained, which affects the target estimates. The moving target problem exists in tabular RL as well, but is amplified in deep RL. This is because updating the value estimates of one state may change the estimates of all other states when using an NN with a gradient-based optimizer to approximate the value function. In turn, this causes the bootstrapped target estimates to vary more rapidly. The problem is most notable in algorithms that use off-policy learning, like Q-learning. The following steps are why this could cause issues: The bootstrapped target value is calculated using $\max\limits_{a' \in A} Q_\pi(x_{t+1}, a'; \phi)$

for an experience (x_t, a_t, r_t, x_{t+1}). The ensuing update of the NN parameters ϕ might change the value of the estimate for the state action pair (x_{t+1}, a'), that is, for some action a' available in the new state x_{t+1}. However, the policy might never take action a' from that particular state, meaning the value estimate will only be indirectly updated when ϕ is. This constitutes a risk of potential overestimation of the value estimate without any corrections. This could lead to divergence, which could spread to other value estimates, by diverging values in the bootstrapped targets.

One way to mitigate the moving target problem is to use a target network. This is an additional network with the same architecture as the one used to represent the value function. This network is not optimized with gradient descent, instead, the parameters are periodically updated by copying the parameters of the main network. This ensures that the bootstrapped target values are relatively close to the estimates of the main value function, but the training stability is improved as the target values are not updated as often as often as the value function network. The target network is used to compute the bootstrapped target values in (2.12) instead of the main network. This eliminates the target values from the main network and makes it more stable as it does not need to compute backward passes through both instances of the value function during backpropagation. The two instances are the current Q-value and the bootstrapped target value. It is important to mention that the moving target problem is a bigger issue in off-policy algorithms like Q-learning. On-policy algorithms only use data collected under the current policy for training, reducing instability.

The second issue of deep Q-learning is that consecutive experiences are highly correlated. Because the environment is described as an MDP, consecutive samples are correlated as the experience at timestep t is directly dependent on the experience at timestep $t-1$. Furthermore, the distribution the experiences is sampled from depends on the policy and will thus change during training. These two factors result in the sampled experiences during training not being independent and identically distributed, which is generally necessary for the machine learning algorithms that train neural networks.

To avoid using only sequential experiences in training, a so-called *experience replay buffer* $\mathcal{D} = (x_t, a_t, r_t, x_{t+1})_{t=1,\ldots,\mathcal{D}_{\text{size}}}$, where $\mathcal{D}_{\text{size}}$ is the size of the buffer, can be used to collect experiences. During the training of the value network, a batch of experiences is sampled from the replay buffer. The sampling is performed uniformly at random. This offers two additional benefits. First, it provides the ability to reuse the same experience multiple times, improving the sampling efficiency. Second, the computed gradient becomes more stable with lower variance when batches of experiences are used compared to only a single experience. The replay buffer is usually of finite size, and the oldest experiences are continually replaced with new experiences when the buffer is at maximum capacity. Experience replay buffers that continually collect new experiences are used to train off-policy RL algorithms, as the experiences in the buffer are generated by different policies at previous times. In on-policy algorithms, all experiences are generated by the current policy, used to update the network, and then the replay buffer is emptied, and collection starts under the new policy [1].

Combining a neural network to approximate the action-value function with the two solutions to the previously mentioned problems, using a target network and a replay buffer, we get deep Q-networks (DQN). This is one of the most influential and earliest deep RL algorithms. DQN substantially speeds up training and increases performance compared to only using a neural network to describe the action value function and not addressing the previously mentioned problems. Using only target networks would still leave the problem of correlated samples, and using only a replay buffer would make the optimization unstable due to the moving target problem. The DQN algorithm was first described in [14].

2.2.3 Policy gradient methods

Deep Q-learning is a value-based RL algorithm, which means that the agent learns a parameterized value function. However, it is not necessary to learn the value function to solve a problem. Certain algorithms focus on learning the policy instead. A family of RL algorithms that are distinguished by learning a representation of a policy as a parameterized function is called *policy gradient* (PG) algorithms. In these algorithms, the policy is not derived from the represented value function but is instead directly represented by a neural network. This has two major advantages.

The first advantage of using a neural network to parameterize a policy applies to environments with discrete action spaces. The neural network that represents the policy in PG algorithms outputs a probability distribution over the possible actions. This allows for arbitrary stochastic policies to be learned, which is important in complex environments like multi-agent settings. The same is not possible in value-based algorithms, which typically use ϵ-greedy for exploration, and otherwise follow a deterministic policy. Value-based approaches are thus more limited in their ability to represent arbitrary (stochastic) policies. They could struggle in environments where the optimal policy is not deterministic, like partially observable environments or multi-agent games.

The second advantage of parameterizing a policy using a neural network is that policies for environments with continuous action spaces can now be represented. Value-based algorithms can not represent continuous actions as their neural networks have one output for each possible action an agent can take in a certain state, corresponding to the action-value estimate of that action given the current state. Continuous action spaces would need an infinitely sized output layer, as there are an infinite number of choices for continuous actions, which is not possible. However, the parameterized policies of PG algorithms can represent these continuous action outputs. This is done by designing the output layers such that the numerical and bounded value of a continuous action is the output. It is possible since the output of the policy networks is the actions, not the corresponding value for each possible action.

The same NN architecture that is used in the value-based approach can be used to represent probabilistic policies in discrete action spaces. The network outputs a scalar output for each possible action available for the input state. These outputs are then transformed into a probability distribution using the *softmax function*:

$$\pi(a \mid x; \theta) = \frac{e^{\ell(x,a;\theta)}}{\sum\limits_{a' \in A} e^{\ell(x,a';\theta)}}, \tag{2.13}$$

where θ are the parameters of the NN representing the policy and $\ell(x, a; \theta)$ is the scalar output for a particular action a, representing the preference the policy has for selecting action a in state x. The agent's action at the current time step is then sampled from this probability distribution [1].

2.2.3.1 Policy gradient theorem

To use gradient-based optimization algorithms to train a parameterized policy, we need the representation of the policy to be differentiable. We also need a well-defined loss function that leads to a good policy. Intuitively, it would be good to use a loss function that has its minimum when the expected episodic rewards are maximized. However, there is a problem with this approach. Changing the policy not only changes the actions and, by extension, the returns, but also the distribution of states the agent will end up in. Furthermore, the distribution over the next states depends on the environment's transition kernel, which is generally unknown to the agent. Thus, the expression of the policy gradient may not include a derivative of the state distribution.

When learning the policy parameters, a gradient of a scalar performance measure $J(\theta)$ with respect to the policy network parameters θ is used. The PG methods aim to *maximize* performance, updating the parameters using gradient ascent according to

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t), \tag{2.14}$$

where $\nabla J(\theta_t)$ is the stochastic estimate of the gradient of $J(\theta_t)$, α is the learning rate, and θ_t is the policy network parameters at time step t. Typically, $J(\theta)$ is defined as

$$J(\theta) = V_{\pi_\theta}(x_0), \tag{2.15}$$

where $V_{\pi_\theta}(x_0)$ is the true value function for the parameterized policy π_θ with parameters θ, given an arbitrary starting state x_0. As mentioned previously, $V_\pi(x)$ is the expected future returns given the state x following policy π, see (2.1).

Using the *policy gradient theorem*, it is possible to derive an analytical expression for the gradient of $J(\theta)$ with respect to θ that does not include the derivative of the state distribution under the current policy. This solves the concern that was presented above. The expression can be written as

$$\nabla_\theta J(\theta) \propto \sum_{x \in X} \Pr(x \mid \pi) \sum_{a \in A} Q_\pi(x, a) \nabla_\theta \pi(a \mid x; \theta), \tag{2.16}$$

where $\Pr(x \mid \pi)$ represents the state-visitation distribution in the environment given π, and $Q^\pi(x, a)$ represents the value given action a in state s. Importantly, (2.16) does not depend on any unknown information in the environment like the transition kernel or the reward function and thus, policy gradient can effectively be used in RL where these attributes typically are unknown to the agent. For a complete proof of the policy gradient method, see [23, Sec.13.2].

The expression can further be rewritten as

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{x \sim \Pr(x|\pi),\, a \sim \pi(a|x;\theta)} \left[Q_\pi(x, a) \nabla_\theta \log \pi(a \mid x; \theta) \right]. \tag{2.17}$$

This expression clearly shows that the state distribution and action values are given under the current policy π. Furthermore, it is clear from this expression that using policy gradient methods limits (without modifications) the optimization to using on-policy data, data generated by the current policy and not by older ones [1].

2.2.4 Actor-critic

A common approach in RL is to use *actor-critic* algorithms. This combines policy gradient with the previous approach of learning the value function. However, these functions are learned separately, in two different neural networks. The actor network learns the policy while the critic network learns the state-action value function Q_π, which like the name implies, evaluates the actions the actor outputs.

Using the actor-critic approach has two main benefits. First, using the value estimates from the critic network, it is possible to estimate the episodic reward from the experience of a single time step. This permits more frequent updates, as single steps can be evaluated and not just entire episodes. This could improve training efficiency compared to other algorithms that only update at the end of each episode, when the entire episodic return is known. Second, the variance is reduced due to the bootstrapped reward estimates only depend on the current state, received reward, and the next state, not on the entire history of the episode. However, this comes at a cost of increased bias as the value function might not accurately represent the true underlying value function. This trade-off often improves training stability [1].

Actor-critic methods can replace the action-value function in the policy gradient expression with the *advantage* $A_\pi(x, a)$ [13]. Formally, the advantage is defined as

$$A_\pi(x, a) = Q_\pi(x, a) - V_\pi(x). \tag{2.18}$$

With the advantage, the gradient of the objective function becomes

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{x \sim \Pr(x|\pi),\, a \sim \pi(a|x;\theta)} \left[A_\pi(x, a) \nabla_\theta \log \pi(a \mid x; \theta) \right]. \tag{2.19}$$

The advantage depends on the state-value function, which is known from the critic network, and the action-value function, which has to be estimated. The current action-value function $Q(x_t, a_t)$ can be estimated using $V(x_{t+1})$ and the immediate rewards r_t. Recall that the value of any terminal state is zero, the advantage can be approximated according to

$$A_\pi(x_t, a_t) = Q_\pi(x_t, a_t) - V_\pi(x_t) = \begin{cases} r_t - V_\pi(x_t), & \text{if } x_{t+1} \text{ is terminal} \\ r_t + \beta V_\pi(x_{t+1}) - V_\pi(x_t), & \text{otherwise.} \end{cases} \tag{2.20}$$

The advantage measures how much better or worse taking a particular action a in state x is compared to the average expected return of that state under the current policy π. The advantage is positive if a leads to a higher expected return than following π and negative otherwise. A positive advantage leads to a policy network parameter update that increases the probability of repeating the action in the same state.

2.2.5 Trust region policy optimization

A problem that arises when updating the policy parameters using the gradient from the policy gradient theorem is that any single update step might lead to large changes of the policy. If the policy update is large, there is a significant risk of reducing the expected performance of the policy. This is a problem even when using small learning rates. The risk of these large changes caused by single optimization steps can be greatly reduced using *trust regions*.

Trust regions mark an area in the policy parameter space where the policy would not change significantly. Hence, we can *trust* that the new policy will not lead to large performance decreases. This is done in the *trust region policy optimization* (TRPO) policy gradient algorithm [19]. In TRPO, a *clipped surrogate function* is used, which approximates the performance measure objective function while constraining the policy update to the trust region. The objective function maximization problem in the TRPO case becomes

$$\begin{aligned} &\max_{\theta} \mathbb{E}_{x \sim \Pr(x \mid \pi_{\theta_{\text{old}}}),\, a \sim \pi_{\theta_{\text{old}}}} \left[\frac{\pi_\theta(a \mid x)}{\pi_{\theta_{\text{old}}}(a \mid x)} A_{\pi_{\theta_{\text{old}}}}(x, a) \right] \\ &\text{subject to } \mathbb{E}_{x \sim \Pr(x \mid \pi_{\theta_{\text{old}}})} \left[D_{\text{KL}} \left(\pi_{\theta_{\text{old}}}(\cdot \mid x) \,\|\, \pi_\theta(\cdot \mid x) \right) \right] \leqslant \delta, \end{aligned} \tag{2.21}$$

where $D_{\text{KL}}(P \,\|\, Q)$ is the Kullback-Leibler divergence and can be intuitively thought of as a measure of how much a probability distribution Q differs from another probability distribution P, and $\delta > 0$ is a parameter controlling how large differences are allowed. The ratio

$$\rho := \frac{\pi_\theta(a \mid x)}{\pi_{\theta_{\text{old}}}(a \mid x)} \tag{2.22}$$

is the *importance sampling weight* and measures how close the updated policy is to the old policy used for collection. It would be equal to 1 if $\theta = \theta_{\text{old}}$. The probability distributions $\pi_\theta(a \mid x)$ and $\pi_{\theta_{\text{old}}}(a \mid x)$ are the action distributions given the state x under the new and old policies. [9].

The theory used to derive TRPO suggests a penalty term instead of a trust region constraint. This would give the following objective function

$$\max_{\theta} \mathbb{E}_{x \sim \Pr(x \mid \pi_{\theta_{\text{old}}}),\, a \sim \pi_{\theta_{\text{old}}}} \left[\rho A_{\pi_{\theta_{\text{old}}}}(x, a) - \beta\, D_{\text{KL}} \left(\pi_{\theta_{\text{old}}}(\cdot \mid x) \,\|\, \pi_\theta(\cdot \mid x) \right) \right] \tag{2.23}$$

where $\beta \in \mathbb{R}$ is some coefficient. The reason why a hard constraint is used instead of a penalty is that it is challenging to choose a single β that performs well in varied problems. Although TRPO introduces advantages with the use of trust regions to avoid large policy changes, it also comes with drawbacks. TRPO uses second-order derivatives to enforce the trust region, which is computationally inefficient and expensive to calculate. The computational complexity grows with the size of the action space [19].

2.2.6 Proximal policy optimization

Proximal policy optimization (PPO) is an algorithm developed to solve TRPO's problem of needing to calculate second-order derivatives to enforce the trust region [18]. Instead of approximating D_{KL}, the objective function is changed to penalize policies that make the importance sampling weight ρ deviate too far from 1. This is done by clipping ρ, resulting in the following objective function

$$J^{\mathrm{CLIP}}(\theta) = \mathbb{E}_{x\sim\Pr(x|\pi_{\theta_{\mathrm{old}}}),\, a\sim\pi_{\theta_{\mathrm{old}}}}\Big[\min\left(\rho,\, \mathrm{clip}\left(\rho, 1-\epsilon, 1+\epsilon\right)\right) A_{\pi_{\theta_{\mathrm{old}}}}(x,a)\Big], \quad (2.24)$$

where

$$\mathrm{clip}(r, 1-\epsilon, 1+\epsilon) = \begin{cases} 1-\epsilon, & \text{if } r < 1-\epsilon \\ r, & \text{if } 1-\epsilon \leqslant r \leqslant 1+\epsilon \\ 1+\epsilon, & \text{if } r > 1+\epsilon \end{cases} \quad (2.25)$$

is the clipping function with parameter $\epsilon \in [0,1]$, which ensures there is no incentive to let the new policy diverge too much from the current one.

Using PPO, it is possible to use the same data to train the networks multiple times without worrying about the changes to the policy being too large. This is because all the data used in each training iteration comes from the same old policy, and the more the current policy is updated with this data, the further it might diverge from the old policy. When the difference is large enough for the importance sampling weight to be clipped, the gradient becomes zero, and no further learning is possible until a new batch of data is sampled using the new policy. Thus, PPO guarantees small policy changes in a computationally efficient way, which simplifies the learning process and ensures problem scalability.

PPO, being a stochastic policy, does not use ϵ-greedy or added action noise for exploration, as opposed to deterministic policies. Instead, the objective function can be augmented with an entropy term to prolong exploration. The entropy term

$$\mathcal{H}(\pi(\cdot \mid x;\theta)) = -\sum_{a\in A} \pi(a \mid x;\theta) \log \pi(\cdot \mid x;\theta) \quad (2.26)$$

is subtracted from the objective function. The entropy of a policy π is maximized if it selects all actions with equal probability and can be seen as a measure of a policy's uncertainty. Thus, using the entropy to regularize the objective function penalizes quick convergence to any, possibly suboptimal, policy and increases exploration [20].

2.2.7 General advantage estimation

As stated in the previous section, PPO uses the advantage $A_\pi(x,a) = Q_\pi(x,a) - V_\pi(x)$ to approximate how good a certain action is relative to the action the current policy would have selected. However, using the estimated values from $V_\pi(x)$ and $Q_\pi(x,a)$ introduces bias as explained in Section 2.2.2. To mitigate the bias, an approach called *general advantage estimation* (GAE) has been developed [17].

GAE introduces a bias-variance trade-off controlled by parameter $\lambda \in [0,1]$ and the advantage estimate for the experience collected at time t under GAE is

$$\hat{A}_t^{\mathrm{GAE}(\beta,\lambda)} = \sum_{\ell=0}^{\infty} (\beta\lambda)^\ell \delta_{t+\ell}^V, \quad (2.27)$$

where V is an approximated value function (e.g. by a neural network), β is the discount factor from the underlying MDP, and $\delta_t^V = r_t + \beta V(x_{t+1}) - V(x_t)$. Notice that δ_t^V is an estimate of $A_\pi(x_t, a_t)$, and is the true advantage function if $V = V_\pi$, the true value function under policy π. In practice, the true value function is too complex to describe completely and the parameterized value function is always an approximation.

Setting $\lambda = 0$ results in $\hat{A}^{\mathrm{GAE}(\beta,0)}(x_t, a_t) = r_t + \beta V(x_{t+1}) - V(x_t)$, which has low variance but a high level of bias. Setting $\lambda = 1$ gives $\hat{A}^{\mathrm{GAE}(\beta,1)}(x_t, a_t) = \sum_{\ell=0}^{\infty} \beta^\ell r_{r+\ell} - V(x_t)$, which has a high variance due to the sum of terms, but lower bias from the value function estimation as the sum contains the actual values of the one-step rewards given to the agent. With GAE, it is possible to write the policy gradient (2.19) as

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{x \sim \Pr(x|\pi),\, a \sim \pi(a|x;\theta)} \left[\nabla_\theta \log \pi(a \mid x; \theta) \hat{A}^{\mathrm{GAE}(\beta,\lambda)}(x, a) \right] \tag{2.28}$$

where the former expression is restored when $\lambda = 1$. The expectation in (2.28), over the x and a, is usually estimated by summing over all states and actions collected during one training iteration. Estimating the expectation in (2.28), combined with writing out $\hat{A}^{\mathrm{GAE}(\beta,\lambda)}$, allows the expression to be rewritten as

$$\nabla_\theta J(\theta) \approx \mathbb{E} \left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi(a_t \mid x_t; \theta) \sum_{\ell=0}^{\infty} (\beta\lambda)^\ell \delta_{t+\ell}^V \right]. \tag{2.29}$$

This enables the policy gradient to be calculated given a set of collected experiences, and then the network to be updated.

2.3 Markov games

A Markov game (MG) is a repeated game with probabilistic transitions and state-dependent interactions. Markov games generalize MDPs to multiple interacting agents, where the environment transition kernel and reward function depend on the combined actions of all agents. A Markov game is defined as a tuple consisting of five elements $(I, X, (A^i)_{i\in I}, q, (r^i)_{i\in I})$ where:

- $I = \{1, 2, ..., N_\mathrm{a}\}$ is a finite set of agents, with N_a being the number of agents in the game.
- X is a set of possible states.
- A^i is agent i's action space. For each state x there is a subset $A^i(x)$ of A, which constitutes that state's admissible actions. The set of all action spaces at state x, called the joint action space, is defined as $A(x) = \bigtimes_{i\in I} A^i(x)$.
- q is the transition kernel. The probability of ending up in state x' given a joint action $a = (a_1, \dots, a_{N_\mathrm{a}})$ in state x is $q(x' \mid x, a)$. It obeys

$$\sum_{x' \in X} q(x' \mid x, a) = 1, \quad \forall x \in X, a \in A(x). \tag{2.30}$$

- $r^i : XAX \to \mathbb{R}$ is agent i's one-step reward function.

The game starts at an initial state $x_0 \in X$. At each time step t, the environment exists in state x_t, which is announced to each agent $i \in I$. This announcement is done through an *observation* o_t^i. The observation is a representation of the current state of the environment from the agent's point of view. It contains all the information in the state, and the game is thus fully observable. A general Markov game is not fully observable, but all Markov games discussed in this thesis will be. Each agent then chooses an action $a_t^i \in A^i(x_t)$ based on its policy $\pi^i(a_t^i \mid o_t^i)$. The environment transitions to the new state $x_{t+1} \in X$ according to the transition kernel $q(x_{t+1} \mid x_t, a_t)$ and each agent receives a reward $r^i(x_t, a_t, x_{t+1})$ and the process repeats until termination or truncation [21].

