Karlsruhe Institute of Technology

Swarm Reinforcement Learning with Graph Neural Networks

Bachelor's Thesis
of

Christian Burmeister

KIT Department of Informatics
Institute for Anthropomatics and Robotics (IAR)
Autonomous Learning Robots (ALR)

Referees: Prof. Dr. Techn. Gerhard Neumann
Prof. Dr. Ing. Tamim Asfour

Advisor: Niklas Freymuth

Duration: Juli 17st, 2021 — January 17st, 2022

Erklärung

Ich versichere hiermit, dass ich die Arbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Karlsruhe, den 17. January 2022

Christian Burmeister

Zusammenfassung

Das stehts wachsende Forschungsgebiet des Multi-Agent Reinforcement Learning (MARL) betrachtet Echtweltprobleme, in welchen mehrere Aktoren, genannt Agenten, entweder in kooperativer, kompetitiver oder in einer gemischten Art und Weise miteinander arbeiten müssen. Dies findet Anwendung bei der Beschreibung des Verhaltens von Ameisen, Bienen oder Vögeln. Ebenso findet dieser Ansatz für Such- und Rettungseinsätze, sowie der Modellierung von Netzwerksystemen Verwendung. Hierfür müssen die Agenten etwaige große Mengen an Beobachtungen über ihre Umwelt verarbeiten und lernen, um in der Lage zu sein die gestellte Aufgaben zu lösen. Somit ist eine effiziente Datenstruktur für die Observationen der Agenten unumgänglich, um große Mengen von Agenten lernen lassen zu können. Die Lernprozesse Graph Neural Networks (GNN) wurde entworfen, um als Graph konstrurierten Daten effektiv zu lernen. Jene eignen sich gut, um die Kommunikation zwischen den Agenten zu repräsentieren. In dieser Arbeit möchten wir die Auswirkung und den Nutzen von GNNs auf das Lernen von MARL Problemen untersuchen. Der Fokus liegt hierbei auf dem Effekt mehrerer Durchläufe von GNN auf die Informationsübertragung in Umgebungen mit sehr geringer partieller Sichtbarkeit.

In dieser Arbeit stellen wir eine Trainingsarchitektur vor, welche den Proximal Policy Optimization (PPO) Algorithmus verwendet. Zusätzlich besitzt sie ein GNN, welcher problemlos auf beliebig viele Durchläufe skalieren kann. Dieser Ansatz basiert auf der bereits veröffentlichten Arbeit für eine Multi-Agent Deep Reinforcement Learning Architektur, welche man in Ruede et al. (2021) findet. Zudem wurde die Architektur um die Fähigkeit erweitert, mit heterogenen Graphen arbeiten zu können. Jene wird nämlich für unser Observationsmodell in komplexeren Aufgaben benötigt. Anschließend wird in mehreren Multi-Agent Aufgaben der Nutzen von mehreren GNN Durchläufen beurteilt.

Unsere Ergebnisse zeigen, dass unter verschiedenen Restriktionen mehrere GNN Durchläufe das Ergebnis positiv beeinflussen können. Zu diesen Restriktionen zählen kleine latent Dimensionen, geringe Observations Radien und geringer Observationsinformation. Somit können MARL Probleme von mehreren GNN Durchläufen profitieren.

Abstract

Multi-Agent Reinforcement Learning (MARL) is an ever expanding field of research, which deals with real world problems that require multiple entities, called agents, to work cooperatively, competitively, or as a mixture of both. Practical applications range from the simulation of ants, bees and birds, coordinating network systems aswell as search-and-rescue operations. The agents need to learn to correctly use potentially large amounts of observation data to make informed decisions. Therefore an efficient way to process observation data is needed for effective learning, that can scale to a large number of agents. Graph Neural Networks (GNN) allow for learning processes to work on graphs. Currently, they are a growing field of research. GNNs are a natural representation for modelling inter-agent communications. We want to investigate the effect of GNNs on learning and the resulting policy on MARL tasks. We will focus on the effect of multiple GNN passes on information propagation, especially in partial observability with a short range.

This work will introduce an architecture, that uses Proximal Policy Optimization (PPO) for training with a graph base. This allows scaling to multiple GNN passes. Our approach is based on the previosly proposed multi-agent deep reinforcement learning architecture found in Ruede et al. (2021). Furthermore we expand the architecture to work on heterogeneous graphs, which is required for our observation model of more complex tasks. We then evaluate this architecture in multiple multi-agent tasks to show the benefit of multiple GNN passes.

Our results show that under different kinds of constraints, multiple GNN passes do improve performance. These include small latent dimensions, small observation radii and low amount of observation information. Therefore, MARL Problems can benefit from multiple hops.

Table of Contents

Chapter 1.

Introduction

Any system that requires multiple agents to work together or against each other can be modelled as a Multi-Agent System (MAS). They are able to achieve more for a common goal, than a single agent could. An Agent usually has a limited sensory perception range, so cooperation is required. Observations can be shared to improve performance of the group. These agents can organize themselves and achieve more complex behavior together. They can improve the reliability and robustness of systems, as single failures can be compensated. Additionally, a MAS allows for flexibility in the number of participants. Agents can be added to the group as and when necessary.

Multi-Agent Systems have several real-world applications. For example, in Chu et al. (2020) the authors defined a traffic control system using MAS. They created a grid of traffic lights, where each of them is an agent. Then the agents learned to optimize the total waiting times at the intersections. Similarly, robot swarms can also be used to solve pathfinding problems. For example, indoor and maze-like scenarios are discussed in Aurangzeb et al. (2013). The structure of the environment causes the communication to be severely limited, which does not pose a problem for the MAS System. These Swarms are used to simulate physical systems of liquids, solids and deformable materials (Sanchez-Gonzalez et al., 2020).

Reinforcement Learning (RL) was originally designed for environments with a single agent. More advancement has come from the field of deep reinforcement learning. Because of those, it has become possible to adapt single-agent RL to Multi-Agent Systems. This is called Multi-Agent Reinforcement Learning (MARL). We will focus on simple homogeneous agents that will cooperate to solve a common task. For example, MARL has been used to solve complex strategic and tactical problems found in videogames. More specifically research can be found that focuses on Real-Time Strategy games (Zhou et al., 2021).

With more and more agents the needs for a more sophisticated structure to process observation data is needed in MARL. We want to explore the viability of using Graph Neural Networks (GNNs) as a processing step for Multi-Agent Reinforcement Learning. GNNs allow for learning of Neural Networks with graphs inputs. Graphs are a natural and fitting representation for the communication between agents. They also allow for data processing between agents that are spatially adjacent, which fits with many real-world applications.