The goal of each agent i is, like in the case of MDPs, to maximize the expected total discounted reward or *value function*

$$V^i_{\pi^i;\pi^{-i}}(x) = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t r^i(x_t, a_t, x_{t+1}) \mid x_0 = x\right] \tag{2.31}$$

where π^i is the policy of agent i and $\pi^{-i} = (\pi^1, ..., \pi^{i-1}, \pi^{i+1}, ..., \pi^{N_\mathrm{a}})$ is the set of all other agents' policies, and $\beta \in [0, 1)$ is the discount factor. In other words, the goal of policy π^i is to maximize the expected reward for agent i when all other agents obey π^{-i}. This is known as *best response*. It is harder to find the optimal joint policy in a multi-agent setting than in single-agent games, where the state space updates are only caused by the agent's own actions. A concept that becomes important when dealing with solutions of MGs is *Nash equilibrium*.

The joint policy $\hat{\pi} = (\hat{\pi}^1, ..., \hat{\pi}^{N_\mathrm{a}})$ is a Nash equilibrium of a Markov game if

$$V^i_{\hat{\pi}^i;\hat{\pi}^{-i}}(x) \geqslant V^i_{\pi^i;\hat{\pi}^{-i}}(x), \quad \forall i \in I, \pi^i : X \to A^i, x \in X. \tag{2.32}$$

In other words, the joint policy is a Nash equilibrium if no agents i can (unilaterally) change its policy and improve its expected future cumulative reward. This is equivalent to every agent's policy π^i being the best response to $\hat{\pi}^{-i}$. There might exist several different Nash equilibria to any given game. These equilibria are found using, for example, multi-agent reinforcement learning [1].

In the case of a cooperative game, the goal of the MG is to find a good joint policy for all agents $\pi = (\pi^1, ..., \pi^{N_\mathrm{a}})$, similar to the solution of an MDP. The combination of all agents' individual policies π is the *joint policy*. In an MG, the policies π^i only depend on the current state and not on any other information, like previous states or the current time. This type of policy is called a *stationary policy* [6].

It is also possible to construct the action-value function in the multi-agent setting:

$$Q^i_{\pi^i;\pi^{-i}}(x, a) = \sum_{x' \in X} q(x' \mid x, a)\left[r^i(x, a, x') + \beta V_{\pi^i;\pi^{-i}}(x')\right] \tag{2.33}$$

where $Q^i_{\pi^i;\pi^{-i}}(x, a)$ is the action value of agent i, following policy π^i, given all other agents' policies π^{-i} when the joint action a is chosen in state x. The action value function is important when the policy gradient theorem and PPO are generalized to the multi-agent setting in the following section.

2.4 Multi-agent deep reinforcement learning

Multi-agent reinforcement learning (MARL) extends RL to multi-agent settings, and the problems are modeled as Markov games. Just like in the case of RL, this chapter will focus on deep MARL, as the inability of tabular RL algorithms to generalize to unvisited states makes them highly impractical in continuous state spaces. This generalization comes from using neural networks as function approximators. The extension to the multi-agent setting is necessary as the single-agent case is unsuitable to model realistic environments with decentralized actors or complex behavior like cooperation between different decision-makers.

In addition to inheriting the drawbacks from RL, like the exploration-exploitation dilemma, unknown environment dynamics, and the moving target problem, MARL algorithms suffer from additional challenges due to the increased complexity in the multi-agent setting. First, non-stationarity is a major issue in MARL. This non-stationarity arises from multiple agents learning from interactions with each other and could lead to situations when one agent learns from other agents' policies, and the other agents then adapt to the new policy, resulting in a back-and-forth in policy updates. This makes it hard to guarantee convergence to the optimal policy and risks slowing down training.

Second, it might be harder for agents to learn from rewards, as it is hard to distinguish whose actions contributed to the received reward. This is because it is harder for the agents, who only observe the current and last states, chosen actions, and the reward, to correctly extract which actions contributed. This problem becomes more prevalent in games with global rewards, which is common in cooperative games, and in games with large time delays between important actions and rewards. Global rewards are rewards given to all agents, independently of which agents actually contributed to the reward. This problem is referred to as the *credit assignment problem*. Finally, MARL suffers from a scalability problem. Increasing the number of agents usually means that the action and state spaces grow exponentially. However, this is only a problem in some environments, particularly if the agents' positions are a part of the state space.

The simplest approach to using RL to learn agent policies in Markov games is to use either *central learning* or *independent learning*, both of which simplify the problem to a single-agent problem. Central learning means that a single central policy is trained from local observations of each agent and outputs an action for each agent in the environment. It is part of a framework called centralized training and centralized execution (CTCE). In independent learning, each agent learns its own policy using its local observations. However, they do not observe the other agents, and the other agents' actions are instead just assumed to be part of the environment's transition kernel. The training approach used in independent learning belongs to a framework called decentralized training and decentralized execution (DTDE).

Even if both of these approaches are used with some success, they both suffer from significant limitations. Central learning lacks realism in decentralized environments, and the joint action space grows exponentially in the number of agents. Independent learning has a high risk of non-stationarity due to the simultaneous policy updates of the other agents, which change the transition kernels of each agent's

MDP as each agent assumes the other agents' actions are a part of the environment. To address these issues, a third training paradigm, centralized training and decentralized execution (CTDE), has been developed [2].

CTDE is very common in deep MARL as it combines the benefits of centralized training with decentralized execution. During training, an algorithm could utilize the joint local information to train all agents' policies, resulting in faster and more robust training. However, each agent's policy only needs the agent's local observation to select the new action, facilitating decentralized decision-making under realistic conditions.

2.4.1 Environments with homogeneous agents

Homogeneous agents are identical copies of each other, and as such, interchanging their policies in any way would result in the same solution for a given task. This can be done in two different ways. The agents could either be *weakly homogeneous* or *strongly homogeneous*. An environment has weakly homogeneous agents if for any permutation of the individual policies, the joint policy $\pi = (\pi^1, ..., \pi^{N_\mathrm{a}})$ produces the same value function. An environment has strongly homogeneous agents if the optimal joint policy, $\pi^* = (\pi^1, ..., \pi^{N_\mathrm{a}})$ consists of a collection of identical polices $\pi^1 = \pi^2 = ... = \pi^{N_\mathrm{a}}$, meaning all agents have identical policies. There exist methods that exploit properties of environments with homogeneous agents and improve the sample efficiency and scalability.

The first method is called *parameter sharing* and means that the agents use the same set of parameters in the value and/or policy networks, specifically

$$\theta_{\text{shared}} = \theta^1 = \theta^2 = ... = \theta^{N_\mathrm{a}}, \tag{2.34}$$

$$\phi_{\text{shared}} = \phi^1 = \phi^2 = ... = \phi^{N_\mathrm{a}}. \tag{2.35}$$

Sharing parameters θ results in all agents having identical policies, which is beneficial in environments with strongly homogeneous agents for two reasons. First, the shared policy is trained using experiences generated by all agents, which improves the sample efficiency as a larger and more varied dataset is used. Second, the number of network parameters does not increase linearly with an increasing number of agents, but is constant. Fewer parameters mean faster training. Sharing value function parameters ϕ is beneficial if all agents share the same reward function for the same two reasons as sharing policy parameters.

It is important to identify whether homogeneous agents in an environment can be considered weakly or strongly homogeneous. If the agents are to be entirely interchangeable, which is important when building autonomous systems with no single points of failure, the best choice is to treat the agents as strongly homogeneous and use parameter sharing on both the actor and critic networks. In the remaining thesis, all mentions of homogeneous agents refer to strongly homogeneous agents.

2.4.2 Policy gradient methods in MARL

A class of algorithms that usually uses the CTDE paradigm is the policy gradient methods in MARL. When generalizing the policy gradient theorem (see Section

2.2.3.1) to a multi-agent setting, the aim is to have a centralized value function with decentralized policies. The gradient of the expected return for agent i can be written as

$$\nabla_{\theta^i} J(\theta^i) \propto \mathbb{E}_{x\sim \Pr(x|\pi_\theta),\, a^i \sim \pi(a^i|o^i;\theta^i)} \left[Q^i_\pi(x_{\text{info}}, a^1, ..., a^{N_\text{a}}) \nabla_{\theta^i} \log \pi^i(a^i \mid o^i; \theta^i) \right]. \quad (2.36)$$

Here each agent i has its own policy π^i with parameters θ^i that outputs an action a^i based on the agent's current observation o^i, $Q^i_\pi(x_{\text{info}}, a^1, ..., a^{N_\text{a}})$ is a centralized action-value function under the joint policy $\pi = (\pi^1, ..., \pi^{N_\text{a}})$ which takes the actions of all agents $a = (a^1, ..., a^{N_\text{a}})$, in addition to the environment state information x_{info}, as input, and outputs agent i's action-value. The state information x_{info} can either be the current state x, or the joint observation $o = (o^1, ..., o^{N_\text{a}})$. In the case of partially observable Markov games, it could also be the observation history of the agents, the observations from all previous time steps. The centralized critic Q is used to update each decentralized policy π^i [10].

As the CTDE paradigm is used, the policy gradient method used is actor-critic (see Section 2.2.4) with a centralized critic. The critic's loss function is, just like in the deep Q-learning case, the squared error between the target value and estimated action-value:

$$\mathcal{L}(\phi) = \mathbb{E}_{x\sim \Pr(x|\pi_\theta),\, a\sim\pi_\theta} \left[\sum_{i\in I} \left(y^i - Q^i_\pi(x_{\text{info}}, a^1, ..., a^{N_\text{a}}; \phi) \right)^2 \right], \quad (2.37)$$

where, $y^i = r^i(x, a, x') + \beta \max_{a'} Q_\pi(x', a'; \phi)$ and ϕ are the action value function network parameters [2].

A schematic of the interaction between the environments used for experience collection and the training of the actor and critic networks is shown in Figure 2.2. The policy networks take the observations from the environment as input and output the chosen action. The observations are used as inputs to the policy networks. The observations, rewards, and chosen actions are used as input to the critic, which outputs the Q-value.

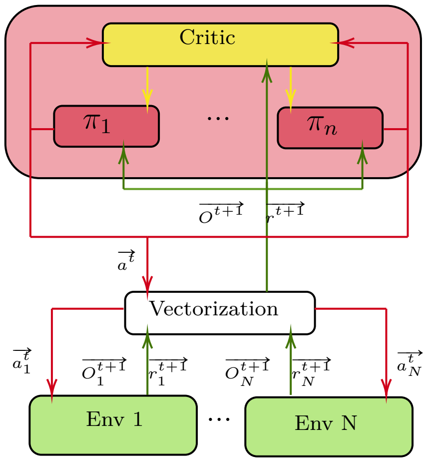


Figure 2.2: The figure shows an example of the CTDE approach with a shared critic network in MARL. The green lines represent observations and rewards from the environments, the red lines represent the action output from the policy networks, and the yellow lines represent the Q-value output from the centralized critic to update the policy networks.

2.4.2.1 Multi-agent PPO

PPO, being a policy gradient algorithm that has shown stable and sample-efficient behavior for being on-policy, has been extended to the multi-agent setting. There are two intuitive ways of extending PPO to environments with multiple agents. The first is based on the DTDE paradigm and is called individual PPO (IPPO), where all different agents have their own decoupled PPO training scheme. However, as discussed earlier, it is preferable to have an algorithm following CTDE as the centralized critic reduces the non-stationarity problems, which the second multi-agent implementation of PPO does, multi-agent PPO (MAPPO).

MAPPO can be implemented in two ways. Either using *individual policies* if the agents are heterogeneous, or using parameter sharing in the case where the agents are homogeneous. It is possible to have environments with different groups of agents, where each group consists of homogeneous agents. An example is team games. There could still be a central critic in these games, but only parameter sharing within each agent group [1, Sec.9.7.1].

The MAPPO objective function for agent i's policy network in multi-agent settings becomes

$$J^{\text{CLIP}}(\theta^i) = \mathbb{E}_{x\sim \Pr(x|\pi_{\theta_{\text{old}}}),\, a^i\sim\pi_{\theta^i_{\text{old}}}} \left[\min\left(\rho^i,\, \text{clip}\left(\rho^i, 1-\epsilon, 1+\epsilon\right)\right) A_{\pi_{\theta^i_{\text{old}}}}(x,a)\right], \tag{2.38}$$

where a is the joint action, and

$$\rho^i = \frac{\pi_{\theta^i}(a^i \mid o^i)}{\pi_{\theta^i_{\text{old}}}(a^i \mid o^i)},$$

is the importance sampling weight for agent i, measuring how close the action distribution of the updated policy is to the old policy. The parameters θ^i_{old} are the parameters of the old policy network for agent i. The parameter a^i is the action for agent i, and o^i is agent i's observation. In fully observable environments, the observation contains the same information as the current state, but possibly transformed relative to each agent. When parameter sharing is used, all agents share the same parameters and all agents' experiences are used to update the network [27, 2].

3 Methods and experimental setup

This chapter describes methods and the experimental setup. First, the simulator used for experience collection is described, and how it models the information relaying environment. Thereafter, how the training was performed, what parameters were used, and how the performance of the agents was evaluated are described. Finally, the different scenarios of the information relaying game tested are presented.

3.1 Simulating the information relaying environment

The simulator used for this project is based on PettingZoo's implementation of Multi-Particle-Environment (MPE) [24], which itself is derived from the original version developed by OpenAI [10]. MPE is a set of different games designed for testing different MARL algorithms in tasks that require cooperation or competition between agents. The simulator has been altered to fit our problem formulation, the largest modification being the communication model between the agents. This section presents the simulator's functionality relevant to the project and how the simulator relates to the Markov game representation. The term *environment* refers to an area where agents can choose actions and interact. The information relaying environment is the set of similar but different games tested in this project, with the goal of each being the successful relaying of one or more messages. It contains several *scenarios* in which game attributes like the number of messages or the agent's physical properties are different to see how that affects the information relaying problem. The scenarios are presented in detail in Section 3.4. The general structure of the simulator can be divided into *the backbone*, which contains the functions describing the physics of the environment and is the same for all scenarios tested, and the *scenario-specific functions*, which are the functions changed between the different scenarios.

The information-relaying environment is modeled as a Markov game, which consists of the tuple (I, X, A^i, q, r^i) as presented in Section 2.3. This is done by creating a world object containing the list of agents and their attributes (and also the list of non-acting entities in the environment). All agents and bases in the environment are collectively referred to as the entities of the environment. The agents, all of which are homogeneous, observe the environment and choose actions based on their policy and this observation. Having homogeneous agents in this context means that all agents share the same state space, action space, reward function, and transition kernel. The bases are scripted entities in the environment, which means they do not choose actions in the environment like agents do, but are instead a part of the environment, and their states are updated entirely based on the transition kernel. Furthermore, the simulation keeps track of the dimensions of the observation and action spaces, making sure that the dimensions always fit the input and output layers of the neural networks. The implementation of the transition kernel uses the actions of the agents to transform the environment's current state into a new state. Finally, the reward function returns a reward each time step based on the agents' actions and the state of the environment. A game instance is referred to as an *episode*. A visualization of the environment with two bases and two agents can be seen in Figure 3.1.

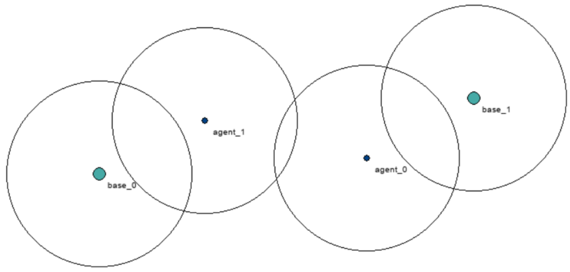


Figure 3.1: A visual representation of the simulated environment with two bases and two agents. The agents are not equipped with an antenna for directed communication. The larger circles show the transmission radius.

3.1.1 State space

The state space consists of the collection of the state spaces of all agents with the state spaces of all bases $X = (X_\mathrm{a}, X_\mathrm{b})$. The environment contains N_a agents and N_b bases, and thus is $X_\mathrm{a} = (X_\mathrm{a}^1, ..., X_\mathrm{a}^{N_\mathrm{a}})$ and $X_\mathrm{b} = (X_\mathrm{b}^1, ..., X_\mathrm{b}^{N_\mathrm{b}})$. Furthermore, the state spaces of individual agents and bases are divided into the physical state and the message buffer. The state space of agent i is represented as $X_\mathrm{a}^i = (X_{\mathrm{a,\,phy}}^i, X_{\mathrm{a,\,com}}^i)$, while the state space of base j is represented as $(X_{\mathrm{b,\,phy}}^i, X_{\mathrm{b,\,com}}^i)$.

A state $x \in X$ is the concatenation of all agents' and bases' states

$$
\begin{aligned}
&x = (x_\mathrm{a}^1, ..., x_\mathrm{a}^{N_\mathrm{a}}, x_\mathrm{b}^1, ..., x_\mathrm{b}^{N_\mathrm{b}}) = \\
&(x_{\mathrm{a,\,phy}}^1, x_{\mathrm{a,\,com}}^1, ..., x_{\mathrm{a,\,phy}}^{N_\mathrm{a}}, x_{\mathrm{a,\,com}}^{N_\mathrm{a}}, x_{\mathrm{b,\,phy}}^1, x_{\mathrm{b,\,com}}^1, ..., x_{\mathrm{b,\,phy}}^{N_\mathrm{b}}, x_{\mathrm{b,\,com}}^{N_\mathrm{b}}).
\end{aligned}
$$

Agent i's state consists of

$$
x_\mathrm{a}^i = \begin{bmatrix} x_{\mathrm{a,\,phy}}^i \\ x_{\mathrm{a,\,com}}^i \end{bmatrix} = \begin{bmatrix} p^i \\ \theta^i \\ b^i \end{bmatrix}, \tag{3.1}
$$

where $x_{\mathrm{a,\,phy}}^i = [p^i, \theta^i]^\top$, $x_{\mathrm{a,\,com}}^i = b^i$, p^i is the position in the two-dimensional environment, $\theta^i \in [0, 2\pi]$ is the orientation of the agent's antenna, and b^i is the agent's message buffer, which is explained below.

The state of base j consists of its position in the 2D-plane and its message buffer

$$x_{\mathrm{b}}^{j} = \begin{bmatrix} x_{\mathrm{b,\,phy}}^{j} \\ x_{\mathrm{b,\,com}}^{j} \end{bmatrix} = \begin{bmatrix} p_{\mathrm{b}}^{j} \\ b_{\mathrm{b}}^{j} \end{bmatrix}, \tag{3.2}$$

where $x_{\mathrm{b,\,phy}}^{j} = p_{\mathrm{b}}^{j}$ and $x_{\mathrm{b,\,com}}^{j} = b_{\mathrm{b}}^{j}$.

The difference between the agents' and bases' message buffers is that the agents have a fixed-sized buffer b^i while base message buffers b_{b}^{j} are infinite in size. Agent i's buffer is represented as an array b^i with a maximum capacity of L messages, $b^i = [b_1^i, ..., b_L^i]$. The message buffers are initially empty, and a received message occupies a buffer slot. An agent's decision to transmit a certain message does not automatically delete it from the buffer. One buffer slot ℓ in agent i's message buffer is represented as a tuple

$$b_{\ell}^{i} = \begin{cases} (1, \mathrm{ID}, j_{\mathrm{dest}}, k, h), & \text{if slot } \ell \text{ has a message,} \\ (0,0,0,0,0), & \text{if slot } \ell \text{ is empty.} \end{cases} \tag{3.3}$$

Here, the first element of the tuple is 1 if the slot has a message and 0 if it is empty. ID is the unique identifier of the message, j_{dest} is the destination of the message, $k \in \mathbb{N}$ is how many time steps the message has existed in the communication chain (since originally picked up from the transmitting base), and h is the relay history of the messages which is a list of the entities the message has previously passed through or originated from.

Various alternatives of the information relaying game, called scenarios, are tested. These are

- **One-way transmission:** One message and the agents transmit their messages isotopically to each other to relay between the two bases, called undirected communication.
- **One-way directed transmission:** One message and the agents use a directed antenna to relay the message between the two bases, called directed communication.
- **Two-way transmission:** Two messages and undirected communication between two bases.
- **Two-way directed transmission:** Two messages and directed communication between two bases.
- **Four messages:** The two-way communication scenario with four messages instead of two.
- **Three bases:** The directed one-way transmission scenario with an extra base added.

More details on these scenarios can be seen below in Section 3.4. In the scenarios with undirected communication, the agents do not use directed antennas but are instead assumed to transmit isotropically. The use of the directed antenna in a scenario is indicated by the boolean variable defined below,

$$\Theta^i \in \{0, 1\}, \quad \text{where } \Theta^i = 1 \iff \text{agent } i \text{ has a directed antenna.}$$

The size of the state space $\dim(X)$ is calculated according to

$$\dim(X) = 2N_{\mathrm{b}} + N_{\mathrm{a}}(2 + 4L + \Theta). \tag{3.4}$$

3.1.2 Observation space

In a Markov game, the entire state space is observed by the agent. However, we represent the state in a relative manner for each agent without losing any information. This representation to the agents is referred to as the agents' observations. This is distinct from the partially observable Markov game formulation since there is a one-to-one correspondence between what the agents observe and what the underlying state is. As the agents are homogeneous, their observation spaces are all the same size, equal to the size of the state space $\dim(X)$. The agents observe their environment according to their observation space and the representation of the current state of the environment at each time step.

Assuming a state $x \in X$, agent i observes the relative positions to all other agents, their antenna orientations, and their message buffers. Its observation is given by

$$o^i = \begin{bmatrix} p^i - p_\mathrm{b}^1 \\ \vdots \\ p^i - p_\mathrm{b}^{N_\mathrm{b}} \\ p^i \\ p^i - p^1 \\ \vdots \\ p^i - p^{N_\mathrm{a}} \\ \theta^1 \\ \vdots \\ \theta^{N_\mathrm{a}} \\ b^1 \\ \vdots \\ b^{N_\mathrm{a}} \end{bmatrix}, \tag{3.5}$$

where $p^i - p_\mathrm{b}^j$ is the relative position between agent i and base j, $p^i - p^k$ is the relative position between agent i and agent k, the agents observe the x and y components of the relative distances separately. θ^k is the antenna direction of agent k. The element θ^k is only observed when agent k uses directed communication, $\Theta^k = 1$, and b^k is the message buffer of agent k. An agent observes its message buffer and its absolute position. It should be noted that the entire message buffer is not observed by the agents. The information not observed is the relaying history of the message. However, this information is not used in the MG and therefore the game retains full observability.

Observing the relative position instead of all entities' absolute position was chosen for two reasons. First, it makes the games translation and orientation invariant as the agents have no information on where in the coordinate system they are. Second, it makes the games more realistic as the agents perceive the environment from their perspective. The bases do not observe the environment as they are scripted entities that do not choose their actions in the environment.

3.1.3 Action space

Like the state space, the action space of agent i consists of two parts, $A^i = (A^i_{\text{phy}}, A^i_{\text{com}})$, with the first part controlling the physical actions and the second part controlling the communication. The agents are homogeneous and, thus, share the same action space. This action space is discrete. Agent i's action during a particular time step consists of

$$a^i = \begin{bmatrix} a^i_{\text{phy}} \\ a^i_{\text{com}} \end{bmatrix} = \begin{bmatrix} v^i \\ \omega^i \\ c^i \end{bmatrix}, \tag{3.6}$$

where $a^i_{\text{phy}} = [v^i, \omega^i]^\top$ and $a^i_{\text{com}} = c^i$. $v^i \in \{-v_{\max}, 0, v_{\max}\}^2$ is the 2D velocity and $\omega^i \in \{-\omega_{\max}, 0, \omega_{\max}\}$ is the angular velocity of the antenna. Both of these velocities are discrete. The communication control action is an integer $c^i \in \{0, 1, ..., L\}$ where the chosen value results in

$$c^i = \begin{cases} 0, & \text{no transmission,} \\ \ell, & \text{transmitting message } b^i_\ell \text{ for } \ell \in \{1, ..., L\}. \end{cases} \tag{3.7}$$

This means that an agent can send a maximum of one of the messages in its buffer at each time step. In summary, each action consists of four sub-actions: velocity in x and y directions v_x and v_y, antenna controls ω, and communication control action c.

The actions are, as mentioned previously, the outputs from the policy networks. These networks output, through a softmax activation function, a list of probabilities from which one of the actions is sampled. The agents can move in both directions and transmit in the same time step. Thus, the output of the policy networks was chosen to be the Cartesian product of all sub-action combinations, representing all possible combinations of sub-actions. One value for each sub-action is chosen. Specifically, the set of all possible combinations of actions for agent i is defined as

$$A^i_{\text{phy}} \times A^i_{\text{com}} = \{(v^i_x, v^i_y, \omega^i, c^i) \mid [v^i_x, v^i_y, \omega^i] \in A^i_{\text{phy}} \text{ and } c^i \in A^i_{\text{com}}\},$$

where $v^i = [v^i_x, v^i_y]$. As an example, this results in a total of $3 \cdot 3 \cdot 1 \cdot 2 = 18$ possible combinations of actions in a scenario without antenna controls and only one message buffer slot, and $3 \cdot 3 \cdot 3 \cdot 4 = 108$ actions in a scenario with antenna controls and three message buffer slots. The number of possible combinations of actions is the size of the action space $\dim(A)$.

3.1.3.1 Handling inadmissible actions

In a general MG, not all actions are available to the agents in all states. One such state-action pair is when an agent transmits a message from an empty message buffer slot, in other words, sending a message that the agent does not have. There are two approaches to handling these *inadmissible actions*. The first is *action masking*, which means inhibiting the agents from performing inadmissible actions by making the probability of sampling these actions from the action probability distribution outputted from the policy networks zero. The second is to allow the agents to choose inadmissible actions, but penalize them for doing so.

Action masking is used to mask out certain actions in the full action space during action selection in policy gradient algorithms. This is done by replacing the logits, the output from the NN before the activation function, of the masked (inadmissible) action with a large negative number, and as such making it almost impossible for that action to be sampled, after the output has gone through the softmax function. This changes the gradients inside the network and could be a problem for the policy gradient theorem, which assumes that the policy is differentiable with respect to the network parameters. However, as the masking procedure only applies either the identity function or a constant, both being differentiable, the policy is still differentiable with respect to the parameters. Thus, action masking produces a valid policy gradient [8].

Instead of using action masking to inhibit the agents from performing inadmissible actions, it is possible to allow the choice of these actions but heavily penalize them. If an agent chooses to perform an inadmissible action, e.g., trying to send a message from an unoccupied spot in its message buffer, the environment ignores the communication sub-action and the agent is instead penalized. The idea is that the agents learn when a message can not be sent instead of not having the ability to do so in the case of action masking. In games with relatively small action spaces, like the information relaying game, little difference is anticipated between using action masking and inadmissible action penalties. If the action space were much larger, it would be expected that action masking would greatly outperform action penalties, as the agents have to explore a lot longer to identify which state-action pairs correspond to an inadmissible action.

3.1.4 Transition kernel

The transition kernel q maps an admissible state-action pair $(x, a) \in XA$ to a future state $x' \in X$. The transition kernel implemented in the simulator is stochastic, and the next state is chosen according to the probability distribution

$$X \ni x' \mapsto q(x'|x, a) \in [0, 1]. \tag{3.8}$$

The transition kernel, just like the action and state spaces, consists of a physical kernel q_{phy}, which describes the physical dynamics of the environment, and a communication kernel q_{com}, which governs all communication processes in the environment.

3.1.4.1 Physical kernel

The physical kernel takes the physical state and the physical actions as input and outputs the new physical state according to

$$X_{\text{phy}} \ni x'_{\text{phy}} \mapsto q_{\text{phy}}(x'_{\text{phy}}|x_{\text{phy}}, a_{\text{phy}}) \in [0, 1].$$

The physical kernel describes the dynamics of the environment. The underlying dynamics are modeled as a discrete-time stochastic difference equation described by the following difference equation

$$x_{\text{phy},t+1} = Ax_{\text{phy},t} + Ba_t + \Omega_t,$$

where

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} h & 0 & 0 \\ 0 & h & 0 \\ 0 & 0 & h \end{bmatrix},$$

with $h > 0$ being the time step size, and $x_{\mathrm{phy},t}$ one agent's physical state at time step t, a_t is one agent's chosen action at time step t

$$x_{\mathrm{phy},t} = \begin{bmatrix} p_{x,t} \\ p_{y,t} \\ \theta_t \end{bmatrix}, \quad a_t = \begin{bmatrix} v_{x,t} \\ v_{y,t} \\ \omega_t \end{bmatrix}, \tag{3.9}$$

where $p_{x,t}$ and $p_{y,t}$ are the x and y positions of an agent at time step t. $\Omega_t \leftarrow \mathcal{N}(0, \Sigma_t)$ is the noise term with

$$\Sigma_t = \begin{bmatrix} (v_{x,t}\sigma_x)^2 h & 0 & 0 \\ 0 & (v_{y,t}\sigma_y)^2 h & 0 \\ 0 & 0 & (\omega_t\sigma_\omega)^2 h \end{bmatrix},$$

where σ_x, σ_y and σ_ω are the scales of the noise. Because of symmetry, $\sigma_x = \sigma_y$.

In the case when an agent chooses its action in such a way that it exceeds the maximum speed in any direction, the velocity vector is normalized and then scaled to the maximum velocity

$$\begin{aligned} \bar{v}_x &= \frac{v_x \cdot v_{\max}}{\sqrt{v_x^2 + v_y^2}}, \\ \bar{v}_y &= \frac{v_y \cdot v_{\max}}{\sqrt{v_x^2 + v_y^2}}. \end{aligned}$$

This ensures that the agents can move at the same maximum speed in their eight possible directions, which is more realistic than having higher possible velocities in the diagonal directions than along the coordinate axes. This velocity normalization is similar to what is done in [28].

3.1.4.2 Communication kernel

The communication kernel describes how the message buffers of the agents and bases change based on the environment state and communication actions. It takes the entire state and the communication actions as input and outputs the new communication state according to

$$X_{\mathrm{com}} \ni x'_{\mathrm{com}} \mapsto q_{\mathrm{com}}(x'_{\mathrm{com}} | x, a_{\mathrm{com}}) \in [0, 1].$$

The environment contains two different types of transmissions. First, each agent i can choose to transmit a message through an action, $a^i_{\mathrm{com}} = c^i$. Second, the bases transmit messages based on a Bernoulli process with parameter p_{T}. The bases also have a second Bernoulli process with parameter p_{M}, determining when they will generate new unique messages. A message is successfully delivered to other entities if the signal strength between the transmitter and receiver is larger than a certain signal-to-noise ratio (SNR) threshold. The SNR depends on the distance and relative antenna orientation between the transmitter and receiver. Formally, the SNR between a transmitter with state x^t and a receiver with state x^r is defined as

$$\mathrm{SNR}(x_t, x_r) \propto \begin{cases} \frac{\cos(\widetilde{\theta}^t)\cos(\widetilde{\theta}^r)}{R^2}, & \text{if both } \widetilde{\theta}^t, \widetilde{\theta}^r \in [-\frac{\pi}{2}, \frac{\pi}{2}], \\ 0, & \text{if either } \widetilde{\theta}^t, \widetilde{\theta}^r \notin [-\frac{\pi}{2}, \frac{\pi}{2}]. \end{cases} \tag{3.10}$$

Here $\widetilde{\theta}^t$ is the angle between the central axis of the sender's antenna orientation and the line connecting the sender and receiver, $\widetilde{\theta}^r$ is the angle between the central axis of the receiver's antenna and the line connecting sender and receiver, and R is the distance between the transmitter and the receiver.

To simplify the environment, it is assumed that all entities can receive messages isotopically, $\widetilde{\theta}^r = 0$. Both the bases and agents (when not equipped with directed antennas) transmit isotropically as well. In scenarios where agents are equipped with antennas, they employ directed transmission instead and $\widetilde{\theta}^t \neq 0$.

The angle $\widetilde{\theta}^t$ is calculated as the difference between the transmitting agent's antenna direction θ and the relative angle between the transmitter and receiver. Using $\mathrm{atan2}(\sin(\widetilde{\theta}^t), \cos(\widetilde{\theta}^t))$, the angle is then wrapped to the interval $[-\pi, \pi]$ instead of $[0, 2\pi]$.

Each time step, all transmitting entities in the environment send their messages simultaneously. The bases check if they will update their message. Thereafter, all entities receive messages based on the SNR formula in (3.10), which depends on the current environment state. The entities only receive messages with SNR higher than a certain SNR threshold. The receiving entity ignores messages with lower SNR than the threshold. Agents have a finite number of buffer slots, and if they receive more messages than the number of unoccupied slots, the messages with the highest SNR are selected. The messages are placed in the free slots uniformly at random. Each agent's message buffer can only hold one message with the same ID. If a message that has the same ID as a message in the buffer is received, the received message replaces the message in the buffer if the received message has existed in the communication chain for a shorter time, that is, if its k is smaller. Finally, the bases place all messages they receive in their message buffers. The communication kernel is defined algorithmically in Algorithm 1.

3.1.5 Reward function

The agents are homogeneous and thus have the same reward function. The reward function provides the agents with a reward when a message is successfully delivered and because of it suffers from delayed feedback due to the large amount of time steps between the start of the game and the reward. Additionally, it also includes action penalties that penalize the agents for performing certain actions.

The reward function $r^i : XAX \to \mathbb{R}$ for agent i is defined as

$$r^i(x, a, x') = r_{\mathrm{g}}(x, x') - c_{\mathrm{a}}(a^i), \tag{3.11}$$

where $r_{\mathrm{g}} : XX \to \mathbb{R}$ is the global reward for a message arriving at the correct base,

Algorithm 1 The Communication Kernel

```
Initiate an empty set to contain all messages 𝓜
for each base j = 1 to N_b do
    Sample B_T^j ~ Ber(p_T), B_M^j ~ Ber(p_M)
    if B_T^j = 1 then
        Add the current message to 𝓜
    end if
    if B_M^j = 1 then
        Update message identifier ID^j
    end if
end for
for each agent i = 1 to N_a do
    if c^i > 0 then
        Add message b^i_{c^i} to 𝓜
    else
        No message transmitted
    end if
end for
for each agent i = 1 to N_a do
    Compile pre-buffer as an (SNR) ordered set (b^i_pre, ⩾) of recieved messages
    for each message m ∈ b^i_pre do
        if any b^i_ℓ, ℓ ∈ [1, ..., L] is a free slot then
            Update the relay history h and time k of m
            Add m uniformly random to a free slot in b^i
        end if
        if incoming message ID exists in buffer and has lower k then
            Replace the message in the buffer with m
        end if
    end for
end for
for each base j = 1 to N_b do
    Add all received messages to the base's message buffer b^j
end for
```

and $c_\mathrm{a} : XA \rightarrow \mathbb{R}$ is the action penalty. The global reward is defined as

$$r_\mathrm{g}(x, x') = \sum_{j=1}^{N_\mathrm{b}} \sum_{m \in b_\mathrm{rew}^j} R_\mathrm{g} - C_\mathrm{k} k(m). \tag{3.12}$$

Here, $R_\mathrm{g} > 0$ is the global reward for one message correctly delivered, and b_rew^j is the set of messages received by base j, whose IDs are not yet present in the base's message buffer. Remember that the set of received messages for base j, b_rec^j, only contains messages delivered to the correct base. The parameter $C_\mathrm{k} > 0$ is the cost per time step for not delivering a message, and $k(m)$ is the number of time steps message m has been in the relay chain. The global reward R_g depends on the number of unique messages in the environment N_mes, and C_k depends on the maximum number of time steps until the environment truncates and resets T_max, see the next Section 3.1.6. The parameters R_g and C_k have the following expressions,

$$R_\mathrm{g} = \frac{r_\mathrm{const}}{N_\mathrm{mes}}, \quad C_\mathrm{k} = \frac{r_\mathrm{const}/2}{T_\mathrm{max} N_\mathrm{mes}},$$

where r_const is the numerical scaling factor of the reward given for delivering messages. The maximum reward an agent can receive in one time step is R_g and this decreases to $r_\mathrm{const}/2$ if the agents receive all messages at the start of the episode and deliver them just before truncation. This is because the maximum value of $k(m)$ is T_max, as such

$$r_\mathrm{g} = N_\mathrm{mes}\left(\frac{r_\mathrm{const}}{N_\mathrm{mes}} - \frac{r_\mathrm{const}/2}{T_\mathrm{max} N_\mathrm{mes}} T_\mathrm{max}\right) = \frac{r_\mathrm{const}}{2}.$$

The action penalties c_a are defined as

$$c_\mathrm{a}(a^i) = (a_\mathrm{phy}^i)^\top C_\mathrm{phy} a_\mathrm{phy}^i + C_\mathrm{com} \mathbb{1}_{\{1,\ldots,L\}}(c^i), \tag{3.13}$$

where a^i the action of agent i as discussed above, $C_\mathrm{phy} = \mathrm{diag}([v_\mathrm{cost}, \omega_\mathrm{cost}])$ are the costs for physical controls and they are non-negative, $C_\mathrm{com} > 0$ is the cost for transmitting a message, and $\mathbb{1}_{1,\ldots,L}(c^i)$ is an indicator function that equals 1 if the agent transmitted a message this time step, and 0 otherwise.

One approach that could be used to mitigate the problems with training agents in environments with sparse rewards or high complexity is *reward shaping*. Reward shaping can be summarized as adding an extra help term to the reward function. The most common implementation of reward shaping is *potential-based*, which means that a potential-based shaping function $\beta\Phi(x') - \Phi(x)$ is added to the reward function, where $\Phi(x)$ is a potential function. It can be shown that the optimal policy remains unchanged when adding this term to the reward function. Another reward shaping implementation is to use a time-decaying term, an extra helping term that decays to zero as training progresses [16]. Another approach is curriculum learning, which is explained in detail in Section 3.2.3.

The action penalties are set such that if an agent moves throughout an entire episode and delivers the message at the end, the net cumulative episodic reward is still positive, but close to zero. This gives the following restriction $v_\mathrm{cost} < \frac{r_\mathrm{const}}{2T_\mathrm{max}}$. In practice, v_cost was set lower than this upper limit to account for value discounting.

This was done to ensure that it is always beneficial to deliver the message from all states. The penalty term v_{cost} is larger than ω_{cost} and C_{com}. This ensures that it is more beneficial to change antenna orientation and transmit rather than moving closer to the destination base if it already is in range.

In the case when inadmissible actions are allowed but punished instead of hindered using action masking, the action penalty is defined as

$$c_{\text{a}}(a^i) = \begin{cases} (a^i_{\text{phy}})^\top C_{\text{phy}} a^i_{\text{phy}} + C_{\text{com}} \mathbb{1}_{\{1,\dots,L\}}(c^i), & \text{if } a^i \text{ is admissible,} \\ C_{\text{ia}}, & \text{otherwise,} \end{cases} \tag{3.14}$$

where C_{ia} is the cost of inadmissible actions and is set to a large value to penalize agents for choosing actions that are physically impossible in certain states.

3.1.6 Termination and truncation

It is not reasonable to let the simulations run forever. Therefore, there needs to be instances where the simulation is aborted and resets. Two different outcomes would result in the end of an episode, either the agents solve the given task, which would result in a *termination*, or the agents fail to solve the task under the predefined time constraint, which would result in a *truncation*. These conditions vary between the different scenarios. All episodes are truncated after T_{max} time steps, with T_{max} being the same in all scenarios.

The termination condition varies slightly between the different scenarios. In scenarios with only one message, the game terminates once the message reaches its correct destination. Scenarios with more than one message terminate once all messages have been successfully delivered to their correct destination.

3.1.7 Functions of the simulator

The simulator mainly consists of the following functions:

- init - Initializes the environment. Creates the observation and action spaces and calls init_world.
- reset - Resets the environment after either a termination or truncation condition is fulfilled. Calls reset_world.
- step - Takes the current actions as input. Runs the transition kernel and outputs observations, rewards, terminations, and truncations. It is called inside the training loop and its outputs and inputs are the inputs and outputs of the training networks.
- action_space - Returns the shape of the action space. Used as the output shape of the neural networks.
- observation_space - Returns the shape of the observation space. Used as the input shape of the neural networks
- render - Renders the environment for visual evaluation
- init_world - Initializes a world object with agents and bases and sets their attributes in this environment.
- reset_world - Resets the world object. Sets the starting locations of the agents and bases.

- observe - Returns an observation for one agent.
- reward - Returns the reward for one agent.
- terminate - Checks if the termination condition is fulfilled and terminates.

3.2 Training and scenario setup

The simulator is integrated into *BenchMARL* and uses BenchMARL's implementation of the MAPPO algorithm [4]. MAPPO has been used as it is an algorithm with centralized training and decentralized execution, allows for discrete action outputs, and has performed well relative to other algorithms on MPE tasks [27]. BenchMARL is a MARL training library with several implemented algorithms and interfaces with many common RL environments, including PettingZoo, and in turn uses the *TorchRL* Python library [12]. BenchMARL has been modified to a certain extent to accommodate the information relaying environment simulator.

The focus of this project has been to examine how well MARL can be used to train agents to build communication networks autonomously and with decentralized control. Therefore, several different scenarios with increasing complexity have been tested. The aim is to discover what challenges emerge as the game, state space, and action space increase in complexity. The different scenarios are presented below in Section 3.4. Before the scenarios, the training process, overall structure, and the hyperparameters and network architecture used are presented.