Our Approach is based on the previously published work found in Ruede et al. (2021). It already describes a GNN for processing observation data. We improve on that by implementing multiple GNN passes inspired by Jiang et al. (2018). We want to run our experiments in continuous environment, which is why we chose not to use the learning approach found in Jiang et al. (2018), but to the same algorithm as Ruede et al. (2021). We want to show that multiple GNN passes can be a benefit to the training results.

This thesis is structured like this: Firstly we will go over the necessary preliminaries and notation needed to understand this thesis in Chapter 2. This includes Reinforcement Learning basics in the single-agent case, then addresses how it is applied in multi-agent tasks and is followed by a brief overview of Graph Neural Networks. In Chapter 3 related work will be discussed. The subsequent Chapter 4 is focused on the details of our proposed model. It will describe our policy architecture respect to both homogeneous and heterogeneous Graph Neural Networks. Our experiments, including the general setup and the specific tasks that we trained our mode on, are described in Chapter 5. The goal of our experiments, their interpretation and evaluation of our results can be found in Chapter 6. We will complete this thesis by distilling the conclusions about our findings and provide future avenues for improvements in Chapter 7.

Chapter 2.

Preliminaries

2.1. Notation

The following table contains common used notation through the thesis. It features mathematical, Reinforcement Learning (RL) and Graph Neural Network (GNN) notation.

Table 2.1.: Overview of mathematical, RL and GNN notation

Symbol	Description
$\oplus$ or $\otimes$	Aggregation
a	Agent
act	Action
π	Policy
r	Reward
s	State
o	Observation
x_v	Node-features
x_e	Edge-features
X_g	Global-features
$\mathscr{N}_i$	Neighborhood for agent i

2.2. Reinforcement Learning

In Reinforcement Learning (RL), an agent interacts with an environment to iteratively learn about its task. Every timestep it uses an action a and the environment generates a reward r, which is used to update the strategy of the agent, called policy π. The policy π can be deterministic $\pi(s) = a$, or probabilistic $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$. Rewards can be dense (outputted at every step), or sparse (outputted irregularly). It also generates an observation o. The observation is based on the entire information of the environment state S^e (fully observable) or part of the environment state S^e (partially observable). In both cases it is a potentially noisy mapping of the state. It may still be different from the state, even if fully observable. This is used by π to generate a new action for the next time-step.

Single-Agent Reinforcement Learning in fully observable environments are formally described using a Markov Decision Process (MDP). A MDP relies on the Markov property. It defines that the current state is enough to infer future states through a transition probablitity. So the history of previous states is not needed. Therefore it can be described by the 4-tuple (S, A, T, R), where:

- S is a set of states called the state space.
- A is a set of actions called action space.
- $T(s, a, s')$ is the state transition function. For a given action a and state s it outputs the probability of landing in state s'. It's probability is denoted with $\mathbb{P}(S'|S, a)$.
- $R(s, a, s')$ is the expected immediate reward function. It returns the reward for transitioning from state s to state s'. It may also include the action.

If the environment is partially observable, a MDP can be expanded to a partially observable MDP (POMDP). Now, the policy takes the observation as input instead of the actual environment state. The observation is based on given agent and includes subjective partial information. We can formally describe a POMDP as a tuple (S, A, O, P, R, Z), where we have in addition to MDPs:

- O a finite set of observations. An observation may be any potentially noisy mapping of the state. As such, it may contain incomplete or subjective partial information.
- $Z^a_{s'o}$ an observation function which gives us the probability of observation o from the agent's perspective for a new environment state s'.

To evaluate a policy, we can use two functions, the state-value function and action-value function. The state-value function $V_\pi(s)$ tells us the expected future reward of a given start state s and a policy π. This is the expected sum of the rewards we will get following the policy starting from s. The sum is further discounted by the discount rate $\gamma \in [0, 1]$. It is used to discount rewards that are further in the future, any may be interpreted as the probability of a rollout continuing after any given step. The value function may be described as

$$V_\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t r_t | S_t = s\right]$$

On the other hand we have the action-value function $Q_\pi(s,a)$, also known as Q-function. It tells us the discounted expected future reward for a policy π, a state s and an action a. This describes how good it is to be in a given state and to use a given action, so this can be formulated as

$$Q_\pi(s,a) = \mathbb{E}_\pi[\sum_{t=0}^{\infty}\gamma^t r_t | S_t = s, A_t = a]$$

So, an optimal policy π^* maximizes the value-function and the action-value-function. For simple and discrete environments, py may be represented as a tabular mapping from state to action. For more complex tasks, like continuous environments that have a lot of states, a direct representation of π is not practical. The action-state space explodes in size. Therefore, an approximation is required.

In this thesis, we consider a parameterized policy pi, whose parameters are optimized using policy gradient methods. The optimal policy is denoted as π^*. In Actor-Critic methods, we optimize two participants. The actor which controls how the agent behaves is based on the policy π. The critic measures how good the current policy is by using the value-function. While learning, the actor tries to improve the policy by using the advice from the critic, which learns how to correctly critique the policy of the actor.

In order to reduce variance when training, we can use the advantage function $A^\pi(s,a)$. It describes how much extra reward the agent could get by taking a given action.

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

This is called Advantage Actor Critic (A2C). Common Actor-Critic algorithms include Proximal Policy Optimization (PPO) (Schulman et al., 2017), Trust Region Policy Optimization (TRPO) (Schulman et al., 2015), Trust Region Layers (PG-TRL) (Otto et al., 2021).

2.3. Multi-Agent Reinforcement Learning

In this thesis we explore Multi-Agent Reinforcement Learning (MARL) or Swarm Reinforcement Learning (SwarmRL). In a Multi-Agent System (MAS) n agents influence the environment. The set of actions for all agents at a given timestep is called the joint actions $A = \langle a_1, ..., a_n \rangle$, where a_i is the action of agent i. Each agent has its own observation and we therefore have a joint observation $O = \langle o_1, ..., o_n \rangle$. There are a few extensions to POMDPs for multi-agent environments, for example decentralized POMDPs (Dec-POMDP). The main difference to POMDPs can be seen in the joint observations and joint actions.

- I is the set of n agents.
- S is a set of states.
- A is a set of joint actions.
- $T(s, a, s')$ is the state transition function. For a given joint action a and state s it gives the probability of landing in state s'.
- O is a set of joint observations.
- $Z^a_{s'o}$ is an observation function which gives us the probability of an observation o from an agent a's perspective for a new environment state s'.
- $R(s, a, s')$ is the expected immediate reward function. It returns the reward for transitioning from state s to state s'. It may also include the action.