Recall that MAPPO is a stochastic policy. That means the exploration comes inherent to the algorithm. Instead of using something like ϵ-greedy to pick one action at random some of the time, the actions are always sampled from a probability distribution. This means that even if the agents find themselves in the same state multiple times, they may choose different actions. The probability distribution reflects what actions the policy prefers and as such, it is less likely to sample non-beneficial actions.

3.2.1 The training loop

All scenarios were trained using the same training loop. Several environments were used in parallel to speed up training. The training loop started by initializing the environments used for the collection of experiences and the policy and value networks. There was only one of each network as the agents were homogeneous and parameter sharing was utilized, see Section 2.4.1. Then, the training were run for a certain number of training iterations depending on the complexity of the scenario.

One training iteration is described by the following steps: First, an empty experience buffer was initialized with batch size $\mathcal{D}_{\text{size}}$. Then, $\mathcal{D}_{\text{size}}$ *experiences* were collected from the environments by running the simulation. If one environment reached termination or truncation, it was reset, and the simulation was restarted. One experience consisted of one time step from one of the environments. It contained the current observation, the action taken, the reward received, the next observation, and termination and truncation conditions of each agent in the environment at that time step.

When the experience buffer was full, the collected experiences were split into N_{mb} minibatches and used to train the networks for N_{e} epochs per training iteration. The ability to train the networks multiple times with the same data is common with PPO as the clipping prevents too large changes. That is because the importance sampling weight (see (2.22)) is calculated based on the policy used during collection, and the policy can not change too far from the previous one, even if the number of parameter updates approaches infinity. Thus, each training iteration could consist of several smaller updates that together give the policy a better ability to converge to an optimum. The neural networks were trained using the Adam optimizer. After updating the network parameters, the training iteration was completed and the experience buffer was reset. The process was repeated until N_{iter} training iterations had been performed.

The policy used during data collection for training was stochastic, which means that the chosen action was sampled from the probability distribution outputted from the policy network. Because of this, it was not guaranteed that the agents would select actions leading to high rewards, and as such, deterministic evaluations were performed every ΔN_{eval} iteration. A deterministic policy in this context refers to always sampling the action with the highest probability, which is the greedy action. The evaluation was carried out over 100 episodes, and the average performance of the agents over these episodes was calculated. The training loop is presented as an algorithm in Algorithm 2.

Algorithm 2 The Training Loop

Initialize policy and value function parameters θ, ϕ, respectively
Initialize environments
for training iteration $j = 1, ..., N_{\text{iter}}$ **do**
  while step $< \mathcal{D}_{\text{size}}$ **do**
    Initialize empty data buffer $\mathcal{D}$
    Initialize environment state x_0 according to the inital conditions
    for timestep $t = 1, ..., T_{\max}$ **do**
      for each agent i **do**
        Sample action $a_t^i \sim \pi_\theta(a_t^i | o_t^i)$
      end for
      Execute $q_{\text{phy}}(x_{\text{phy},\,t+1} | x_{\text{phy},\,t}, a_{\text{phy},\,t})$ and $q_{\text{com}}(x_{\text{com},\,t+1} | x_t, a_{\text{com},\,t})$
      Observe r_{t+1}, o_{t+1}
      Store experience $(o_t, a_t, r_{t+1}, o_{t+1})$ in $\mathcal{D}$
      if x_{t+1} is terminal **then**
        break
      end if
    end for
  end while
  for epoch $e = 1, ..., N_{\text{e}}$ **do**
    Divide $\mathcal{D}$ into N_{mb} minibatches
    for minibatch $k = 1, ..., N_{\text{mb}}$ **do**
      Calculate $\hat{A}$ using GAE on all experiences in minibatch k
      Actor loss $L(\theta) = \mathbb{E}\left[-\min \left\{ \begin{array}{l} \dfrac{\pi_\theta(a \mid x)}{\pi_{\theta_{\text{old}}}(a \mid x)} \cdot \hat{A}, \\ \text{clip}\left(\dfrac{\pi_\theta(a \mid x)}{\pi_{\theta_{\text{old}}}(a \mid x)},\ 1-\epsilon,\ 1+\epsilon \right) \cdot \hat{A} \end{array} \right\} \right]$
      Critic loss $\mathcal{L}(\phi) = \mathbb{E}\left[\left(r(x, a, x') + \beta \max\limits_{a' \in A(x')} Q(x', a'; \phi) - Q_\pi(x, a; \phi) \right)^2 \right]$
      Adam update θ by minimizing the actor loss $L(\theta)$
      Adam update ϕ by minimizing the critic loss $\mathcal{L}(\phi)$
    end for
  end for
  if $\Delta N_{\text{eval}} \mod i = 0$ **then**
    Evaluate N_{eval} episodes with deterministic policy
  end if
end for

3.2.2 Initial conditions

The initial condition is the state the game is in after each reset of the environment. During the training of all scenarios, the bases' initial states were always given based on the same rule. The bases are initialized on the edge of a circle, which is randomly rotated. This ensures the bases have a specific distance between each other but with different orientations, making the agents learn a more general policy that is invariant under rotation. The ability to effectively control the distance between the bases was important, as the main goal was to see whether the agents were able to construct static communication networks. The bases were immovable and stayed in the initial positions throughout each episode. The distance between the bases was sampled uniformly in the range $[R_{\mathrm{b,min}}, R_{\mathrm{b,max}}]$.The minimum distance $R_{\mathrm{b,min}}$ differed between the different scenarios but was never small enough for the bases to communicate directly. The maximum distance between the bases $R_{\mathrm{b,max}}$ was set at the point where the agents could always build static communication networks, that is, stand still and successfully transmit a message between the bases, and is given by

$$R_{\mathrm{b,max}} = R_{\mathrm{com}} \cdot \frac{N_{\mathrm{a}} + 1}{2},$$

where R_{com} is the agents' communication radius.

Two different starting configurations for the agents were used during training. In the first approach, the agents were randomly placed within a circular region centered at the same point as the center of the bases. This circular region had a radius equal to or larger than the radius of the circle on which the bases themselves were positioned. This initial configuration was mainly used in scenarios with one to three agents. The second approach, referred to as the *pearl rope* starting position, was to assign each agent its own initial circular area, which it was randomly placed in. The centers of these circular disks were equally spaced out between the bases. This resulted in the agents more often starting the game in states close to where the task is easily solved. As training progressed, the radius of their circular starting areas increased in size, making the games progressively harder. When the sizes of these initial areas are increased throughout training to make the game increasingly harder for the agents, it is referred to as curriculum learning and is described in the next subsection.

3.2.3 Curriculum Learning

Curriculum learning (CL) refers to the process of iteratively increasing the difficulty of tasks as agents learn, resulting in the agents learning progressively harder tasks. This idea is inspired by how humans learn, by learning simple tasks and then building upon that knowledge to more easily learn new concepts. In deep RL, CL can improve sample efficiency by increasing the number of good states or decreasing the number of bad states explored by the agents. Furthermore, CL can increase the likelihood of convergence in games with sparse rewards, like the information relaying problem in this study, by making the early games simpler and thus increasing the frequency of rewards at the start of training. In summary, CL can be useful in RL problems where directly learning the full problem is too hard or slow [3, 22].

CL was implemented differently depending on which starting configuration was used. In the case where all agents started within the same circular disk, CL was implemented by iteratively increasing $R_{\mathrm{b,min}}$ from $R_{\mathrm{b,max}}/N_{\mathrm{a}}$ to $0.9R_{\mathrm{b,max}}$ throughout training. In the pearl rope starting configuration, CL was implemented by iteratively increasing the radius of each agent's starting disk, from 0 to $R_{\mathrm{b,max}}$ during training. This distance increase was controlled by the number of elapsed episodes, which means that the faster the agents solved the game, the faster the distance increased relative to the number of training iterations. This is because the faster the agents successfully solved the task, the shorter the episode was. Thus, the implementation of CL allowed for self-regulation by increasing the difficulty at a faster rate when the agents successfully solved the task quickly and vice versa.

In this study, CL has been used to speed up training and increase the chances of agents to learn good policies in all scenarios except the scenarios with one agent. All scenarios with more than one agent and the one-way transmission with two agents required CL to allow the agents to successfully learn the task. Figure 3.2 shows a comparison of the speed of convergence with and without curriculum learning in the one-way transmission scenario with two agents. This shows the increased speed of convergence when using CL and thus its importance in this project.

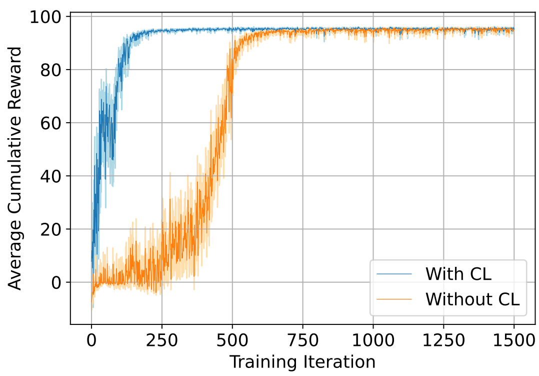


Figure 3.2: The average cumulative episodic reward as a function of training iteration in the one-way transmission scenario with two agents. The orange line shows the performance without any curriculum learning. The blue line shows the performance using CL, where the distance between the bases was increased linearly for the first 10000 episodes, or circa 100 iterations. The 95 % confidence interval is displayed in both cases.

3.2.4 Action masking

Two approaches were tested for handling inadmissible actions: invalid action masking and invalid action penalties. These were tested against each other on the one-way communication scenario using 2 and 4 agents with all other parameters identical. In the 2-agent case, both approaches showed comparable results, with action penalties recording a 3 % lower average cumulative reward than action masking. However, in the scenario with 4 agents, the agents failed to learn a good policy when trained using invalid action penalties, converging to a policy with an average of zero reward. This, as expected, shows that using action masking becomes increasingly important as the complexity of the environment or action space increases, as the exploration risks becoming too time-consuming. Thus, inadmissible action masking was used in all scenarios in this study when training with more than one agent.

3.2.5 Reward shaping

Potential-based and time-decaying reward shaping were both implemented and tested during training. These are two possible methods to help agents learn tasks with sparse rewards, like the scenarios in the information relaying environment. However, neither approach resulted in any learning improvement for the agents, and as such, both were excluded from the final implementation.

The potential-based reward shaping used was based on maximizing the communication capacity of the relaying network. This was done by adding a potential function that maximized the reward shaping reward r_{sh} when the agents were at a predefined optimal distance

$$r_{\text{sh}}(d) = \max\left(0, 1 - \left(\frac{d - d_{\text{opt}}}{d_{\text{opt}}}\right)^2\right) \tag{3.15}$$

from each other. The optimal distance was set to $d_{\text{opt}} = 0.9R_{\text{com}}$ to ensure the agents were far away from each other to maximize the range of the network, but not too far away that they risked moving out of communication range of each other too often. The reward then decreased quadratically as a function of distance from the optimal distance. This resulted in a behavior in which the agents learned the optimal distance and positioned themselves at this distance relative to the other agents. Intuitively, this would result in faster learning of the main objective, but no increased convergence rate or increase in final reward was recorded.

Time-decaying reward shaping was implemented by rewarding agents for delivering a message to another agent closer to that message's destination. This reward was scaled with the distance between the transmitting and receiving agent, with the maximum reward given if the distance was equal to R_{com}. This reward was removed once the agents had learned to transmit towards the correct base. Just like the implementation of potential-based reward shaping, this gave no increase in the final performance of the trained agents or the rate of convergence of the policy, and was therefore not used.

3.2.6 Preprocessing of the state space before NN

One of the hardest challenges for the agents' policies is to capture the complexities of the environments to be able to choose good actions in all states. An example of an approach that could reduce the complexity, as observed by the agents, is to use a coordinate transformation to represent the game state in a way that removes the need for the agents to learn the game in an arbitrarily rotated state. The approach is to find the midpoint between the bases, translate this point to the center of the coordinate system and then rotate the coordinate system until the bases lie on the x-axis. This transformed state is then fed to the neural networks and the output is given in this coordinate system. Then the coordinate system, as well as the chosen action, is transformed back to the original coordinate system. This has the potential to significantly improve training performance as the agents essentially only need to learn a smaller subset of games, games in which the bases are always located on the x-axis.

The coordinate system transformation as pre- and postprocessing will work in all environments with two bases. This is because it is always possible to include to object in the plane on the same circle. The center of this circle is translated to the absolute coordinate system's center and the circle is then rotated until the bases lie on the x-axis. It would also be possible to implement in games with more bases, as long as a certain symmetry is used. Either that bases are always located on some circle, or that, for example in the case of three bases, two of the bases are used as the basis of the coordinate transformation, while the third base is just an extra complexity the agents have to learn - as its position will be anywhere in the $x-y$ plane and not just on the x-axis.

This solution's drawback is that it requires full knowledge of the state, or at least the positions of the two bases, for the agents to perform the coordinate transformation. In partially observable scenarios, this could be solved by the agents searching for both bases while communicating the locations to each other. Once the positions are known, state preprocessing can be used.

3.2.7 Hyperparameters and NN architecture

In this section, the relevant parameters used in the information relaying environment and during training, as well as the neural network architecture, are presented.

3.2.7.1 Environment parameters

This section describes the values of parameters used in the simulator representing the information relying environment. The most influential parameters in connection to the information relaying environment were the maximum velocity v_{max}, the maximum angular velocity of the directed antenna ω_{max}, and the transmission radius R_{com}, as these parameters controlled how quickly the task could be solved by the agents. The values of the parameters are presented in Table 3.1.

The maximum angular velocity, see (3.9), was set to 0.25π/unit time, which means that the antenna could rotate 0.25π radians each time step. This gives the agents a few (eight) choices of antenna directions while still allowing them to steer the antenna to any of the discrete directions quickly. The velocity v_{max} was set low enough so that an agent could not quickly traverse R_{com} in one time step, to incentivize transmission to the next agent when possible instead of moving with the message, but also high enough so that the agents could always solve each task no matter the initial conditions of the game. This tradeoff, together with the truncation time of $T_{\text{max}} = 100$ resulted in the agents being able to move one transmission radius in 15 time steps or $R_{\text{com}} = 15 v_{\text{max}} h$

To add some context, using this maximum speed v_{max} with this time step size h, an agent could move from one base to the other in 38 time steps in a scenario with four agents. Given the maximum number of time steps before $T_{\text{max}} = 100$, the agents have more than enough time to maneuver and build a relay network regardless of starting positions.

The magnitudes of the control noises σ_x, σ_y, and σ_ω were set low enough to not greatly interfere with the agents' ability to learn their controls, and large enough to introduce some stochasticity in the agents' movements, which they have to learn to adjust to.

Table 3.1: Environment parameter values used in all simulations.

Parameter	Value
Maximum velocity ($v_{\max}$)	0.05 (unit length/unit time)
Maximum angular velocity ($\omega_{\max}$)	0.25π(/unit time)
Transmission radius (R_{com})	$15 v_{\max} h$
Velocity action noise (σ_x, σ_y)	0.1
Angular velocity action noise (σ_ω)	0.1
Reward for message delivery (r_{const})	100
Velocity action cost (v_{cost})	0.1
Angular velocity action cost (ω_{cost})	0.05
Communication action cost (C_{com})	0.01
Inadmissible action cost (C_{ia})	100
Prob for base transmission (p_{T})	0.5
Prob for base message change (p_{M})	0.1
Time step length (h)	1
Num time steps before truncation ($T_{\max}$)	100

3.2.7.2 Training hyperparamters

The learning rate was found using a grid search in the two-agent one-way transmission scenario. The evaluated learning rates were $[0.005, 0.001, 7.5 \cdot 10^{-4}, 5 \cdot 10^{-4}, 10^{-4}, 5 \cdot 10^{-5}]$. The best learning rate was 0.001 and was used throughout training in all scenarios with three or fewer agents. Even though a learning rate of 0.005 converged to a good policy faster than a learning rate of 0.001, it produced a 5 % lower reward on average after training. Thus, the chosen learning rate was 0.001. It is important to note that for the best possible results, the grid search should have been carried out separately for all scenarios and different number of agents, as they correspond to different Markov games. It was not performed due to computational cost and the found learning rate led to good results in the other scenarios with fewer than three agents.

The remaining parameters were taken from the original MAPPO article's MPE evaluations [27, Tab.7]. This included training all data in one batch ($N_{\text{mb}} = 1$) and training the networks with the same collected batch of data $N_{\text{e}} = 15$ times. However, some changes have been made as some parameter values were found to work better during training. The PPO clipping parameter ϵ was set to 0.2 as it was the best performing value in the original PPO article [18, Tab.1].

Interestingly, this combination of hyperparameters did not result in the agents successfully learning good policies when the number of agents increased to four. In the games with four agents, the following parameters were found to lead to convergence to a good policy: a learning rate of $\alpha = 0.00005$, number of minibatches $N_{\text{mb}} = 10$, and number of training epochs $N_{\text{e}} = 45$.

The entropy coefficient was kept at 0 during the simulations, as the agents were able to find the main objective well when using curriculum learning to guide them in increasingly difficult tasks. The numerical values of the training hyperparameters are shown in Table 3.2.

Table 3.2: Training parameter values used in all simulations. The values inside parentheses were used in the games with four or more agents.

Hyperparameter	Value
Clipping parameter (ϵ)	0.2
Entropy coefficient	0
GAE lambda (λ)	0.9
Discount factor (β)	0.99
Learning rate (α)	0.001 (0.00005)
Batch size ($\mathcal{D}_{\text{size}}$)	6000
Num minibatches (N_{mb})	1 (10)
Num training epochs (N_{e})	15 (45)
Optimizer	Adam
Adam epsilon	1e-6
Num parallel environments	10
Evaluation	
Evaluation interval (ΔN_{eval})	20
Num evaluation episodes (N_{eval})	100

3.2.7.3 Neural network architectures

As MAPPO was used as a training algorithm, which is an actor-critic algorithm, separate neural networks were needed to parameterize the policy and value function. This, combined with homogeneous agents allowing for a shared parameter approach for both the actor (policy) and critic (value function), resulted in only one policy and one value function network needed to be trained. Both these networks had the same architecture, with a simple multilayer perceptron (MLP) model with two fully connected hidden layers with 256 neurons each. The only difference is the size of the input and output layers, as explained below. The chosen activation function was Tanh. For more details on the MLP architecture and activation functions, see the PyTorch and TorchRL documentation [25, 12].

The policy networks take the local (each agent's own) observation as input, and have an input size of $\dim(X)$. A masked categorical distribution is used as the output distribution to account for inadmissible action masking, see Section 3.1.3.1. This distribution sets the output values of masked logits to zero and normalizes the unmasked values using a softmax function. This results in a probability distribution of possible actions, which is the output of the policy network, where the masked actions have a 0 % chance of being sampled. The action is then sampled from this masked probability distribution. The size of the output layer is $\dim(A)$. The critic network, as a centralized critic is used, takes the global observation and actions as input. This results in an input layer size of $N_{\text{a}}(\dim(X) + \dim(A))$. The output layer is constructed using a single neuron, as the critic network outputs the value estimate $Q(x, a)$, given the current state x as observed by the agents, and the joint action a.

The actor and critic networks had the following layout:

Policy networks
- Linear(dim(X),256)
- Tanh
- Linear(256,256)
- Tanh
- Linear(256,dim(A))
- Softmax

Value function network
- Linear($N_\mathrm{a}(\dim(X) + \dim(A))$, 256)
- Tanh
- Linear(256,256)
- Tanh
- Linear(256,1)

Here, Linear(A,B) represents a linear transformation on the data going from one layer with A neurons to the next layer with B neurons. Tanh and Softmax represent activation functions used after the linear transformations. For more information, see the *PyTorch* documentation [25].

To find which activation function to use, Tanh and ReLU activation functions were evaluated against each other using two and four agents in the one-way transmission scenario with all other parameters identical. Both choices resulted in the agents learning to solve the problem. However, Tanh resulted in faster convergence as well as about 3 % higher average reward at the end. This lines up with the findings of the original MAPPO article [27], where the conclusion was that Tanh activation function performed better in certain MPE environments than ReLU. As mentioned in Section 3.1, the information-relaying environment is based on the MPE environment. The Tanh was chosen as the activation function during training in all scenarios.

3.3 Evaluation metrics

The metric used to evaluate the performance of the agents is the average cumulative episodic reward (ACER). The cumulative part means that the undiscounted rewards given in all time steps in an episode are summed up for each agent. The average is then taken over all agents in an episode, and then over all episodes of each training iteration. Thus, there is one numeric value for each training iteration to easily compare and check how the performance improves throughout training. The 95 % confidence interval, as well as the minimum and maximum values of the cumulative episodic rewards, are also recorded to evaluate how much these values differ within each training iteration. The average cumulative episodic reward is recorded both during training with the stochastic policy and during the periodic evaluations with the deterministic policy. ACER is calculated as

$$\mathrm{ACER} = \frac{1}{N_\mathrm{a} \cdot N_\mathrm{ep}} \sum_{n \in N_e} \sum_{(x,a,r,x')} \sum_{i \in I} r^i(x, a, x'), \tag{3.16}$$

where N_ep is the number of complete episodes each training iteration, I is the set of agents, $r^i(x, a, x')$ is the reward function defined in (3.11), and the sum over (x, a, r, x') is the sum over all experiences (time steps) each episode.

Furthermore, each fully trained configuration of agents was evaluated over 1000 episodes. In these longer evaluations, two further evaluation metrics beyond the average cumulative episodic reward were calculated: The percentage of episodes in which the agents solved the given task, and the average number of time steps needed in the episodes where the task was solved. These two performance metrics allowed for comparison of how easy different scenarios were to solve for the agents, as the episodic reward is not necessarily comparable between different scenarios as it depends on the time it takes to deliver messages and the amount of movement needed, which both are influenced by the physical attributes of the environment, like distances between the bases, which differ between the different scenarios and amount of agents. Finally, as a qualitative evaluation, episodes with the fully trained agents were visually inspected to check for reasonable behavior.

3.4 Scenarios

In this section, the different scenarios that have been investigated are presented. Each scenario is a different version of the information relaying environment, with slightly different state spaces, action spaces, and transition kernels. The objective in all scenarios is to deliver N_{mes} messages to the correct destinations, with N_{mes} changing between different scenarios.

All scenarios were truncated and reset if not solved after 100 time steps. In the scenarios, when one to three agents were trained, the starting configuration where all agents started on the same disk was used. The radius of the disk was $2R_{\mathrm{b,max}}$ in the case of one agent, and $1.5R_{\mathrm{b,max}}$ in the case of two and three agents. The parameter $R_{\mathrm{b,max}}$ is the maximum distance allowed between the bases for static relaying chains to be possible. When four or more agents were trained, the pearl rope starting configuration was used, and the maximum radius of each agent's disk was $R_{\mathrm{b,max}}$. See Section 3.2.3 for a deeper explanation. The decreasing radius for higher number agent games was due to $R_{\mathrm{b,max}}$ increasing and the need to limit the number of time steps the agents needed to solve the task to within reasonable levels. Otherwise, they could spend the entire episode just moving in a straight line toward their relay location without reaching it before truncation.

3.4.1 One-way transmission

The one-way transmission scenario is characterized by the use of exactly two bases, with only one of the bases transmitting a message. The base transmitting the message is randomized each episode. The agents are rewarded for successfully delivering this message to the receiving base and the game terminates after one message has been successfully delivered. The agents are not equipped with a directed antenna, instead, they transmit and receive messages isotropically.

In this scenario, and in all scenarios with two bases, the distance between the bases is sampled uniformly at random from the interval $[R_{\mathrm{b,min}}, R_{\mathrm{b,max}}]$. The maximum allowed distance $R_{\mathrm{b,max}}$ is the distance that allows all agents to build a static relay network between the bases and maintain it without moving once built. The minimum distance $R_{\mathrm{b,min}}$ between the bases is defined differently for the different scenarios but never shorter than that at least one agent is needed to transfer the message. This ensures that the bases can not communicate directly with each other.

The bases are located on a boundary of a circle, which is randomly rotated at the start of each episode to make sure that the agents learn the games more generally, invariant of any rotation of the physical environment. The agents solve the Markov game by correctly positioning themselves between the bases and relaying the message. The solution yields higher total rewards when executed with minimal movement. The environment setup can be seen in Table 3.3.

Table 3.3: Environment setup for the one-way transmission scenario.

Parameter	Value
Number of bases (N_b)	2
Number of agents (N_a)	1, 2, 3, 4, 5
Information flow	One-way
Message buffer size (L)	1
Directed antenna used (Θ)	No
Number of unique messages (N_mes)	1
Termination	First message received
Action space dimension ($\dim(A)$)	$3 \cdot 3 \cdot 1 \cdot 2 = 18$
State space dimension ($\dim(X)$)	10, 16, 22, 28, 34

In the one-way transmission scenario with one agent, the training was carried out for 200 training iterations. The distance between the bases was randomized in the interval $[0.9R_\mathrm{b,max}, R_\mathrm{b,max}]$ each episode, but short enough for the agent to receive and transmit the message from one point in all training episodes. The agent always had its starting position randomly inside a circular area with the same center as the bases and with the radius $2R_\mathrm{b,max}$. The starting positions for the bases and agents are the same for the single-agent versions of all scenarios with two bases.

In the cases with two, three, and four agents, a total of 1500 training iterations were used, with curriculum learning increasing the difficulty under the first 10000 episodes, or about 100 training iterations. In the two and three-agent cases, the increased difficulty was the increasing distance between the two bases, with $R_\mathrm{b,min}$ going from $1/N_\mathrm{a} \cdot R_\mathrm{b,max}$ to $0.9R_\mathrm{b,max}$. The pearl rope configuration was used when training the game with four agents, the agents started the game in their own circular area. These circular areas were equally spaced out between the two bases and their radius increased from 0 to $R_\mathrm{b,max}$ throughout the CL part of the training. See Section 3.2.3 for more details on CL.

When five agents were trained in this scenario, 2000 training iterations were used. The five agents were trained twice. First, using the same initial configuration as the former games, with the bases randomly rotating relative to the bases' midpoint. Second, using the preprocessing of the state explained in Section 3.2.6. This was done to show the increased training performance when some of the complexity that the agents had to learn was removed.

3.4.2 Directed one-way transmission

The directed one-way transmission scenario is identical to the one-way transmission scenario described above, except that the agents are equipped with directed antennas for transmission. The agents can still receive messages isotropically. This increases the difficulty of the task, as on top of positioning themselves to relay the message, they also had to control the antenna to send the message in the correct direction. Like in the one-way transmission scenario, one base is sending a message that has the other base as its destination and which base is the transmitting base is randomized each episode. The environment setup can be seen in Table 3.4.

Table 3.4: Environment setup for the directed one-way transmission scenario.

Parameter	Value
Number of bases (N_{b})	2
Number of agents (N_{a})	1, 2, 3, 4
Information flow	One-way
Message buffer size (L)	1
Directed antenna used (Θ)	Yes
Number of unique messages (N_{mes})	1
Termination	First message received
Action space dimension ($\dim(A)$)	$3 \cdot 3 \cdot 3 \cdot 2 = 54$
State space dimension ($\dim(X)$)	11, 18, 25, 32

When one agent was trained, 500 training iterations were used and the distance between the bases was sampled uniformly at random on the interval $[0.9R_{\mathrm{b,max}}, R_{\mathrm{b,max}}]$. In the case of two agents, 1500 training iterations were performed and CL was used by linearly increasing the minimum distance between the bases from $1/N_{\mathrm{a}} \cdot R_{\mathrm{b,max}}$ to $0.9R_{\mathrm{b,max}}$ under the first 10000 environment episodes, which correspond to about the first 100 training iterations.

In the case of three agents, the CL approach was to increase the minimum distance between the bases in one large step after 1200 training iterations from $R_{\mathrm{b,max}}/3$ to $0.9R_{\mathrm{b,max}}$. This was done after the policy had converged in the simpler game, and then the training continued for another 1300 iterations. When four agents were trained, 2500 training iterations were run. The agents used the pearl rope starting configuration with linearly increasing difficulty during the first 25000 episodes, with their distances from the centers of the disks increasing from 0 to $R_{\mathrm{b,max}}$.

3.4.3 Two-way transmission

The two-way communication scenario is similar to one-way communication. The difference lies in the inclusion of one more message in the environment. In this scenario setup, both bases have their unique message, with the other base as the destination. Because of the increased number of messages, the agents' message buffers are also increased to a size of two to hold both messages simultaneously. This resulted in a larger action space as the agents have to decide which message to send. The agents receive half the global reward per successfully delivered message, and the game terminates once both messages have been delivered. The environment setup is presented in Table 3.5.

Table 3.5: Environment setup for the two-way transmission scenario.

Parameter	Value
Number of bases (N_b)	2
Number of agents (N_a)	1, 2, 3
Information flow	Two-way
Message buffer size (L)	2
Directed antenna used (Θ)	No
Number of unique messages (N_mes)	2
Termination	Both messages received
Action space dimension ($\dim(A)$)	$3 \cdot 3 \cdot 1 \cdot 3 = 27$
State space dimension ($\dim(X)$)	14, 24, 34

The single-agent case was trained with 500 training iterations. When two agents were trained, 2000 iterations were used and CL was implemented by linearly increasing the minimum distance between the bases from $R_{\mathrm{b,max}}/2$ to $0.9R_{\mathrm{b,max}}$ over the first 25000 episodes. Finally, in the two-way transmission scenario with three agents, the agents were trained with simpler games at the start, with the distance between the bases being randomized on the interval $[R_{\mathrm{b,max}}/3, R_{\mathrm{b,max}}]$ for the first 2000 training iterations. Then the minimum distance was increased to $0.9R_{\mathrm{b,max}}$ and training continued for another 1000 iterations.

3.4.4 Directed two-way transmission

The directed two-way transmission scenario is identical to the two-way transmission scenario described above, except that the agents are equipped with directed antennas for transmission. The agents can still receive messages isotropically. This increases the difficulty of the task, as on top of positioning themselves to relay the message, they also had to control the antenna to send the message in the correct direction. Compared to the directed one-way scenario, the agents have to direct the antennas towards both bases each game, substantially increasing the difficulty. Like in the two-way transmission scenario, both bases are sending a message that has the other base as its destination each episode. The environment setup can be seen in Table 3.6.

Table 3.6: Environment setup for the directed two-way transmission scenario.

Parameter	Value
Number of bases (N_b)	2
Number of agents (N_a)	1, 2
Information flow	Two-way
Message buffer size (L)	2
Directed antenna used (Θ)	Yes
Number of unique messages (N_mes)	2
Termination	Both messages received
Action space dimension ($\dim(A)$)	$3 \cdot 3 \cdot 3 \cdot 3 = 81$
State space dimension ($\dim(X)$)	15, 26

The directed two-way scenario was trained with one, two, and three agents. However, the last case with three agents was not successful in that the agents did not learn the task. In the case of one agent, training was carried out over 1500 training iterations. With two agents, 3000 training iterations were run with the difficulty slowly increasing over the first 100000 episodes, or about 1000 training iterations.

3.4.5 Four messages

This scenario is identical with the two-way scenario, except that each base transmits two distinct messages (different IDs) each episode, increasing the total number of messages from two to four. The bases only send a maximum of one message each time step and switch between the two messages randomly, which means that a good strategy is for the agents to position themselves within range of both bases to capture all messages. The number of available slots in the agents' message buffers is enlarged to four, which increases the observation and action space sizes and thus the difficulty in learning the game. Table 3.7 presents the environment setup.

Table 3.7: Environment setup for the four messages scenario.

Parameter	Value
Number of bases (N_b)	2
Number of agents (N_a)	1, 2
Information flow	Two-way
Message buffer size (L)	4
Directed antenna used (Θ)	No
Number of unique messages (N_mes)	4
Termination	All four messages received
Action space dimension ($\dim(A)$)	$3 \cdot 3 \cdot 1 \cdot 5 = 45$
State space dimension ($\dim(X)$)	22, 40

The four messages scenario was only trained for one and two agents, similar to directed two-way transmission. The single-agent case was trained for 2500 iterations. The scenario with two agents was trained for 2500 iterations, where the first 25000 episodes used curriculum learning.

3.4.6 Three bases

The three bases scenario is identical to the directed one-way transmission scenario, with three instead of two bases that can transmit the message. Each episode, the base that will transmit a message is randomized, and that message's destination is then randomized between the two remaining bases. This scenario evaluates the agents' ability to distinguish between different destinations, in addition to monitoring different possible origins for the message to quickly deliver to once it is transmitted from the sending base. The environment setup is presented in Table 3.8.

Table 3.8: Environment setup for the three bases with one message scenario.

Parameter	Value
Number of bases (N_b)	3
Number of agents (N_a)	1, 2, 3, 4
Information flow	One-way
Message buffer size (L)	1
Directed antenna used (Θ)	Yes
Number of unique messages (N_mes)	1
Termination	First message received
Action space dimension ($\dim(A)$)	$3 \cdot 3 \cdot 3 \cdot 2 = 54$
State space dimension ($\dim(X)$)	13, 20, 27, 34

The scenario with three bases is set up differently from the other scenarios. Here, the distance between all bases is the same. This is independent of the number of agents, except for the single-agent case, where the distances were small enough such that the agent could always transmit from one base to any of the other bases. The distance between the bases when training multiple agents was set such that two agents could relay a message between each base pair. The distance was the same in the two, three, and four agent cases. The use of this setup allowed for the comparison of agent performance when the number of agents increased while the size of the physical environment stayed the same. The case with one agent was trained for 500 iterations. The training was carried out over 2000 training iterations in the cases with multiple agents. The distance between the bases increased linearly during the first 10000 episodes, using curriculum learning.

4

Results and discussion

In this chapter, the results of the different scenarios are presented. The first part of the results includes Table 4.1, which summarizes the evaluation metrics of all tested scenarios. Then, the results are presented one scenario at a time and with an increasing number of agents. Information about the different scenarios can be found above in Section 3.4. The agents' performance is shown as a function of how long training has progressed for both the stochastic policy used for experience collection and the deterministic policy used for evaluations. In all figures, the average cumulative episodic reward of all agents during one environment episode, the 95 % confidence interval, and the range between the minimum and maximum value are presented.

4.1 The evaluation metrics

The computed evaluation metrics for all scenarios are presented in Table 4.1. These evaluation metrics are the average cumulative episodic reward with a 95 % confidence interval, the percentage of episodes in which the agents solved the task, and the average episode length of episodes in which the task was solved. In summary, the agents achieved higher average episode lengths for solved games and lower average rewards as the number of agents increased. This is expected as the distance between the bases was larger in games with more agents, which means that more time steps are needed to deliver the messages and thus lower scores on these performance metrics. However, the percentage of solved games also decreased with an increase in the number of agents, which shows that these games were harder to solve for the agents. It was also observed that the percentage of solved games decreased with increased complexity of the scenarios, like the use of a directed antenna for transmission or more messages. The increase in the number of messages significantly decreased agent performance across all metrics.

Table 4.1: Agent performance under an evaluation with 1000 episodes in the different scenarios. The evaluation metrics presented are the average cumulative episodic reward (ACER) with a 95 % confidence interval, the percentage of solved tasks, and the average episode length of the successfully solved tasks.

Scenario	# Agents	ACER	Solved (%)	Ep. Length
One-way	1	97.40 ± 0.065	100 %	13.49
	2	94.27 ± 0.75	98.5 %	17.13
	3	85.42 ± 1.35	94.7 %	30.48
	4	72.55 ± 2.07	85.4 %	42.13
	5	6.29 ± 1.77	15.8 %	62.72
Directed one-way	1	96.69 ± 0.35	99.7 %	14.24
	2	94.45 ± 0.42	99.6 %	18.52
	3	85.69 ± 0.88	97.9 %	35.20
	4	23.77 ± 2.46	37.6 %	63.25
Two-way	1	96.86 ± 0.34	99.6 %	16.14
	2	94.30 ± 0.55	98.8 %	25.04
	3	34.74 ± 2.07	25.0 %	67.42
Directed two-way	1	95.88 ± 0.093	100 %	21.92
	2	82.04 ± 1.34	87.9 %	44.72
Four messages	1	93.46 ± 0.33	98.0 %	38.11
	2	18.18 ± 1.18	0.70 %	75.57
Three bases	1	96.07 ± 0.60	99.1 %	14.87
	2	89.28 ± 0.56	99.3 %	35.98
	3	89.82 ± 0.96	97.8 %	27.20
	4	82.91 ± 1.74	91.9 %	31.90

This could be due to two reasons. First, the distance between the bases increased linearly with the number of agents, making the games slightly harder. However, the distance between the bases and the area in which the agents started was set in such a way that it would take approximately the same time for the agents to build the relaying network in all cases, if they learn a good policy. Therefore, the increased difficulty is probably due to the increased complexity in the state space caused by the larger distance, and especially more agents.

The second reason for decreased performance is the credit assignment problem mentioned in Section 2.4. The credit assignment problem refers to the problem of accurately identifying which actions lead to a particular outcome. It becomes increasingly harder in games with sparse rewards, like the scenarios explored in this study, as important actions might be tens of timesteps away from the reward. As the main reward given to the agents (for delivering messages) is global, all agents get the same reward. This means that the neural networks must learn to distinguish which of the agents' actions contributed to the reward and which did not. The problem grows in severity with the number of agents. Explaining the decreased performance as the number of agents increased. These factors contribute to the performance as a function of the number of agents in all scenarios presented below, even if not directly mentioned.

4.2 One-way transmission

Here, the results from the one-way transmission scenario are presented. The performance of the agents in the games with one to four agents is presented in Figure 4.1. CL has been used in all cases, except when only one agent was used. It is clear from Figure 4.1, the worst performing case in Figure 4.7, showing the one-way transmission scenario with five agents, and Table 4.1 that the performance of the agents decreased as the number of agents increased.

From the visual evaluation, it is clear that the performance of the agents decreased as the number of agents increased. In the two-agent case, it is clear that each agent moves to each of the bases and waits for a message. Once an agent receives a message, the two agents effectively collaborate by relaying the message to each other with minimum movement to complete the task. However, when three to five agents solve the task, their movement is more clumped up, with several agents running parallel to each other to transmit the message to the receiving base, which results in lower rewards as they move around more, and the message takes more time to reach its destination.

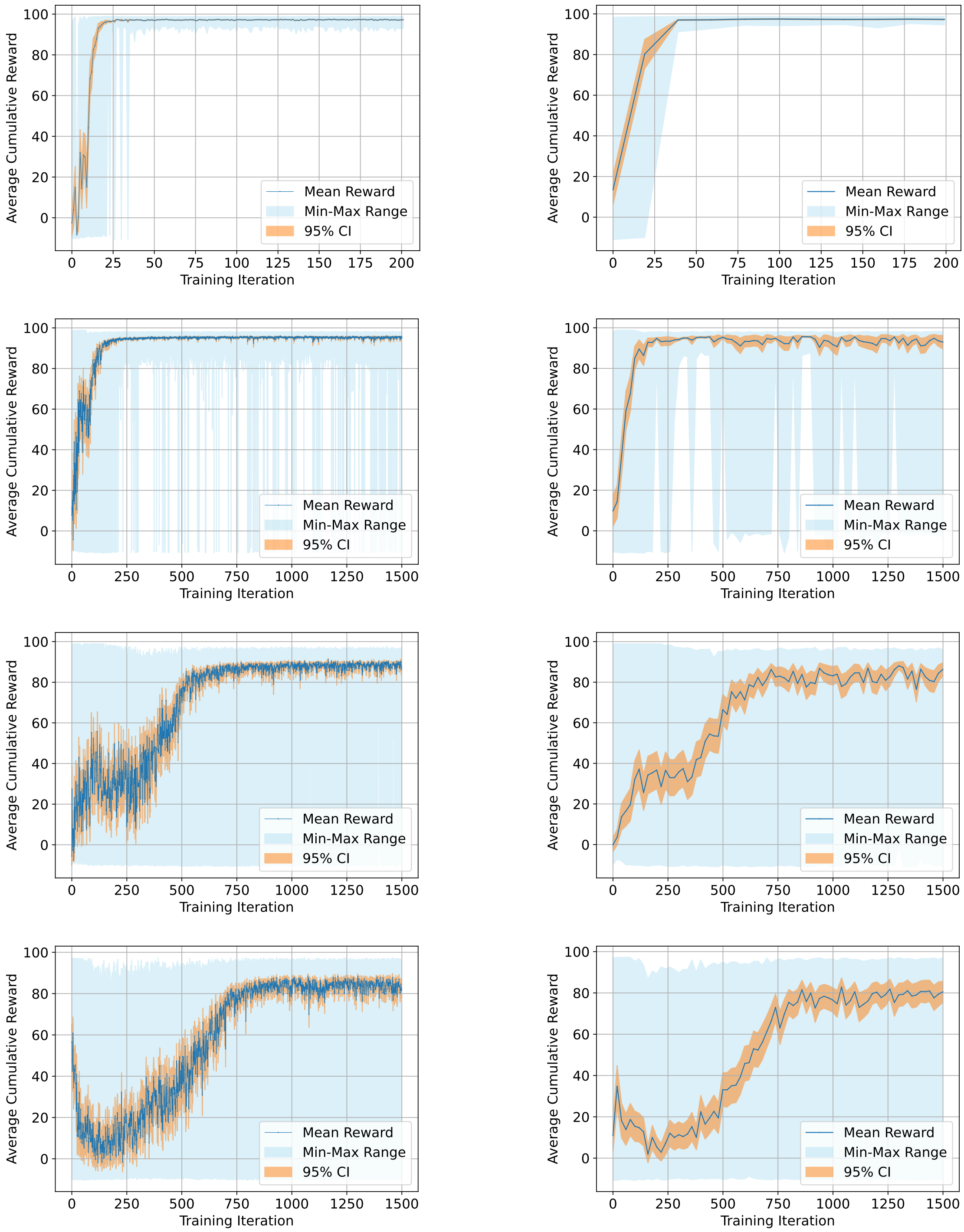


Figure 4.1: Training and evaluation results for one to four agents in the one-way transmission scenario. The blue line represents the average cumulative episodic rewards, the orange area is its 95 % confidence interval, and the light blue area indicates the range between the minimum and maximum values of all collected episodes. The left side shows the results for the stochastic policies that generate the data used in training. The right side shows the results when always sampling the action with the highest probability (deterministic policy). The number of agents increases for each row, with the top row showing the results for one agent and the bottom row for four agents.