2.4. Neural Networks

The Neural Network is the basis for every kind of approximate function learning. Each neuron can hold a real value. Layers are mulitple neurons in a row, where neurons in a given layer are used as inputs for the neurons of the next layer. The new value y is a weighted sum, with weights w, of the neurons of the previous layer it is connected to plus a bias b. This is describes a linear function. The result is then put through an activation function ϕ to control value ranges.

$$y = \phi(\mathbf{W}^T \mathbf{x} + \mathbf{b})$$

If each of the neurons of the previous layer is connected to each node of the next layer we call that fully connected. The first layer is the input of the function and the last layer is the output. A Multi-Layer-Perceptron has multiple fully connected layers, where layers i receives input from layer $i-1$ and in turn produces the input for layer $i+1$.

2.5. Message Passing GNN

In this thesis a graph is given as $G = (V, E, X_v, X_e, X_u)$. We have n Nodes $|V| = n$ with a node feature Matrix $X_v \in \mathbb{R}^{n \times k}$ that contains k features for each node. They describe attributes of a given node for a given task. Edges are defined as $v_i, v_j = e \in E \subseteq V \times V$, $|E| = l$ with their own edge feature Matrix $X_e \in \mathbb{R}^{l \times m}$. We additionally use global features $X_u \in \mathbb{R}^{q \times r}$ that are used to define features for the entire graph. The neighborhood of a node v_i is defined as every node v_j, where $i \neq j$, which is connected to v_i via an edge. Which is denoted as $\mathcal{N}_i = \{v_j \mid (v_i, v_j) \in E\}$.

In Graph Neural Networks (GNNs), we want to learn functions that run on graphs as inputs. The function should have the same result for two isomorphic graphs. Edges are represented by using an adjacency matrix A, where

$$a_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & otherwise \end{cases}$$

Nodes and edges have features, called node-features X_v and edge-features X_e. To preserve isomorphic graphs, we want the function f of our neural network to be permutation equivariant, i.e.

$$f(PX_v, PAP^T) = Pf(X_v, A) \text{ for a given permutation} P$$

In order to do that we define a function $g(x_{v_i}, \mathcal{N}_i)$, that uses a node-feature and the neighborhood $\mathcal{N}_i$ of a node v_i. It is called the local aggregation function. Function g should not depend on the order of nodes in the neighborhood, in other words it should be permutation invariant $g(x_{v_i}, P\mathcal{N}_i) = g(x_{v_i}, \mathcal{N}_i)$. We can then construct f using g so that f achieves permutation equivariance in the following way

$$f(X_v, A) = \begin{pmatrix} g(x_{v_1}, \mathcal{N}_1) \\ g(x_{v_2}, \mathcal{N}_2) \\ \vdots \\ g(x_{v_n}, \mathcal{N}_n) \end{pmatrix}$$

f is called a single Graph Neural Network block or a single GNN hop. In practice f is learned by using an MLP Neural Network. By applying this function multiple times to the graph, information can flow between multiple neighborhoods, called multiple GNN hops. A node is able to get information about the neighbors of his neighbor. The local aggregation function g can have one of three flavors.

$$\begin{aligned} \text{Convolutional: } & g(x_i) = \phi(x_{v_i}, \oplus_{j \in \mathcal{N}_i} c_{ij}\ \psi(x_{v_j})). \\ \text{Attentional: } & g(x_i) = \phi(x_{v_i}, \oplus_{j \in \mathcal{N}_i} a(x_i, x_j)\ \psi(x_{v_j})). \\ \text{Message-Passing: } & g(x_i) = \phi(x_{v_i}, \oplus_{j \in \mathcal{N}_i} \psi(x_{v_i}, x_{v_j})). \end{aligned}$$

In Convolutional GNNs we use constants c_{ij} for each edge which states how much Node j influences the features of Node i. This turns the neighborhood aggregation into a weighted sum. So it is only dependend on the structure of the graph. Attentional GNNs are similar, but the weights are now a function $a(x_i, x_j)$ of Node i and Node j that is learnable. In Message-passing GNNs we have no weights, but the neighbor feature transformation function ψ now takes both node features into account $\psi(x_{v_i}, x_{v_j})$. This way, the sender and receiver work together to create messages that are used across an edge.

Chapter 3.

Related Work

This chapter will highlight some related work in the field of Multi-Agent Reinforcement Learning (MARL) and Graph Neural Networks (GNN).

3.1. Multi Agent Reinforcement Learning

Homogeneous and Heterogeneous Agents:
Our thesis used homogeneous agents, where each of them has the same capabilities. They can sense the same information and do the same actions as any other agent. In the heterogeneous case, the agents can differ in a variety of ways. They are better at certain actions, or are not able to do some of them, having better observation range or different kind of sensors, or their entire movement dynamics can be different. In Ishiwaka et al. (2003) the authors defined an approach to the Pursuit problem Section 5.2.3. Their evader used a deterministic set of rules for movement. Each of their agents used their own Q-Learning network to make its own decision. If an agent can see the evader, it will directly go to it, otherwise it can predict the position using communication between the evaders. Each agent started to create its own prediction which led to heterogeneous behaviour. Some agents chased the evader and others tried to ambush it.

Different Topologies:
How the agents are organized in terms of communication and coordination is called the topology. They can work together in a hierarchical system. In Servin and Kudenko (2008) the authors adapt an MARL approach for detecting network intrusions in computer networks. They apply RL to multiple agents, each responsible for several systems that report heterogeneous

sensor data of the network, called a cell. Because of bandwidth restriction and for better modelling real life computer networks, they opted to use a hierarchical approach. One cell itself can communicate with an agent higher in the hierarchy. Each of the Cells and highest hierarchy learn to monitor the system and to look for abnormal signals. One other benefit they outlined is that this structure would be easy to adapt to different network architectures.

In addition to classic hierarchical structures, agents can be organized in fractal structures. The holon was originally proposed as a model for the social behaviour of biological species back in 1968 (Koestler, 1968). It describes a self-similar structure of connected individuals. In MARL, it has been used to describe a flexible structure of agents. In the holon hierarchy we describe the holons on the same level as sub-holons and the holon of the higher level as super-holons. In Abdoos et al. (2013) the authors use this structure to solve the traffic signal control problem of larger cities, with two levels of holons. The holonic structure allows for communication between sub-holons of a level (intra-level), similar to other MARL Approaches and communication to the super-holon of a level (inter-level). Information about observation, actions and rewards are abstracted for the higher super-holon. Through subdividing the problem, they achieved less average delay time and higher flow rate.