4.3 One-way directed transmission

Comparing the results presented in Figure 4.2 and Table 4.1 with the results from the one-way transmission scenario, it is found that the evaluation metrics show only small differences when one, two, and three agents are evaluated. The scenario with directed communication even shows a slightly higher percentage of solved tasks for two and three agents. However, the performance greatly decreased for four agents. This decrease in performance is more significant than the corresponding decrease in one-way transmission. The reason for this is the credit assignment problem, which is exacerbated when the number of agents increases and by the added difficulty in that the agents not only need to be a certain distance away to transfer a message, but also need to orient the antenna the correct way.

Visually, the agents show good behavior and antenna control in the case of two agents. One episode of this game is visually shown in Appendix A. For games involving more than two agents, the agents, just like in the one-way transmission scenario, are moving less efficiently by moving as a group from one base to the other. A majority of the time, the agents orient their antennas toward the receiving base, which indicates they have learned the connection between antenna direction and the future reward.

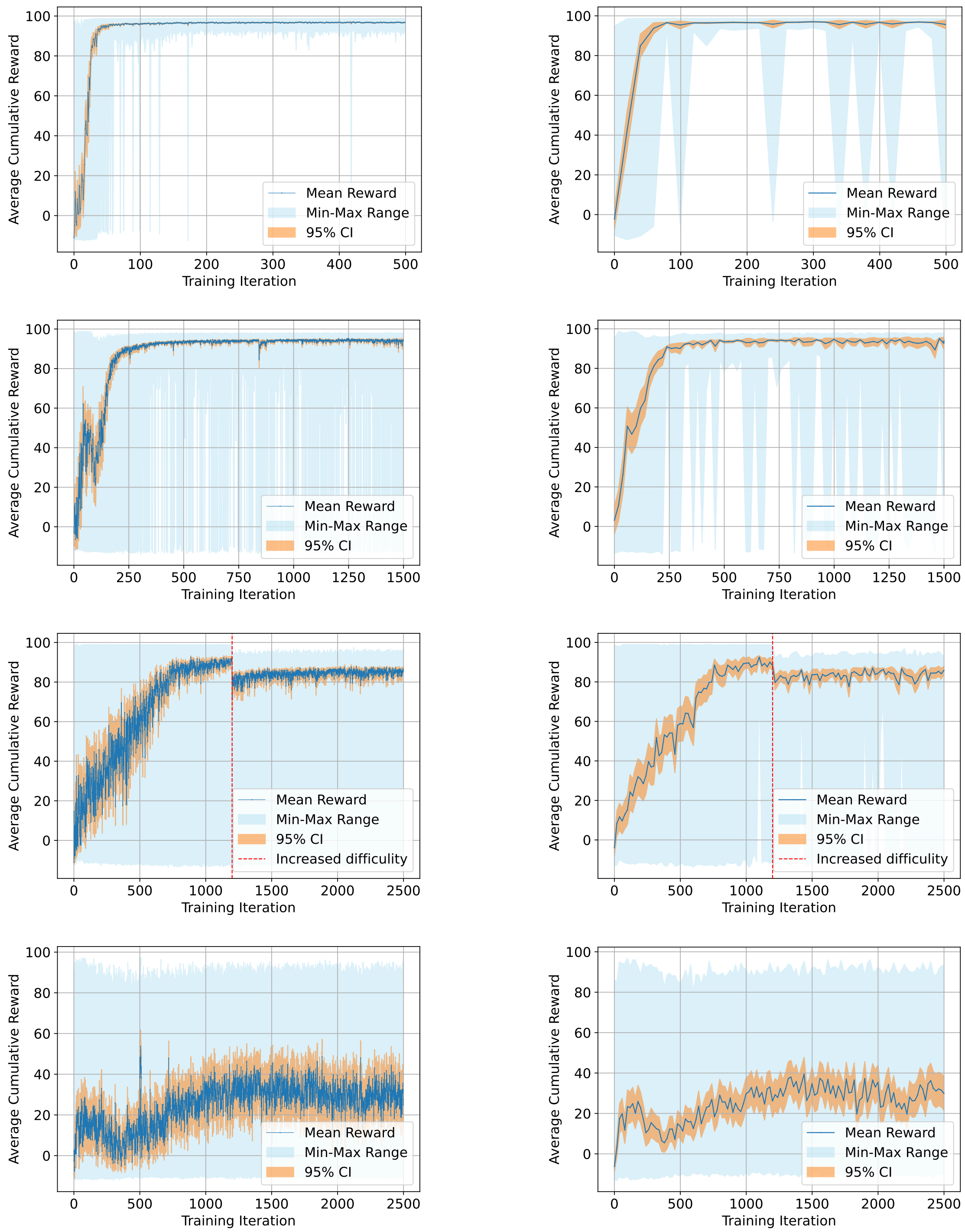


Figure 4.2: Training and evaluation results for one to four agents in the directed one-way transmission scenario. The blue line represents the average cumulative episodic rewards, the orange area is its 95 % confidence interval, and the light blue area indicates the range between the minimum and maximum values of all collected episodes. The left side shows the results for the stochastic policies that generate the data used in training. The right side shows the results when always sampling the action with the highest probability (deterministic policy). The red line shows the point where CL was used to instantly increase the game's difficulty instead of doing it gradually. The number of agents increases for each row, with the top row showing the results for one agent and the bottom row for four agents.

4.4 Two-way transmission

The increased complexity in this scenario is due to the use of a larger message buffer size. All agents had two slots available to transmit one message from each base to the other base to solve the task.

From the results presented in Figure 4.3 and Table 4.1, it is observed that one and two agents solve their task with relative ease, while the performance of three agents is bad as seen on the convergence plot, in addition to all three evaluation metrics, only solving 25 % of the tasks. Comparing this with the directed one-way scenario shows that adding a second message to the game and increasing the message buffer size introduces a larger challenge to the agents than using a directed antenna. The reason could be the larger state space size. However, one-way directed transmission has a larger action space and more possible state-action pairs than two-way transmission, thus, the reason could instead be the even sparser rewards in the two-way scenario, in which agents must deliver messages in both directions and get half the global reward each delivery.

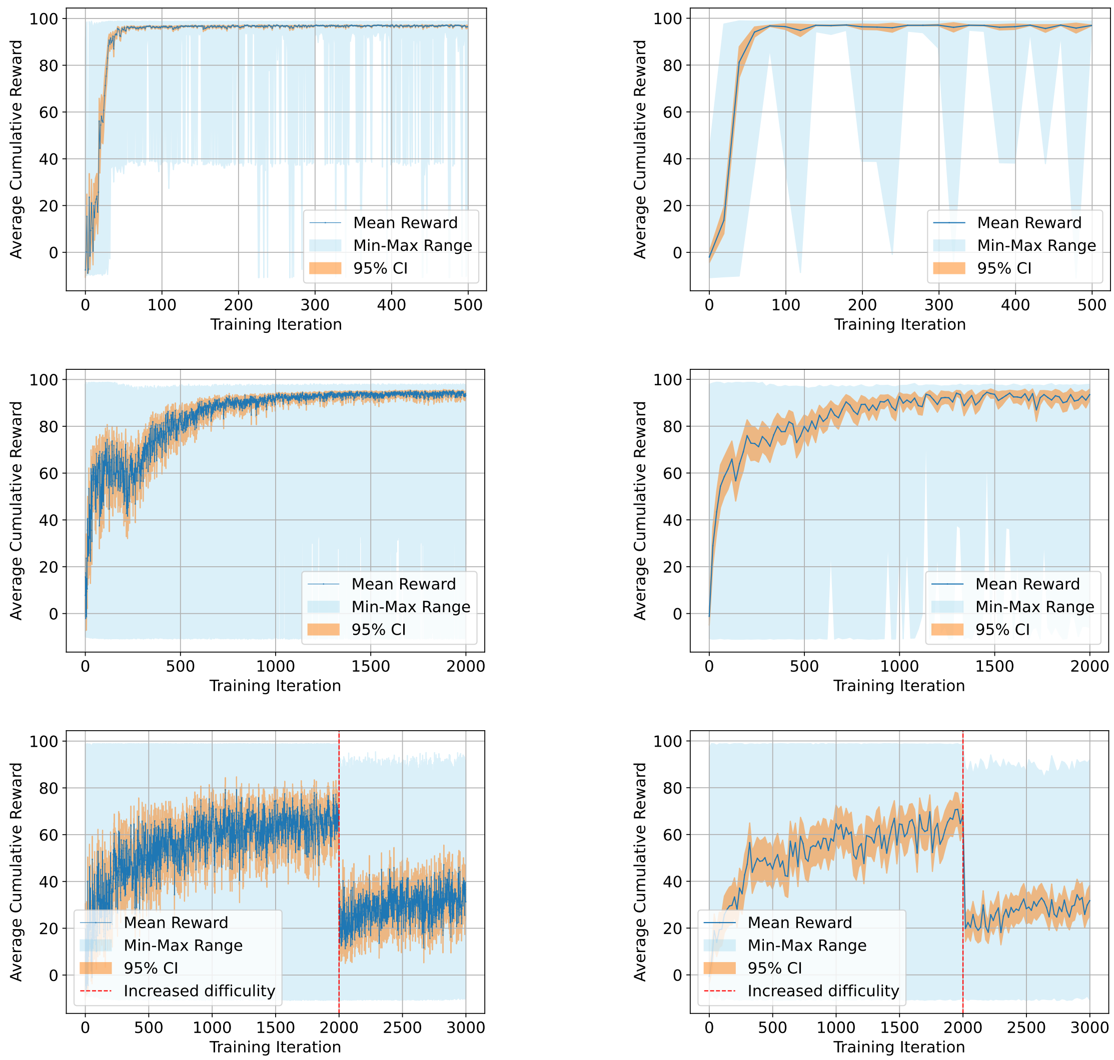


Figure 4.3: Training and evaluation results for one to three agents in the two-way transmission scenario. The blue line represents the average cumulative episodic rewards, the orange area is its 95 % confidence interval, and the light blue area indicates the range between the minimum and maximum values of all collected episodes. The left side shows the results for the stochastic policies that generate the data used in training. The right side shows the results when always sampling the action with the highest probability (deterministic policy). The red line shows the point where CL was used to instantly increase the game's difficulty instead of doing it gradually. The number of agents increases for each row, with the top row showing the results for one agent and the bottom row for three agents.

4.5 Two-way directed transmission

The two-way directed transmission scenario combined the two previous scenarios, ending up with a scenario with both an extra message and the use of a directed antenna for transmission. The training only resulted in good policies when training with one or two agents. When training with 3 agents, the policy eventually converged to a policy that gave an AECR of zero. Comparing the results shown in Figure 4.4 with the previous results, it is clear that the single-agent case managed to solve its task just as well as other scenarios with one agent, even though it took longer to converge to the optimal policy. For two agents, the performance is noticeably worse, with only 88 % instead of approximately 99 % of correct solution frequency. This decreasing performance is probably due to the increased number of possible state-action pairs, making exploration harder and magnifying the credit assignment problem.

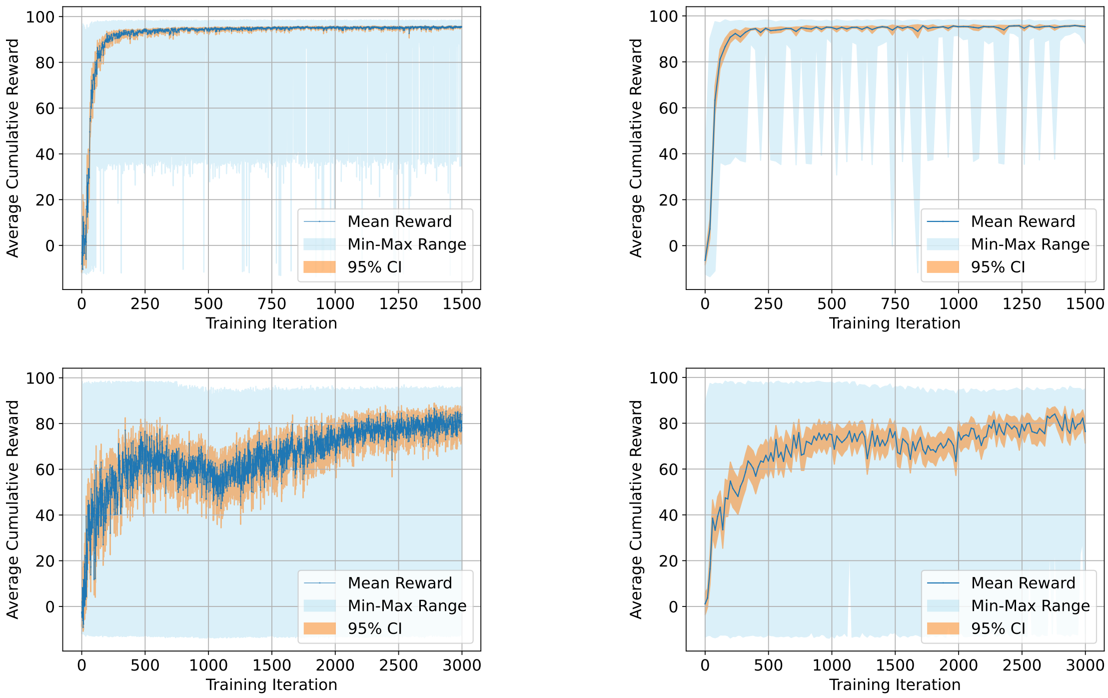


Figure 4.4: Training and evaluation results for one and two agents in the directed two-way transmission scenario. The blue line represents the average cumulative episodic rewards, the orange area is its 95 % confidence interval, and the light blue area indicates the range between the minimum and maximum values of all collected episodes. The left side shows the results for the stochastic policies that generate the data used in training. The right side shows the results when always sampling the action with the highest probability (deterministic policy). The top row shows the results for one agent and the bottom row for two agents.

4.6 Four messages

The four messages scenario had the same setup as two-way communication, but with four messages instead of two. Comparing the result presented in Figure 4.5 and Table 4.1 with the result from two-way transmission, it is clear that this scenario was a harder game to solve, with only about 1 % solution rate compared to 99 %. The low ACER of 18 indicates that only about one message was delivered each game, as each message was maximally worth $r_{\text{const}}/4 = 25$. The agent thus failed to deliver even two messages with the same reliability as in the two-way transmission scenario. This indicates that increased difficulties arise when more messages are added or the message buffer size is increased.

The difficulty in this scenario is, on top of the increased action and state spaces from the larger message buffers given to the agents, a significantly increased time horizon of each episode. This is caused by the need to deliver four messages, transmitted randomly from their respective origin bases. Thus, the policy had to account for planning for many more timesteps than in other scenarios, which is an increasingly difficult task due to the previously mentioned credit assignment problem. This increased number of time steps needed can be seen from the episode length in the table, as it is the scenario with the longest average length.

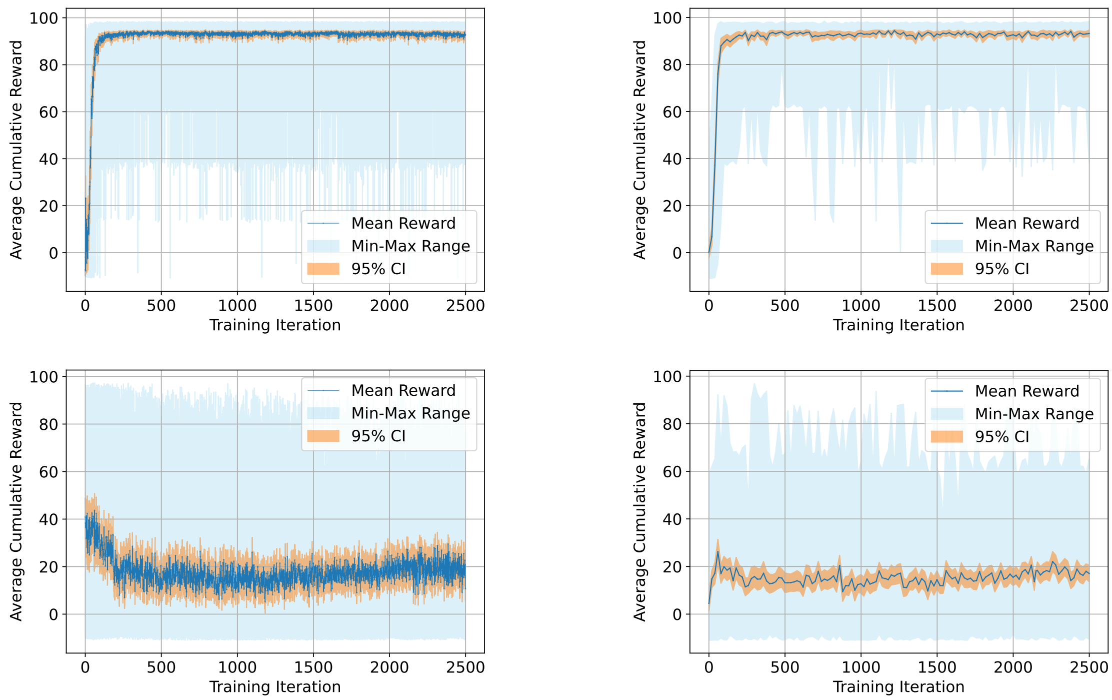


Figure 4.5: Training and evaluation results for one and two agents in the four messages scenario. The blue line represents the average cumulative episodic rewards, the orange area is its 95 % confidence interval, and the light blue area indicates the range between the minimum and maximum values of all collected episodes. The left side shows the results for the stochastic policies that generate the data used in training. The right side shows the results when always sampling the action with the highest probability (deterministic policy). The top row shows the results for one agent and the bottom row for two agents.

4.7 Three bases

The three bases scenario differed from the other scenarios, on top of the extra base, in that the distance between the bases was the same for the sessions with two to four agents. The results can be seen in Figure 4.6. The decreasing success rate in solving tasks as the number of agents increasedindicates that it is harder for the agents to learn good policies when there are more agents in play. Instinctively, the percentage of solved tasks would go up as the number of agents increases, as this would mean more agents available for message relaying in the same physical area. This decline in performance could be attributed to the increased size and complexity of the state space due to the addition of multiple interacting agents, which in turn lead to two related problems. First, the increased risk of agents getting stuck in a local maximum during training due to the increased complexity. Second, worsening of the credit assignment problem as it becomes harder to distinguish how much each agent's action contributed to the global reward.

The results from this scenario show that the decrease in performance as a function of the number of agents was not solely caused by the increased distance between the bases, but also by the increased complexity of the state and action spaces. Like in the other two scenarios with one message, there was a larger performance drop as the number of agents increased from three to four than from two to three. This shows that the difficulty in learning good policies grows exponentially with the number of agents.

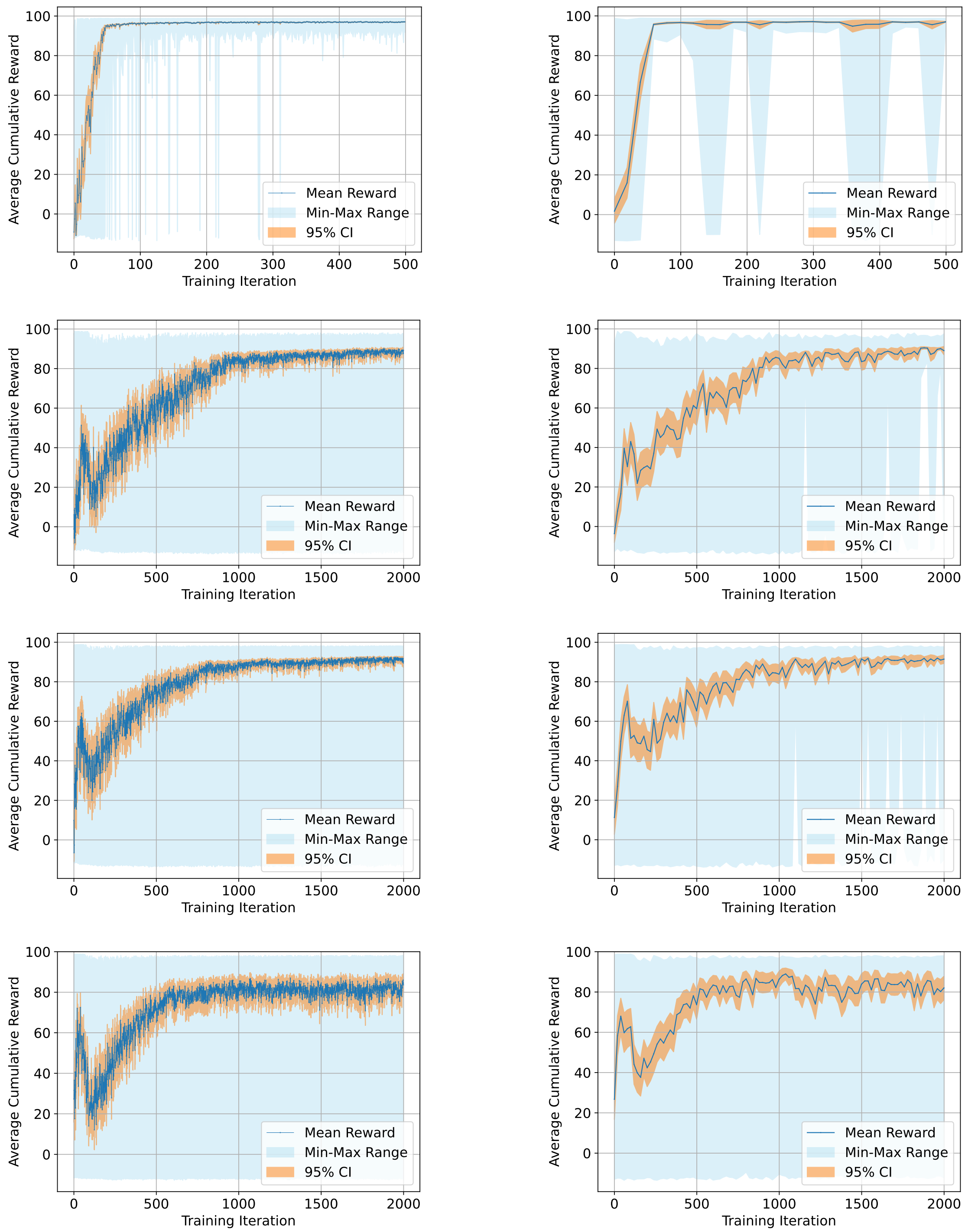


Figure 4.6: Training and evaluation results for one to four agents in the three bases scenario. The blue line represents the average cumulative episodic rewards, the orange area is its 95 % confidence interval, and the light blue area indicates the range between the minimum and maximum values of all collected episodes. The left side shows the results for the stochastic policies that generate the data used in training. The right side shows the results when always sampling the action with the highest probability (deterministic policy). The top row shows the results for one agent and the bottom row for four agents.

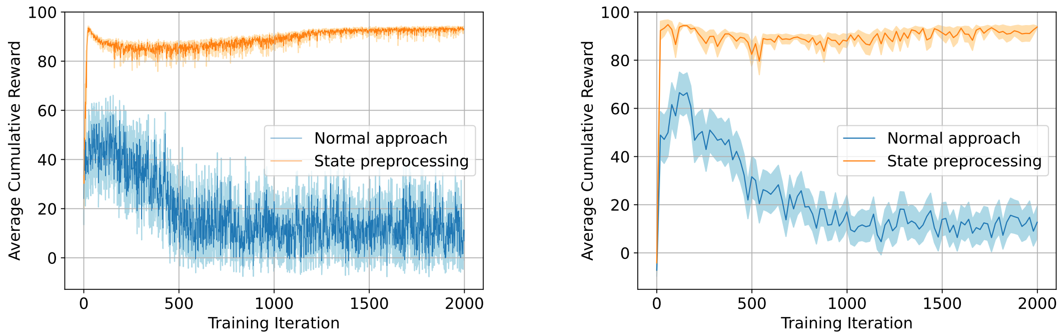
4.8 State preprocessing before neural networks

This section presents the results using the coordinate transformation as preprocessing before and postprocessing after the neural networks as described in Section 3.2.6. The coordinate transformation was implemented for each scenario with two bases and for the number of agents where the failure rate grew heavily in the corresponding non-preprocessed setup. The results are shown in Table 4.2.

Table 4.2: Agent performance when state preprocessing and action postprocessing were used, evaluated with 1000 episodes in the different scenarios. The evaluation metrics presented are the average cumulative episodic reward (ACER) with a 95 % confidence interval, the percentage of solved tasks, and the average episode length of the successfully solved tasks.

Scenario	**# Agents**	**ACER**	**Solved (%)**	**Ep. Length**
One-way	5	90.56 ± 0.76	98.70 %	32.87
Directed one-way	4	33.16 ± 2.63	44.6 %	46.71
Two-way	3	85.56 ± 1.15	94.2 %	42.63
Directed two-way	3	51.05 ± 2.01	48.50 %	71.25
4 messages	2	78.52 ± 1.14	70.1 %	58.96

Comparing the results in Table 4.2 with the results obtained using the previous method presented in Table 4.1, it is clear that the performance of the agents is greatly increased for all scenarios. In the one-way transmission scenario with five agents, the percentage of solved tasks went from 15.8 % to 98.7 %, which is also clearly presented in Figure 4.7. The two-way transmission scenario with three agents showed a similar improvement, from 25 % to 94.2 % success rate. Likewise, the performance in the four messages scenario with two agents increased substantially from 0.70 % to 70.1 %. Moreover, the average episode length for solved tasks also substantially decreased, showing that the agents not only solved more tasks but also solved them with higher efficiency. For the directed two-way scenario, three agents were used which resulted in the agents being able to solve 48.5 % of tasks instead of zero. The only scenario that did not show a substantial improvement was directed one-way with four agents, which only increased the rate of solved tasks from 37.6 % to 44.6 %.



(a) Training results **(b)** Evaluation results

Figure 4.7: One-way transmission with five agents. The blue line shows the ACER as a function of training progression for the old approach without preprocessing of the state. The orange line represents the results obtained when state preprocessing was employed. The shaded areas are the 95 % confidence intervals.

4.9 Summary discussion

In this section, the general observations and results that apply to all scenarios are discussed. Some scenario-specific questions have already been discussed above.

The credit assignment problem significantly contributed to the decreased performance in more complex games with multiple agents. Especially in the multi-agent games with sparse and delayed rewards, like in the information relaying environment, where many agents perform many actions over many tens of time steps, to be rewarded at the very end of the episode. The credit assignment problem refers to the problem of correctly identifying which actions contributed and how much they contributed to the final global reward.

A challenge that arises in all MARL problems is non-stationarity, which can stem from multiple factors. For instance, policy updates by other agents can cause the environment dynamics to shift from the perspective of any single agent. This prompts agents to update their own policies to adapt, which in turn causes other agents to adjust their policies as well, and so on. However, by employing the CTDE paradigm along with parameter sharing (see Section 2.4) for both the policy and value function, issues like non-stationarity induced by policy updates were significantly mitigated. Non-stationarity of the environment caused by all agents updating their polices simultaneously is mitigated as only one policy is updated. Although, there could still be some inherent non-stationarity present due to the transition kernel being a function of many different actors' actions. Without this approach, these problems would have had a greater effect on the training and contributed to worse performance. However, it might lead to problems, with an example being correlated exploration. The agents explore by sampling their actions from a stochastic policy, and thus, actions that the agents think are beneficial are more likely to be sampled. This means that the agents explore similarly and risk not finding certain optimal behaviors.

Scalability and sample inefficiency are other challenges that arise in MARL problems. Scalability is a problem in MARL as the observation space and the complexity of the interactions between agents increase with the number of agents. Algorithms like MAPPO become computationally expensive in these more complex games with more agents. This requires more training data to cover a sufficiently large set of scenarios, which greatly increases training time and the likelihood of convergence to a suboptimal policy. This has clearly been observed in all scenarios examined in this project, as the performance decreases with the number of agents. Sample inefficiency is a problem when using on-policy algorithms, as the experiences generated by each agent are only used once before new experiences are collected. However, when using MAPPO with parameter sharing like in this thesis, the sample efficiency is improved by all agents being able to share experiences and by MAPPO being able to perform several NN updates with each collected set of experiences, improving learning speed and stability. The use of these strategies has most likely significantly improved the training performance.

As mentioned in Section 3.2.7, training four or more agents using the original learning rate and number of epochs (number of times the networks parameters are updated using each set of collected experiences) did not result in a good policy. Instead, the learning rate was decreased from 0.001 to 0.00005 and the number of epochs increased from 15 to 45, producing the results presented earlier in the chapter. The need for a decreased learning rate indicates a parameter landscape with smaller details in in the games with more agents, as the larger learning rate probably was big enough to overshoot optimal parameter values. This is consistent with intuition that states that the state space becomes exponentially more complex with an increasing number of agents, as each agent can move independently from each other agent. The decreased learning rate worsened sample efficiency, as the total possible magnitude of NN parameter change each episode decreased. The number of epochs was increased to mitigate this decreased sample efficiency as more updates where performed with each set of collected experiences. This is why the coverage speed as a function of the number of training iterations is approximately the same for two, three and four agents in Figure 4.6. However, this meant increased computational costs and as such, about double the actual computing time needed for the same number of training iterations. To summarize, the advantage of higher NN resolution (tweaking of the weights) comes with the issue of increased computational costs. This trade-off has to be taken into account in all machine learning applications. In this case, the increased computational costs were justified as it allowed the agents to learn good policies in environments with more agents.

5 Conclusion

This project has found that using MARL to train agents to construct communication relaying networks with decentralized controls works well with a small number of agents in a fully observable environment. In the simple case of one message being transferred from one base to another, using agents with isotropic transmission, four or even five agents can reliably relay the message. With added complexities of the tasks, like equipping the agents with a directed antenna or increasing the amount of messages, the agents' ability to learn the objectives is substantially affected. This is due to the increased sizes of the action and state spaces, which exacerbate the credit assignment problem. One possible solution to mitigate the credit assignment problem could be to supplement the global rewards for transmitting messages with rewards given to the agents that belong to the communication chain that delivered the message.

However, it is not possible to draw any conclusions about whether games with a higher number of agents or more complexity are impossible to find optimal policies for using MARL. To investigate this question, further studies with a changed neural network architecture to better capture the nuances of the more complex games would be necessary. To successfully learn more complex games with more agents, it is imperative to capture the increased complexity by either increasing the number of NN parameters or reducing the complexity of the problem by, for example, doing a coordinate transformation of the state before feeding it to the NNs. This study has shown that reducing the complexity of the state space seen by the agents in such a way, if possible, significantly speeds up training and increases performance. It was also shown that using techniques like curriculum learning to iteratively improve the agents' performance over increasingly harder tasks was paramount to successfully training the agents.

In summary, this study found that MARL can be used to train agents to solve the information relaying game in fully observable environments. However, this ability to solve the task decreased when the number of agents, or the complexity of the game, increased due to non-stationarity in the environment and the credit assignment problem.

5.1 Future work

The most interesting extension to the information relaying environment would be to model it as a partially observable Markov game. A partially observable Markov game is a Markov game, except that the agents do not observe the entire state at each time step. An example could be that the agents only observe other entities within a certain distance. The largest problem with partially observable games would be that decision-making is based on incomplete information. To mitigate this, the agents could observe a history of observations, not just the current state. In practice, partial observability could be introduced by having the last couple of observations as input to the NN or using a summary statistic of the last observations together with the current observation. It could also be possible to use a recurrent neural network (RNN). These approaches would allow the agents to use older information to get a more accurate understanding of the current state [1, p.314].

It would also be interesting to include mixed discrete and continuous actions for the agents, where the physical action would be continuous to allow for more realistic motion, while the communication action remains discrete to fit with the communication model. This could be done by using algorithms that, for each timestep, output continuous values for each action, like *multi-agent deep deterministic policy gradient* (MADDPG) [10]. To translate some of the continuous action outputs to discrete actions, an activation function like the softmax function could be used on some of the logits of the output layer, leaving the rest unchanged. The discrete action(s) are then sampled from the different probability distributions caused by the softmax functions on some of the logits. A similar approach has been tested in [5] and [15]. It could also be done by using two networks for each drone, one for the continuous physical actions and one for the discrete communication actions. Essentially turning the problem into a game with two groups of agents that must learn to cooperate. A problem with the approach is that it is harder for the agents to plan how to move and communicate together, as those decisions are taken by two different neural networks.

Furthermore, it would also be interesting to investigate how the number of agents could be varied throughout the same game, or using the same policy. This allows for new agents to enter play or for other agents to leave. The obstacle is how this is encoded in the NNs, as the size of the input layer depends on the number of agents. The issue could possibly be solved using padding and feeding input entries that correspond to non-existing agents with only zeros. Another interesting extension to look at would be another approach to the state preprocessing by, for example, using equivariant neural networks to capture the symmetries in the environments.

Bibliography

[1] Stefano V Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2024. URL: https://www.marl-book.com.

[2] Christopher Amato. *An Introduction to Centralized Training for Decentralized Execution in Cooperative Multi-Agent Reinforcement Learning*. Tech. rep. Northeastern University, Sept. 2024. URL: https://arxiv.org/pdf/2409.03052.

[3] Yoshua Bengio et al. “Curriculum learning”. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. New York, NY, USA: ACM, June 2009, pp. 41–48. ISBN: 9781605585161. DOI: 10.1145/1553374.1553380.

[4] Matteo Bettini, Amanda Prorok, and Vincent Moens. *BenchMARL: Benchmarking Multi-Agent Reinforcement Learning*. Dec. 2023. DOI: https://doi.org/10.48550/arXiv.2312.01472.

[5] Olivier Delalleau et al. *Discrete and Continuous Action Representation for Practical RL in Video Games*. Dec. 2019. URL: https://arxiv.org/pdf/1912.11077.

[6] Gabriele Farina. *Topics in Multiagent Learning, Lecture 16, Markov (aka stochastic) games*. Nov. 2024. URL: https://www.mit.edu/~gfarina/2024/6S890f24_L16_sg/L16.pdf?utm_source=chatgpt.com.

[7] Onesimo Hernândez-Lerma. *Adaptive Markov Control Processes*. First edition. New York: Springer Science+Business Media, 1989.

[8] Shengyi Huang and Santiago Ontañón. “A Closer Look at Invalid Action Masking in Policy Gradient Algorithms”. In: *FLAIRS* 35 (June 2020). DOI: 10.32473/flairs.v35i.130584.

[9] Bernhard Jaeger and Andreas Geiger. “An Invitation to Deep Reinforcement Learning”. In: *Foundations and Trends in Optimization* 7 (Dec. 2023). ISSN: 2167-3918. DOI: 10.1561/2400000049.

[10] Ryan Lowe et al. *Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments*. June 2017. DOI: https://doi.org/10.48550/arXiv.1706.02275.

[11] Bernhard Mehlig. *Machine Learning with Neural Networks*. Cambridge University Press, Oct. 2021. ISBN: 9781108860604. DOI: 10.1017/9781108860604.

[12] Meta. *torchrl*. 2022. URL: https://docs.pytorch.org/rl/stable/index.html.

[13] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning*. Feb. 2016. URL: https://arxiv.org/abs/1602.01783.

[14] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Dec. 2013. URL: https://arxiv.org/abs/1312.5602.

[15] Michael Neunert et al. *Continuous-Discrete Reinforcement Learning for Hybrid Control in Robotics*. Jan. 2020. URL: https://arxiv.org/pdf/2001.00449.

[16] A Ng, Daishi Harada, and Stuart J Russell. "Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping". In: *International Conference on Machine Learning*. 1999. URL: https://api.semanticscholar.org/CorpusID:5730166.

[17] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *ICLR 2016*. June 2015. URL: https://arxiv.org/pdf/1506.02438.

[18] John Schulman et al. *Proximal Policy Optimization Algorithms*. July 2017. URL: https://arxiv.org/pdf/1707.06347.

[19] John Schulman et al. *Trust Region Policy Optimization*. Feb. 2015. URL: https://arxiv.org/abs/1502.05477.

[20] Wenjie Shi, Shiji Song, and Cheng Wu. *Soft Policy Gradient Method for Maximum Entropy Deep Reinforcement Learning*. Sept. 2019. URL: https://arxiv.org/pdf/1909.03198.

[21] Eilon Solan. *A Course in Stochastic Game Theory*. Cambridge University Press, May 2022. ISBN: 9781009029704. DOI: 10.1017/9781009029704.

[22] Petru Soviany et al. "Curriculum Learning: A Survey". In: *International Journal of Computer Vision* (Jan. 2021). URL: https://arxiv.org/pdf/2101.10382.

[23] Richard Stutton and Andrew Barto. *Reinforcement Learning: An Introduction*. Second edition. Cambridge, Massachusetts: The MIT Press, 2020.

[24] J Terry et al. "Pettingzoo Gym for multi-agent reinforcement learning". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 15032–15043.

[25] The PyTorch Foundation. *Pytorch*. URL: https://docs.pytorch.org/docs/stable/index.html.

[26] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3-4 (May 1992), pp. 279–292. ISSN: 0885-6125. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

[27] Chao Yu et al. "The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games". In: *NeurIPS 2022 Datasets and Benchmarks* (Mar. 2021).

[28] Yu Zhang et al. "UAV-Enabled Secure Communications by Multi-Agent Deep Reinforcement Learning". In: *IEEE Transactions on Vehicular Technology* 69.10 (Oct. 2020), pp. 11599–11611. ISSN: 0018-9545. DOI: 10.1109/TVT.2020.3014788.

A
Visual representation of one episode

This section shows a visual representation of one episode of the directed one-way transmission scenario with two agents, frame by frame. This scenario was chosen as it is completed in a relatively short number of time steps, while still being complex enough to clearly show the agents' learned behavior. Showing one episode visually also clearly illustrates the sparsity of the rewards, as the agents are only rewarded at the very end of these 36 steps it took to deliver the message.