Cooperation: Team, Adversarial, Cooperative
How the agents interact with each other to solve the task at hand can also be different. Tasks can be of adversarial nature. Each of the agents is competing against each other. The authors of Zheng et al. (2018) defined a platform for research on MARL called MAgent, which includes multiple tasks. One of these tasks is called gather. Here multiple agents have to compete for a limited number of resources. They can get a reward directly by eating the food. But they are also able to eliminate other agents and by doing so can monopolize on more food. Their training showed that the agents rushed into a central amount of food to gather as much as possible. Only when two agents were close enough, they tried to eliminate each other.

But agents are not limited to adversarial behaviour, they are able to learn to work together in a cooperative environment. Foraging is a cooperative task used in Yogeswaran et al. (2013). It is based on animal behaviour that can be found in ants, for example. The environment is filled with food and the goal is to collect it. To do so, the agents have a single central drop off point. They are required to search the environment, which is filled with obstacles. Therefore, navigation is an extra part of the problem. The level of cooperation can be increased by using communication to inform other agents of resource locations and to coordinate searching efforts.

In a team-based scenario, the agents are segmented into multiple teams that each must compete with each other. In Baker et al. (2019) the OpenAI team used a team-based task of hide-and-seek. They only used the line-of-sight for the seekers and hiders as the goal and reward for both teams. The reward was given out for the whole team. They have shown that the agents developed multiple stages of strategy and counter-strategy between both teams.

3.2. Graph Neural Networks

Graph Neural Networks (GNN) have been used for different problems and in different structures, but the research is still in its early stages. Graph Network-based Simulators (GNS) is a structure proposed in Sanchez-Gonzalez et al. (2020). Their goal was to create a structure that was able to learn physical interactions between particles, for liquids, solids and deformable materials. Graphs are used as representations for particles with message-passing utilized to express the physical dynamics. Edge represented the interaction of the particles in a radius. They encoded the physical state into a latent graph representation which then went through multiple GNN hops sequentially. The result of these were decoded back into a physical state. Their model was able to ablate from predictions of a single physics-timestep to different starting conditions, more timesteps and more particles in evaluations.

Graph Convolutional Networks (GCNs) were introduced for node-wise classification using graphs (Wu et al., 2019). Multiple graph layers are used that can be learned by a neural network. At the end of the stack, the output is used by a classical linear classifier. Each of the layers has two steps. First, the features of the graph are propagated in a given layer using convolutional constants and the neighborhood as discussed for convolutional GNNs in Chapter 2. Secondly the features are transformed for the next layer, similar to a Multi-Layer Perceptron (MLP). The model outperformed several other recent methods for classification significantly.

An extension to the GCNs is the Graph Attention Network (GAT), which uses the attentional flavour. The convolutional weights in GCN directly depend on the graph structure, which makes GCNs not as flexible. Instead Veličković et al. (2018) uses an attention function to improve upon convolutions. The attention is itself a neural network and therefore learnable. As it only relies on nodes, it is essentially not restricted by the graph structure. In practice though, the combination of the attentional data is used for aggregating of neighbourhood features. Their approach was computationally efficient and does not require, that the entire graph structure is known upfront, which addresses a scalability issue.

Liu et al. (2020) features an application for GNN in MARL called G2ANet. They used two different layers of attentional GNN. Its output is then used for different MARL-Learning techniques. The first layer uses a hard attention function and takes a fully connected graph as input. It has only a binary output and is used to describe if there is a meaningful interaction between two agents. The second layer is a classical soft attention with a value range that defines how important a given edge is to the agents. This way they are able to create a graph structure that itself is learnable. They concluded that this approach was performing better than a lot of the state-of-the-art algorithms they compared it against.

Chapter 4.

Swarm Reinforcement Learning with Graph Neural Networks

First we will introduce the policy architecture used as the base. It is designed to work with PPO and uses the latent node features of the observation for the actor and critic. This base is then expanded with a Graph Neural Network structure that allows multiple hops through a stack of message passing blocks. To support heterogeneous graphs, needed for tasks like pursuit, the architecture is extended to work on multiple graphs as inputs. Most of the changes required for heterogeneous graphs are located in the edge-, node- and global-modules respectively.

4.1. PPO - Policy Architecture

An overview of the policy architecture as desribed below can be seen in Figure 4.1.

Our environments can have different observations structured as a graph $G = (X, E, U)$. For our experiments we view the set of agents as nodes for the graph. Additionally, the interactions between the agents are described by the edges. For now we will talk about homogeneous graphs. Each agent a_k uses information it knows about itself and what it knows about it's neighboring agents $a_i \in \mathcal{N}_k$. The neighborhood is determined by a culling method like kNN or the euclidean distances. The specific data for each environment is described in Chapter 5. Information about a_k is used as the node-features x_k for the graph. The edges in E describe neighborhood relations for a given $\mathcal{N}_k$. For an edge $(x_i, x_j) = e \in E$ edge-features on the other hand define the individual observations from a_i about the state of the agent a_j. Ruede et al. (2021) implicitly uses a similar observation structure. Edge-features are created using an

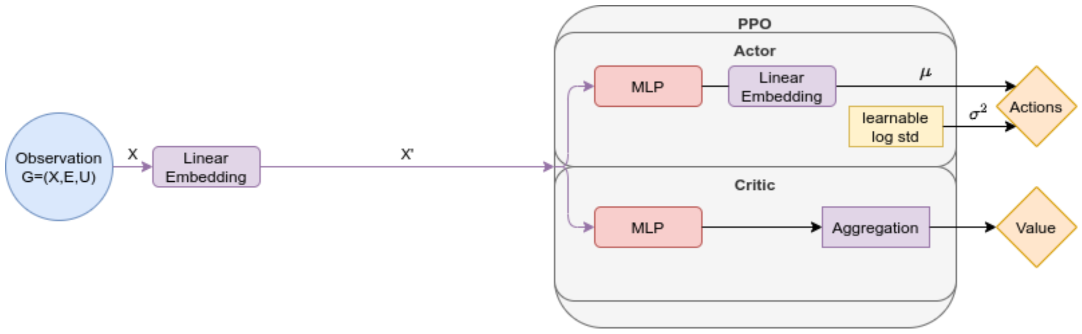


Figure 4.1.: Policy architecture used for Swarm Reinforcement Learning with Graph Neural Networks. The node features of the observation graph is the only data used. Here the actor and critic use the same embedding. It is put into a latent dimension and then used by the PPO Network to calculate the action as a distribution as well as the value.

`observe()` function. The autors explicitly calculate the neighborhood for each agent that uses the policy.

For testing and debugging purposes, the Graph Neural Network (GNN) can be disabled. When it is, only the node-features are used. Once the observation graph is constructed, the node-features are encoded into a latent-space representation. For this embedding we use a linear layer. We allow independet embedding for the actor and critic. They can either share the same embedding or have different embeddings. The Multi-Layer Perceptron (MLP) used in Ruede et al. (2021) is used as an encoding step and to process each element of an aggregation group. Processing is not needed for our embedding, as our GNN is responsible for that, so a simple linear transformation is enough.

The learning head for Proximal Policy Optimization (PPO) is composed of the actor describing the policy π, which gets us the joint action $\text{act} = \langle \text{act}_1, ..., \text{act}_n \rangle$ and the critic calculating the running advantage A. Our actor uses an MLP, followed by another linear transformation. It will reduce the dimension from the latent dimension back to the action dimension. This is then used as the mean μ for a *Diagonal Gaussian distribution* to support non deterministic policies. The variance σ^2 is a learnable parameter, but not conditioned on any input. The critic also uses an MLP with an output-dimension of 1. This defines a value per node in the graph. In Ruede et al. (2021) the authors define a value for each agent. This helps to identify which agent was responsible for the reward. But this is not how the PPO Algorithm was originally designed to work for single-agent environments. Our architecture can use an aggregation to output a single value per graph. This can either be $\min(x)$, $\max(x)$, or $\text{mean}(x)$. Through exploration the agents are still able to identify over time how much an agent was responsible for a given reward. We parameterized node-wise values and graph-wise values for our approach.

4.2. Homogeneous Message-passing GNN Architecture

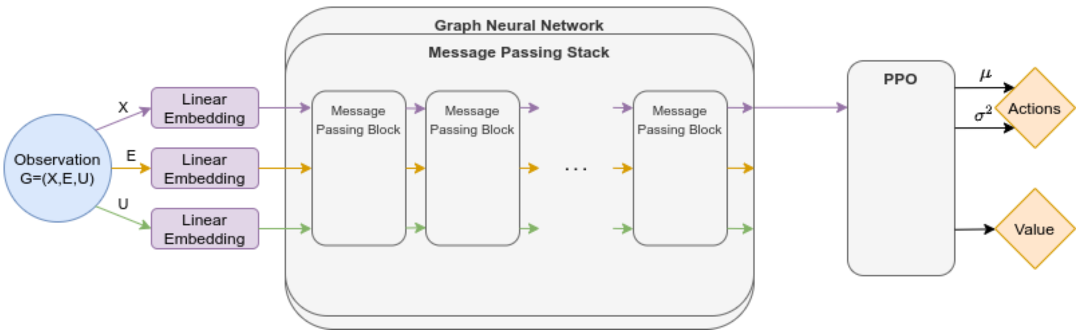


Figure 4.2.: Here a GNN is added as a step before the policy architecture. The entire observation is put into a latent space and then passed through multiple message passing hops. The outputs of one hop is used as the input of the next hop. This is coordinated through the message passing stack.

An overview of the message passing architecture for homogeneous graphs as described below can be seen in Figure 4.2.

When enabling the GNN, it is added as a step between the policy architecture and observation. Node-features, edge-features and global-features are now embedded into a latent-space linearly, similar to above. The weights between the encoders are not shared. Global-features can be used to explicitly give the swarm global information. In our experiments global features were not used.

We chose to use a GNN structure that uses stacks of GNN blocks. Each block defines a single message-passing hop. The stack is an array of blocks that are used sequentially. This way we are able to support multiple GNN hops. We do want to note, that Ruede et al. (2021) model architecture for a single aggregation group can be described via the function:

$$f(a_i) = \text{decoder}(\text{selfObserve}(a_i), \bigoplus_{j \in \mathcal{N}_i} \text{encoder}(\text{observe}(a_i, a_j)))$$

As can be seen, this already implements a common message-passing GNN. In comparison, our architecture adds the ability to process multiple GNN hops. The GNN block is composed of an edge-, node-, and global-module, each responsible for the corresponding features. A diagram of the relationship between these blocks is outlined in Figure 4.3. The functions used by the modules are as following:

$$\begin{aligned}\text{Edge-Module: } x_{e'} &= f_e(x_v, x_u, x_e, x_g)\\ \text{Node-Module: } x_{v'} &= f_v(x_v, \oplus_{\{e'=(v,u)||_v\}} x_{e'}, x_g)\\ \text{Global-Module: } x_{g'} &= f_g(\oplus_V x_{v'}, \oplus_E x_{e'}, x_g)\end{aligned}$$

Similar to the embedding, our architecture allows two seperate GNN for the value and critic. Each of them have seperate embedding and a seperate GNN. This allows both to process observations differently. Each of the aggregation functions used in the entire network are shared and can be set to $\min(x)$, $\max(x)$, or $\text{mean}(x)$. Furthermore the input of a GNN block can be added to the output. These residual connections help with the vanishing/exploding gradients problems by breaking up the multiplications.

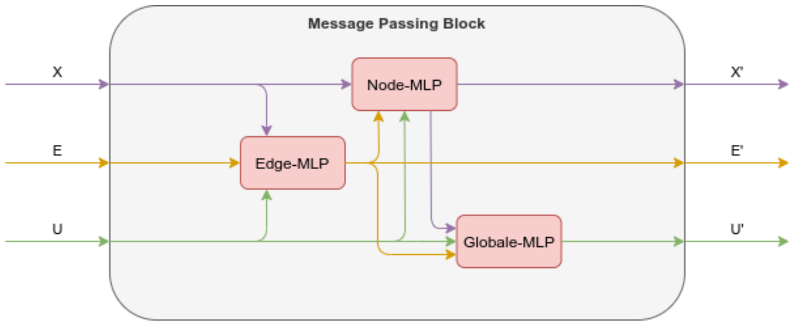


Figure 4.3.: Detailed view of one message passing block. It uses a edge-, node- and global-module to compute a single message passing hop.

4.3. Heterogeneous Message-passing GNN Architecture

An overview of the message passing architecture for heterogeneous graphs as described below can be seen in Figure 4.4.

This extension is used when an environment requires multiple distinct information types. By using heterogeneous graphs we are able to support multiple aggregation groups as in Ruede et al. (2021). An heterogeneous graph $G = (X, E, U)$ is composed of multiple node-types $X_i,\ i = 1, \dots, m$ where $X = \bigcup_{i=1}^{m} X_i$ and multiple edge-types $E_i,\ i = 1, \dots, m^2$ where $E = \bigcup_{i=1}^{m^2} E_i$. An edge-type is uniquely identified by its source node-type and destination node-type where $\forall E_i : \exists j, k \leq m : E_i = (X_j \times X_k)$. If the source- and destination-node-types are not the same i.e. $j \neq k$, we can either define the edge-types as directed, or make it undirected. In the undirected case not all node-type combinations are edge-types, because $(X_j \times X_k) = (X_k \times X_j)$. Here the edge-type of $(X_j \times X_k)$ is sufficient. In the directed case $(X_j \times X_k) \neq (X_k \times X_j)$ and both types are needed. For experiments, both variations are parameterized in our architecture. If $j = k$ the edge-type can be directed or undirected, which depends on the culling method used.

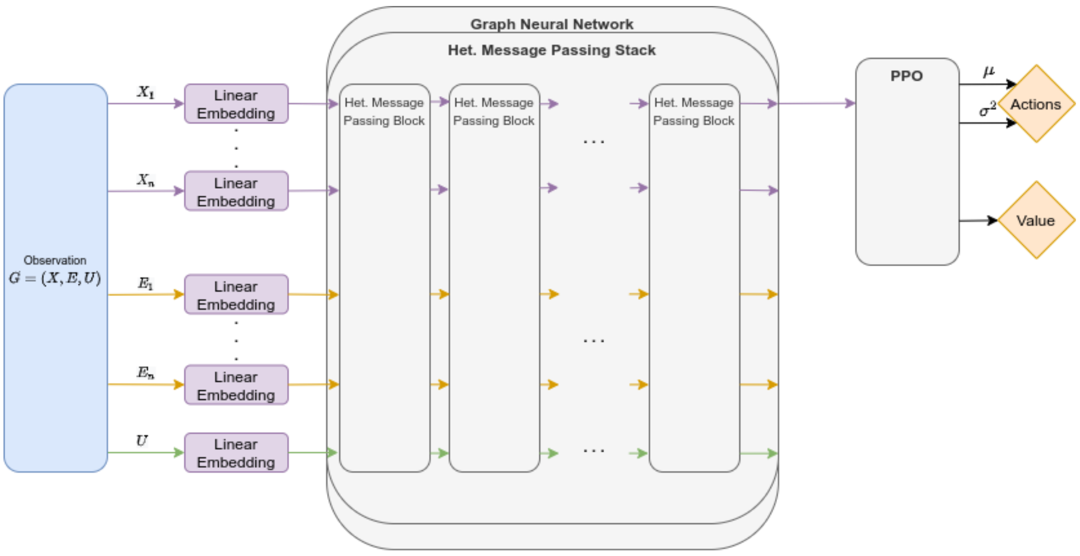


Figure 4.4.: An overview of the heterogeneous version of the architecture. The overall structure stays the same, but the input observation is composed of mulitple node-types and edge-types. Each of the input embeddings per graph is unique. The policy architecture only uses one of the note types as input.

Each of the node-types and edge-types are linearly embedded. Each type has it's own linear transformation. As different types define completely aggregation groups, weight sharing is not used. Global features are also embedded independently. Structurely the heterogeneous GNN stacks and blocks are the same, they only have more inputs compared to the homogeneous counterpart. The underlying heterogeneous edge-, node- and global-modules can be described by following functions:

$$\begin{aligned}
\text{Edge-Module: } & x'_{e_i} = f_{e_i}(x_v, x_u, x_{e_i}, x_g), e_i = (u,v) \\
\text{Node-Module: } & x'_{v'_i} = f_{v_i}(x_{v_i}, [\otimes_j \oplus_{\{e_j=(v_i,u)\}} x'_{e_j}], x_g) \\
\text{Global-Module: } & x'_g = f_g([\otimes_j \oplus_{V_j} x'_{v \in V_j}], [\otimes_k \oplus_{E_k} x'_{e \in E_k}], x_g)
\end{aligned}$$

In addition to the options outlined in the homogeneous GNN section, we are able to define different functions for $\otimes$ and $\oplus$. Aggregator $\oplus$ follows the global aggregation function setting shared by the network, while $\otimes$ on the other hand can be defined as a concatenation or as the global aggregation function. Ruede et al. (2021) only uses a concatenation. The concatenation is more expensive as the input dimension of the MLP of the node-, or global-module is larger. Though it is more expressive than the alternative of using aggregation. In tasks like multi evader pursuit, if the different node-types are aggregated, it is harder for agents to distinguish stimuli from other agents or from the evaders. Both heterogeneous aggregation types are parameterized in the architecture

Chapter 5.

Experiments

5.1. General Setup

The following experiments were written to work with the DAVIS project Freymuth (2021). DAVIS already included the core structure for Multi-Agent Reinforcement Learning research with support for graph observations. Furthermore it simplified the recording of key metrics and training with a cluster. Training was done via the BWUniCluster2 by the state of Baden-Württemberg through bwHPC. We adapted the DAVIS project to work with heterogeneous observation graphs and implemented the tasks needed for our experiments.

All the experiments use, if not otherwise stated, Proximal Policy Optimization (PPO) Schulman et al. (2017) with additional code level optimizations from Engstrom et al. (2020). Additionaly we implemented the ability for the actor and critic to use different Graph Neural Networks or use the same. In the former case the actor and critic are able to learn different graph networks to suit their needs. If not stated otherwise, actor and critic use different GNNs.

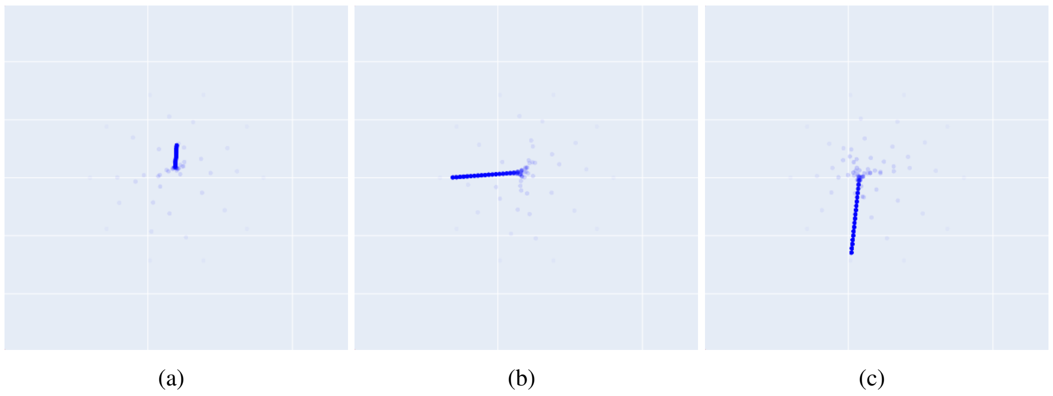


Figure 5.1.: Three example episodes for Rendezvous. The circular start position used for evaluation can be seen. The agents are able to meed in a couple of steps. Then they move a little as a group.

5.2. Tasks

5.2.1. Rendezvous

The goal of the Rendezvous task, is for n agents to converge onto a single point. An example episode can be seen in Figure 5.1.

The environment can be configured as a torus (position is wrapped using modulo) or as an rectangular world with borders (position is clipped). Positions are using floating-point precision. It terminates after a given amount of timesteps. The agents are dots without collisions. They use a direct dynamic model, therefore the two actions they can perform represent movement in the x-Axis and y-Axis respectively. The reward function r consists of two terms. First we use the mean of the normalized pairwise distances between the agents as a distance penalty d_p. Secondly we use an velocity penalty v_p, that scales squared to the mean of the velocity v:

$$\begin{aligned} v_p &= \text{mean}(v^2) \\ d_p &= \text{mean}(\frac{\text{pairwise-distances}}{\text{worldsize}}) \\ r &= v_p + d_p \end{aligned}$$

A culling method is used, so that the agents have a finite sensor range, either kNN or euclidean distance. The observation graph is homogeneous and composed of the following aspects:

- node features:

   1. agent positions, normalized to world size (for debugging, otherwise constant placeholder)
- edge features:
   1. pairwise agent distances, normalized to world size
- global features: None

5.2.2. Dispersion

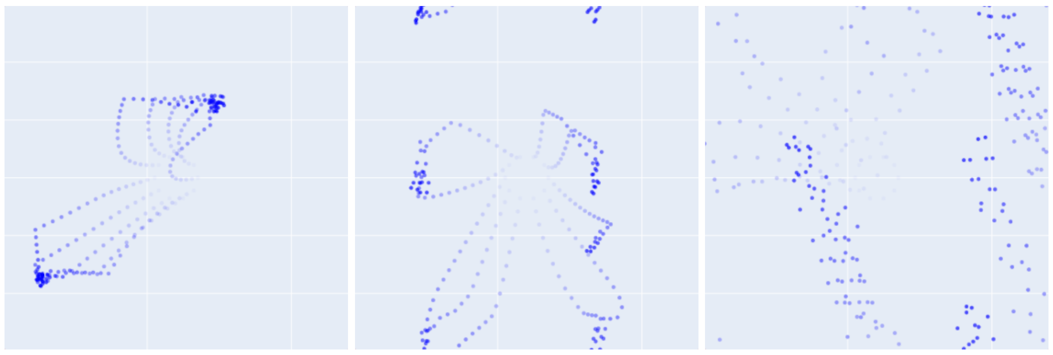
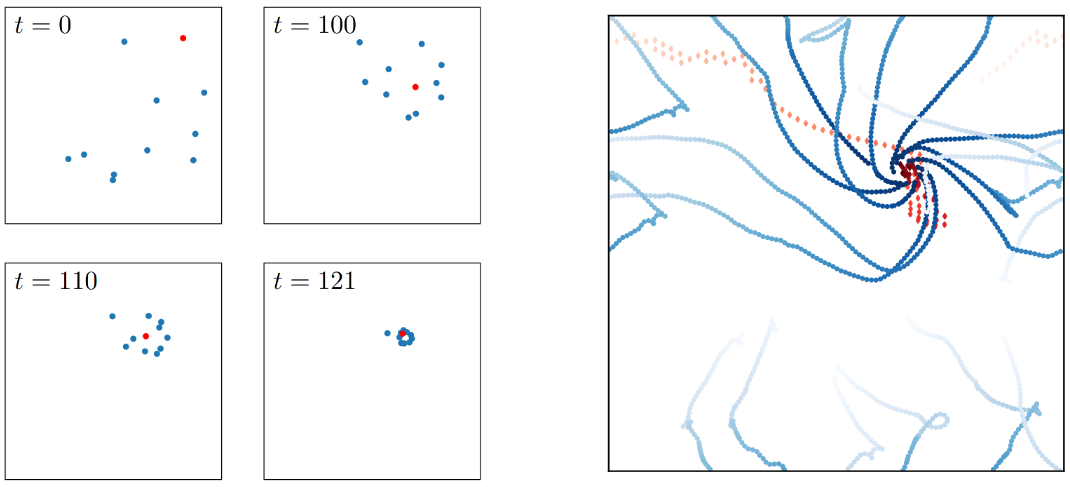

(a) $f(x) = \min(x)$ (b) $f(x) = x^2$ (c) $f(x) = \sqrt(x)$

Figure 5.2.: Three example episodes for Dispersion. The circular start position used for evaluation can be seen. The agents try to disperse and organize into distinct groups. The examples used different reward functions.

In a Dispersion task, the agents should try to maximize the distance between each other. An example episode can be seen in Figure 5.2.

This environment is a variation of Rendezvous and therefore shares most of it's properties. The main difference lies within the reward function r. Our implementation allows for different reward calculations using functions $f(x)$ on the reward, before calculating the mean. Supported functions are x, x^2, $\sqrt{x}$, $\min(x)$, $\max(x)$

$$\begin{aligned} a_p &= \text{mean}(a^2) \\ d_p &= \text{mean}(\frac{f(\text{pairwise-distances})}{\text{worldsize}}) \\ r &= a_p + d_p \end{aligned}$$



(a)

Figure 5.3.: An example for a successfull Single Evader Pursuit episode, taken from Hüttenrauch et al. (2019)

5.2.3. Single Evader Pursuit

For Single Evader Pursuit the agents have to catch a single evader. Given that the evader has a higher velocity than the agents, they have to cooperate. An example episode can be seen in Figure 5.3.

The environment can be configured as a torus (position is wrapped using modulo) or as an rectangular world with borders (position is clipped). Positions are using floating-point precision. It terminates after a given amount of timesteps or when the single evader has been caught. A catch is triggered when one agent is less than 1% of world size away from the evader. The agents are collisionless dots. Like Rendezvous a direct dynamic model is used. The evader is part of the environment and will not learn. It's dynamic is either a simple linear movement or uses Voronoi-regions which is based on Zhou et al. (2016). The minimum normalized distance of the agents to the evader is used for the distance penalty d_p and the velocity penalty v_p is the same as Rendezvous:

$$
\begin{aligned}
v_p &= \text{mean}(v^2) \\
d_p &= \text{mean}(\frac{\text{agent-evader-distances}}{\text{worldsize}}) \\
r &= v_p + d_p
\end{aligned}
$$

Single Evader Pursuit also supports the same culling methods as Rendezvous. The observation graph is heterogeneous consists of the following aspects:

- agent node features:
  1. agent positions, normalized to world size (for debugging, otherwise constant placeholder)
- evader node features:
  1. evader positions, normalized to world size (for debugging, otherwise constant placeholder)
- agent-to-agent edge features:
  1. pairwise agent distances, normalized to world size
- agent-to-evader edge features:
  1. agent to evader distances, normalized to world size
- global features: None

5.2.4. Multi Evader Pursuit

In Multi Evader Pursuit the agents have to catch multiple evaders.

This task is largely the same as single evader pursuit. The catch threshhold is changed to less than 2% of world size. The reward uses the established velocity penalty v_p and a catch count c. It will reward the agents with +1 every time they catch an evader:

$$\begin{aligned} v_p &= \text{mean}(v^2) \\ r &= v_p + c \end{aligned}$$

Chapter 6.

Evaluation

This section elaborates on the results of our experiments. As a proof of concept, we aimed to show, that our architecture is able to learn rendezvous tasks with a more or less GNN Blocks (1). Then we compared training on a set of agents, while evaluating on a different number of agents (2). Afterwards we showed the effect of different aggregation functions on training (3), and how multiple GNN hops can affect the result if the observation radius for the agent varies (4).

Evaluation of the results used different environments compared to training. While the agents trained on randomized starting positions, they were tested with fixed starting positions to allow comparable results. A similar setting was used for evader starting position in Pursuit environments. We considered the average reward per training step and the last reward per training step as performance metrics. The mean gave us a meaningful metric for the progress of the RL algorithm, while the last reward had a more geometric importance for solving tasks like Rendezvous.

A more detailed overview of the PPO parameters we use can be found as an appendix Appendix A. Unless mentioned otherwise, the following experimental settings were used:

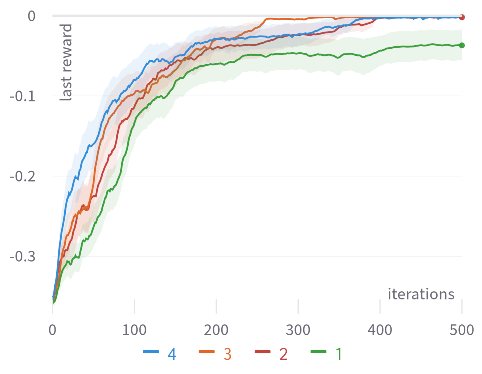
- The plots show the standard error from the input data.
- The environment always used a torus setup (position is wrapped using modulo, no borders).
- Observations did not directly include the positions of agents or evaders which would trivialize the tasks.
- No observation culling was used: Each agent could see every other agent and evader.

- The agents used direct dynamics: Their actions directly influenced movement on the x- and y-Axis.
- Rewards and Observation were normalized using a running mean and std to improve learning.
- Used mean aggregation.
- Independet GNN-stack for the actor and critic, which used residual connections.

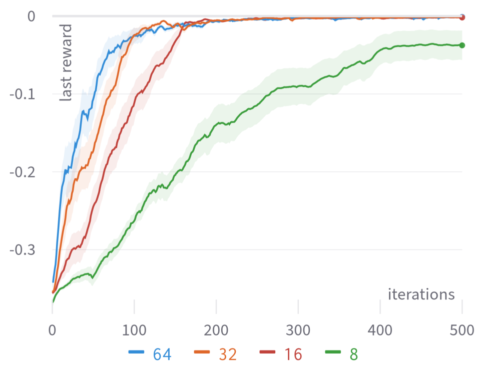
During the evaluation of the experiments, we experienced unexpected hardware issues that caused a subset of our runs to crash. Due to time constraints, we were not able to fully replicate the respective experiments. Note that we do not include these crashed runs in our evaluation, and that the resulting grid searches are thus partially incomplete. The general findings of this thesis remain unchanged.

6.1. Proof of Concept

This proof-of-concept experiment used rendezvous. We wanted to show that our architecture works in general. Furthermore, we wanted to investigate if multiple GNN hops can already have an impact on learning for rendezvous without any observation restrictions. We chose to use 10 agents that have 24 timesteps to meet each other in a torus-world of size 12. For these runs we used a latent dimension in the range of 8 to 64 and 1 to 4 number of hops. The last reward of a given iteration tells us if the agents were able to meet up in the allotted time and therefore we choose that as a metric.



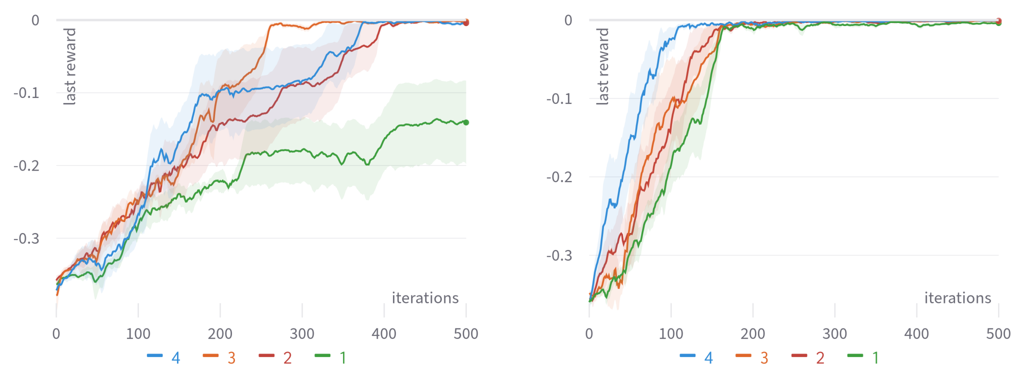
(a) Number of Hops



(b) Latent Dimension

Figure 6.1.: Results of the Rendezvous task for different latent dimensions. The left side shows the different number of hops, while the right side shows different latent dimensions.

Figure 6.1 clearly shows that most of the training environments were able to successfully solve the task. With at least 2 hops, the task was solved and no extra benefit can be seen for more than 2 hops. Earlier in training, the performance shows some difference for more hops. It is clear that the architecture is able to use the observation input more effectively and learn faster. Each extra hop added roughly 13% to the training time needed. If we look at the right site, we can see the effect of different latent dimension to the effectiveness of learning. Only a latent dimension of 8 was not enough for the agents to meet up in the allotted time. In that case, it is clear that the architecture was not able to correctly represent and process observation data for 10 agents.



(a) Latent Dimension: 8 (b) Latent Dimension: 16