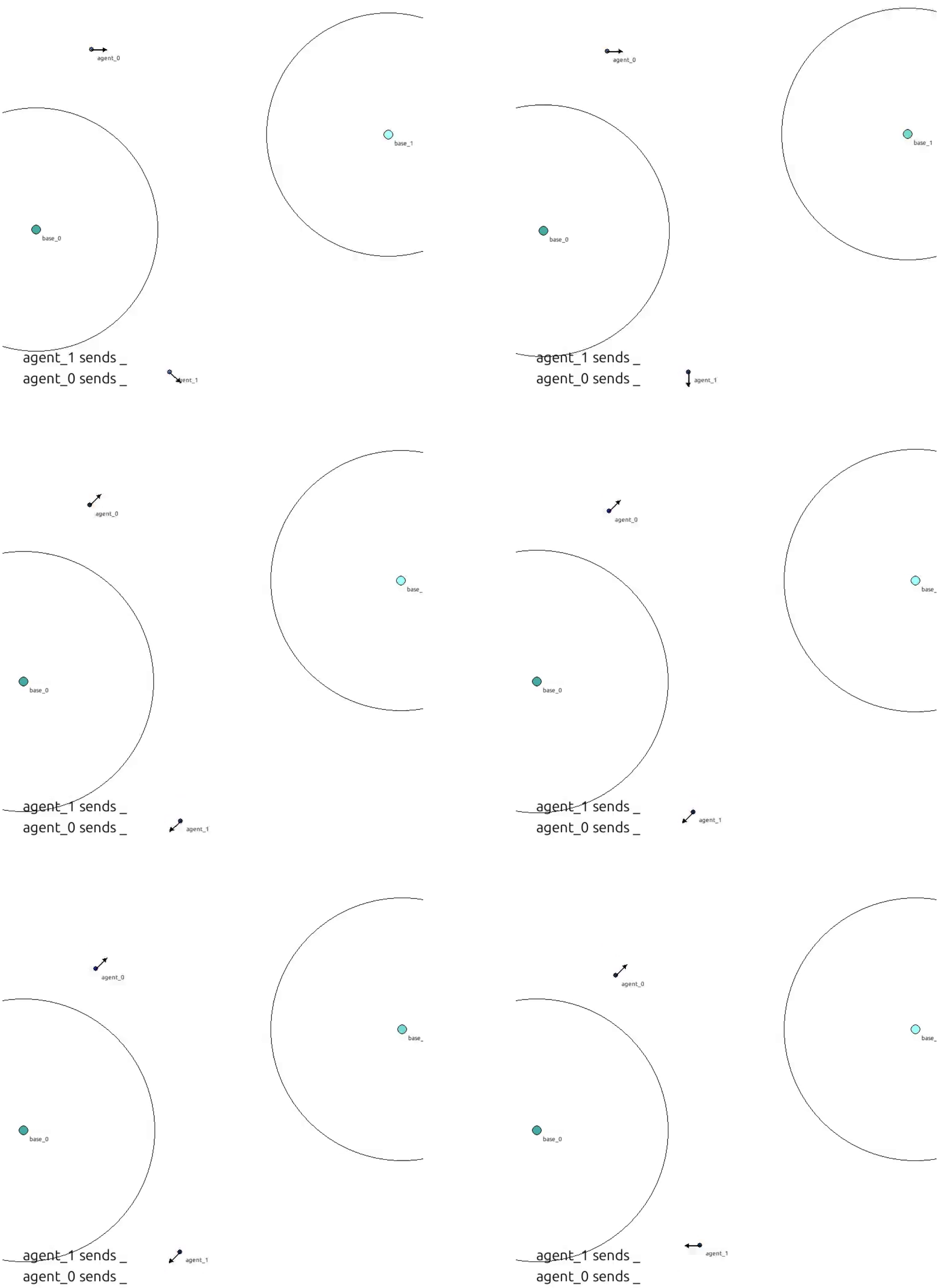


Figure A.1: Video Frames 0000–0005

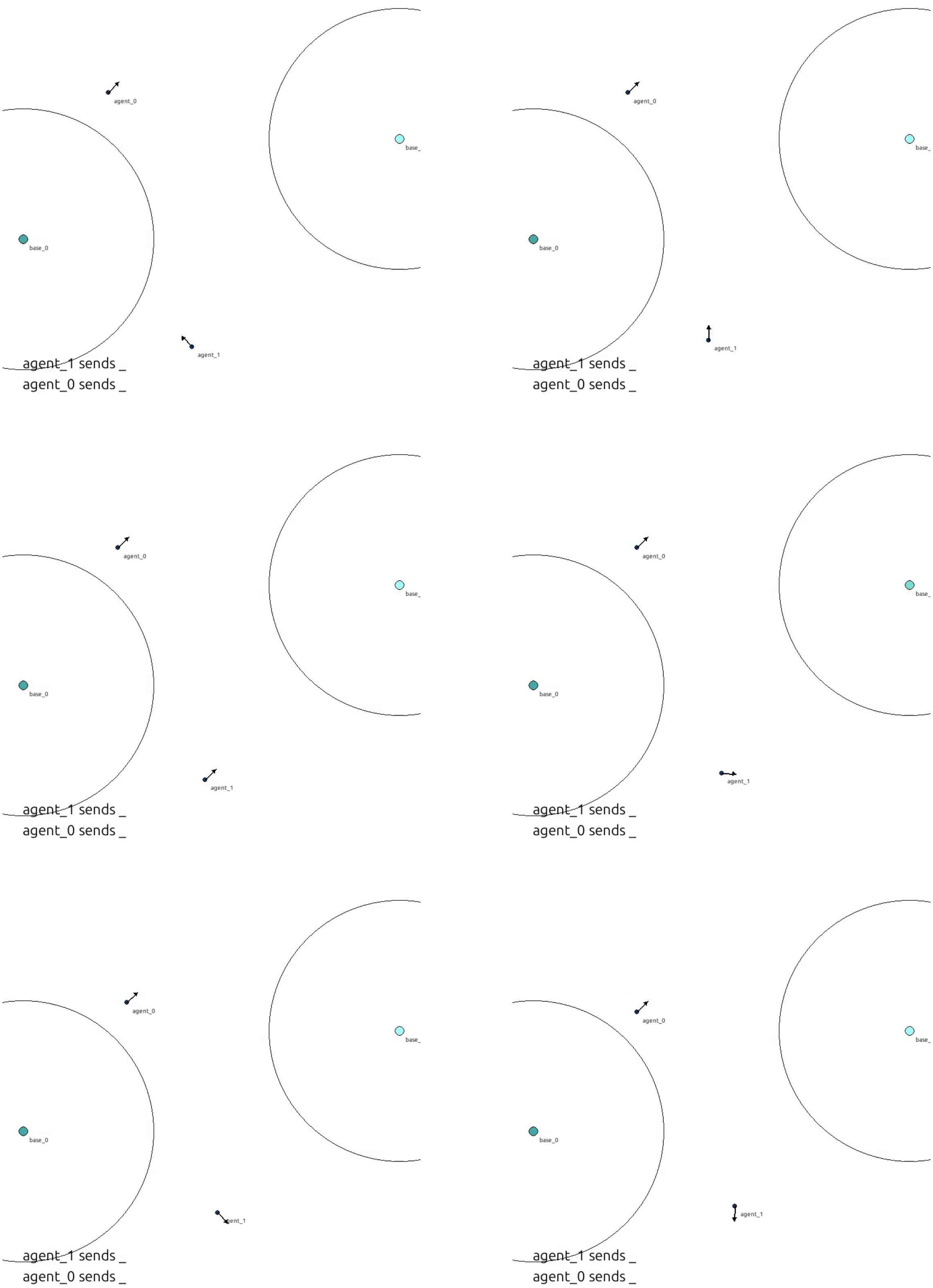


Figure A.2: Video Frames 0006–0011

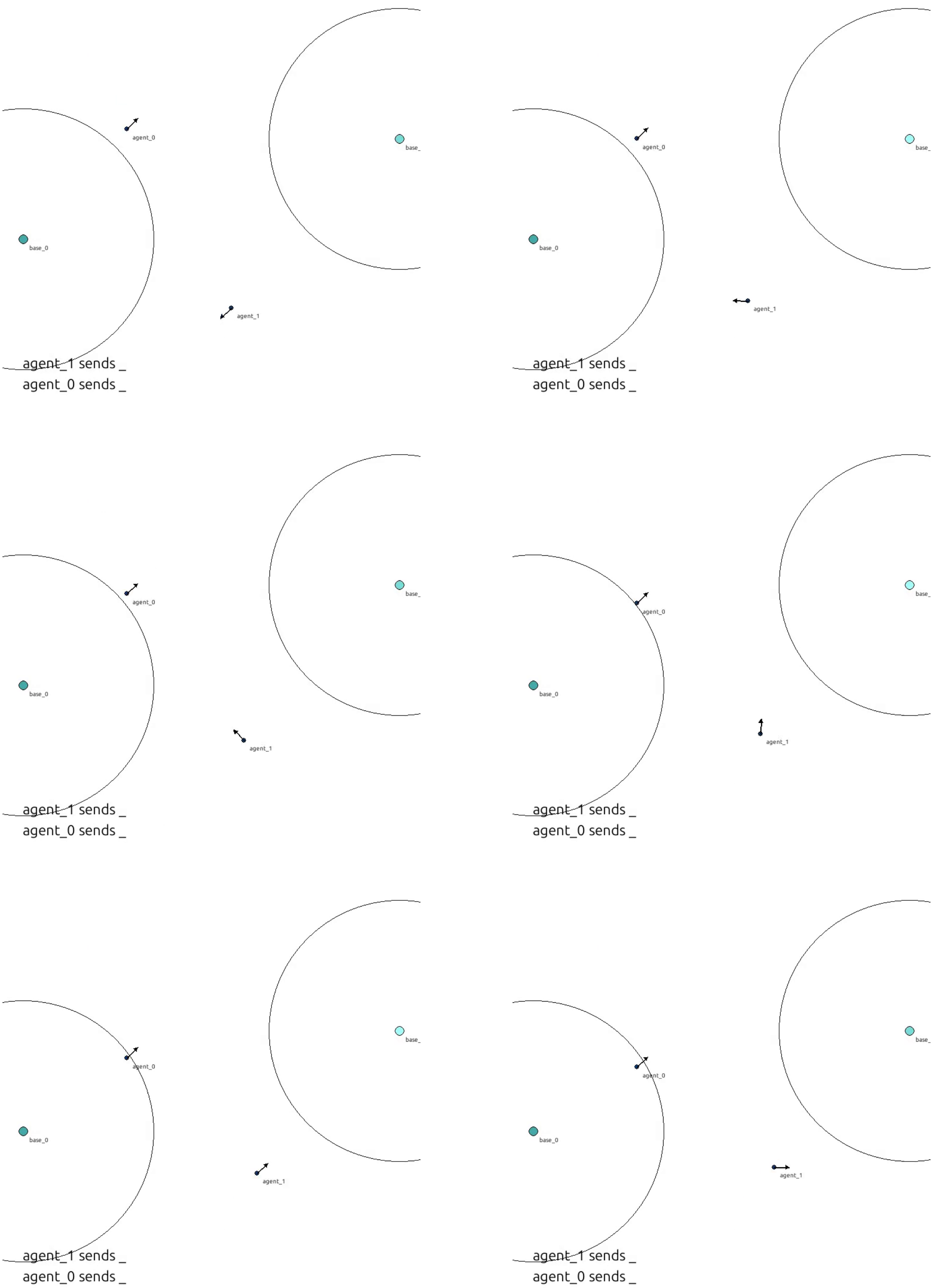


Figure A.3: Video Frames 0012–0017

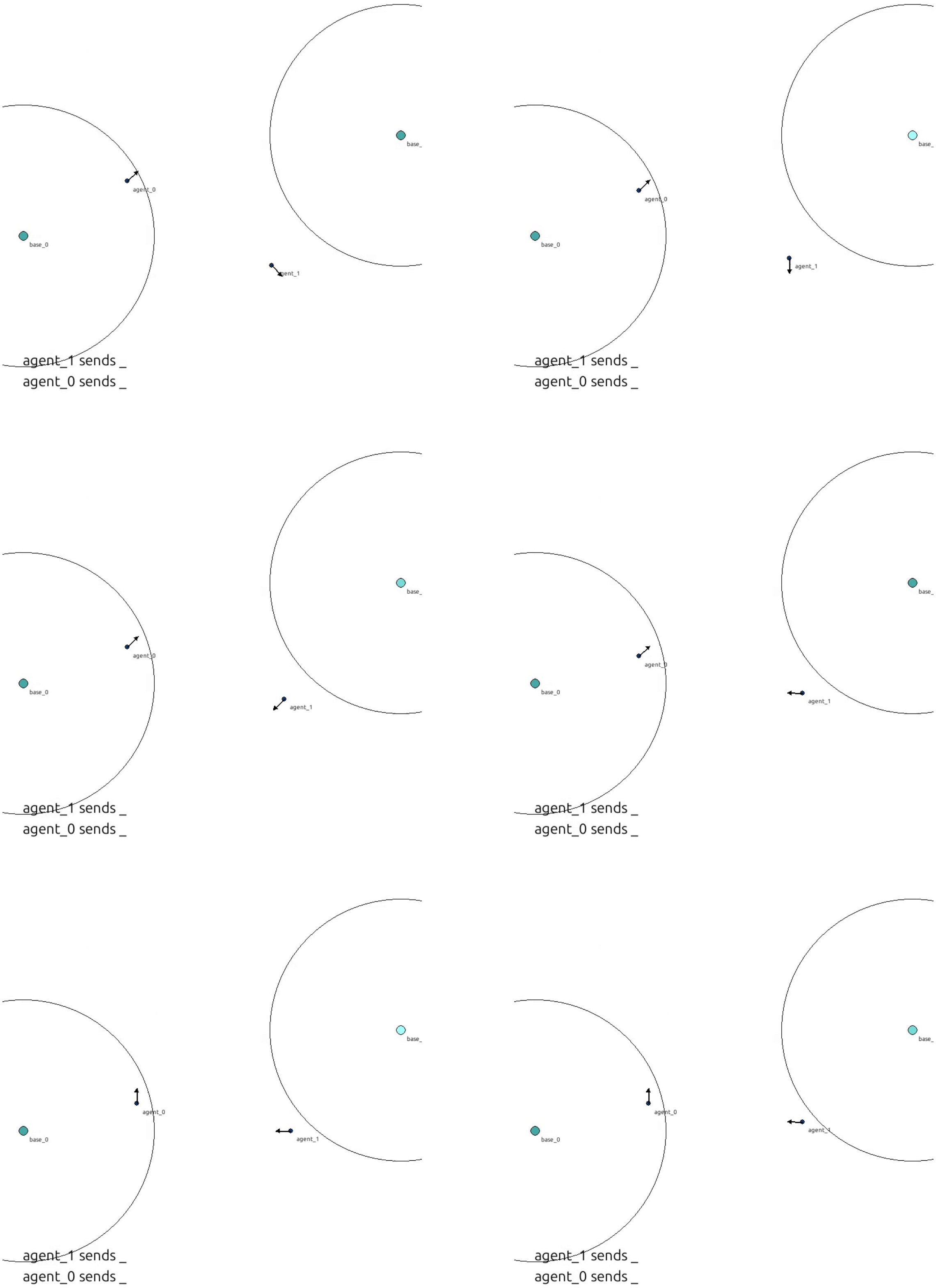


Figure A.4: Video Frames 0018–0023

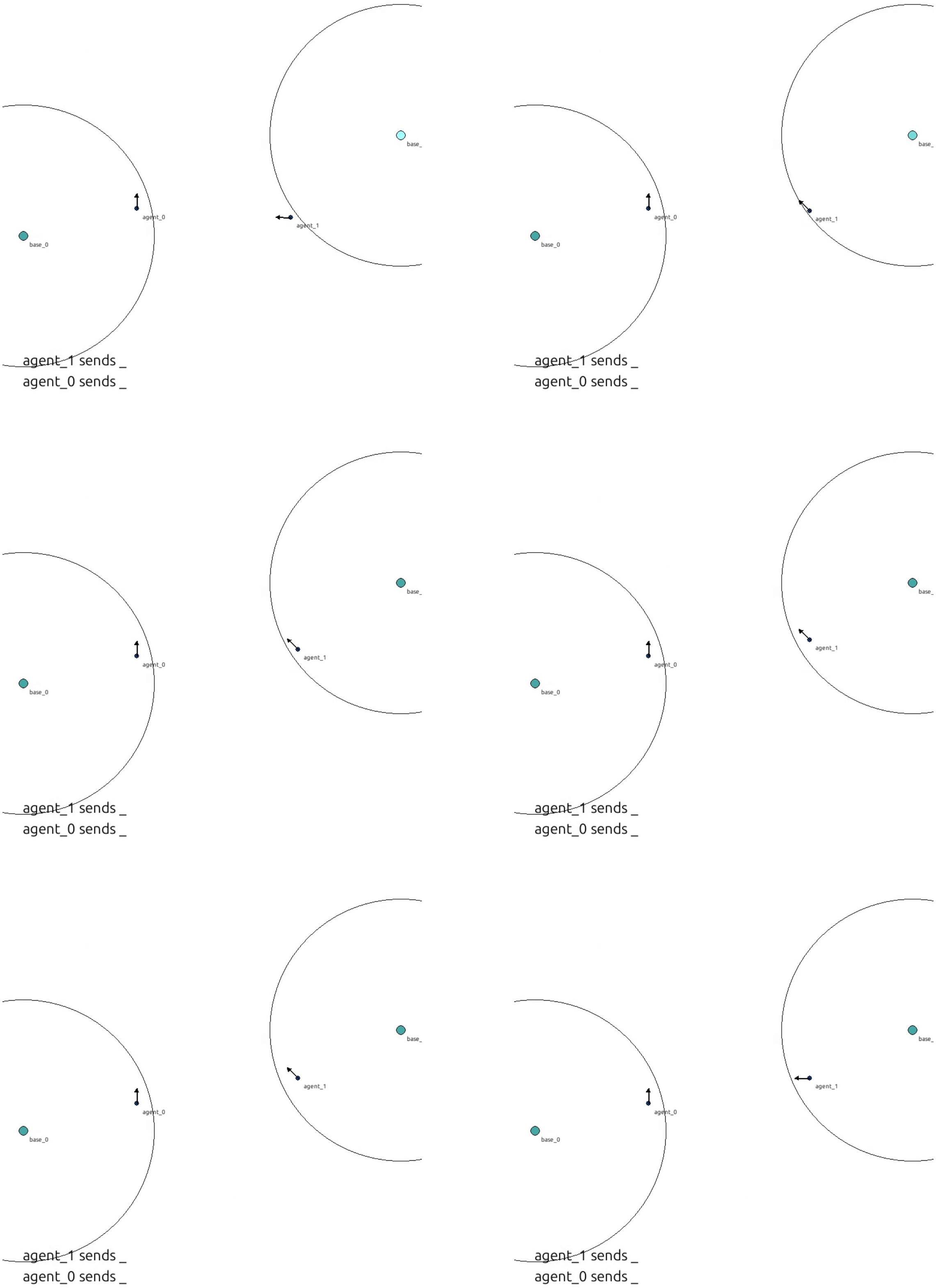


Figure A.5: Video Frames 0024–0029

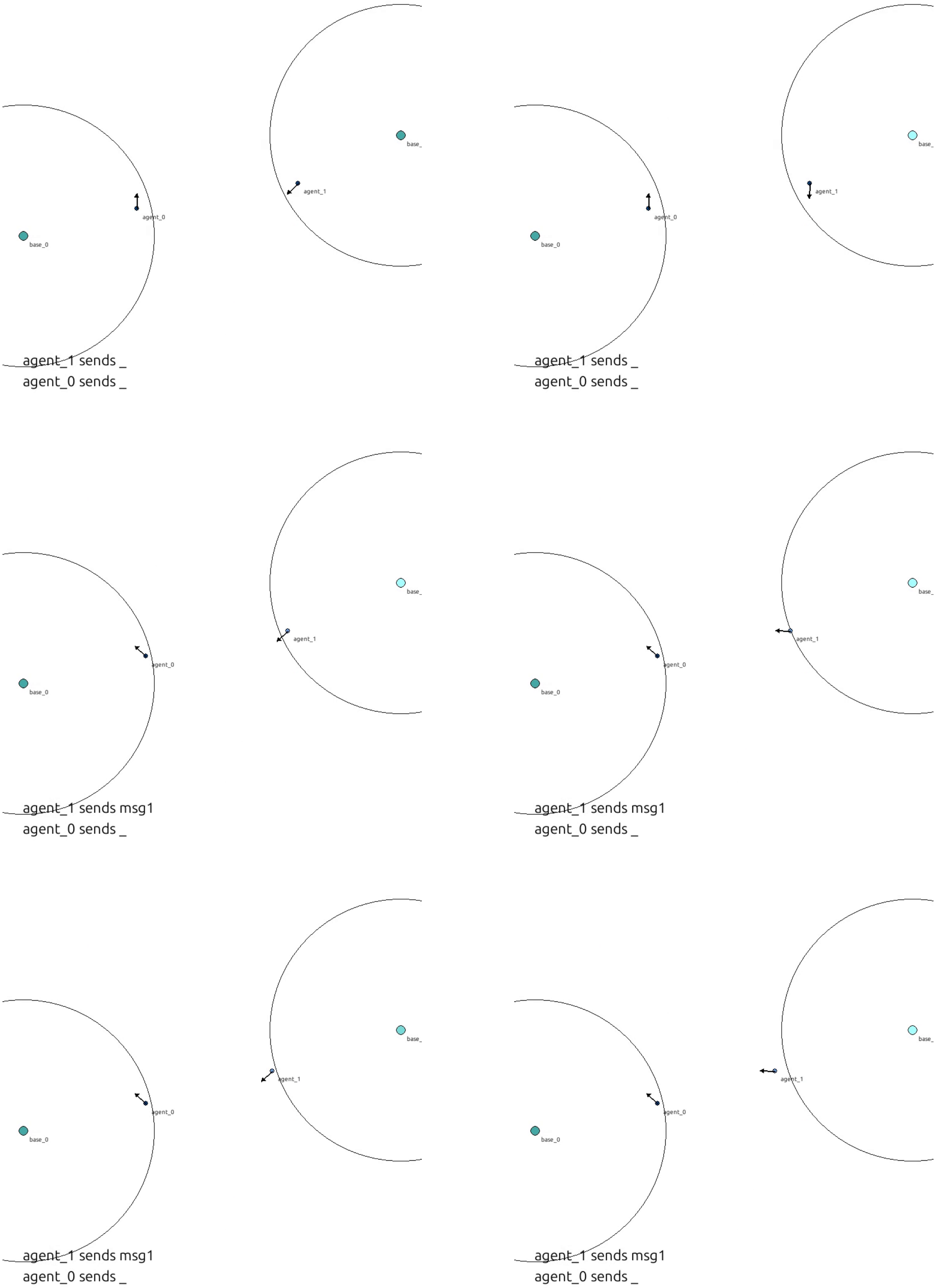


Figure A.6: Video Frames 0030–0035

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY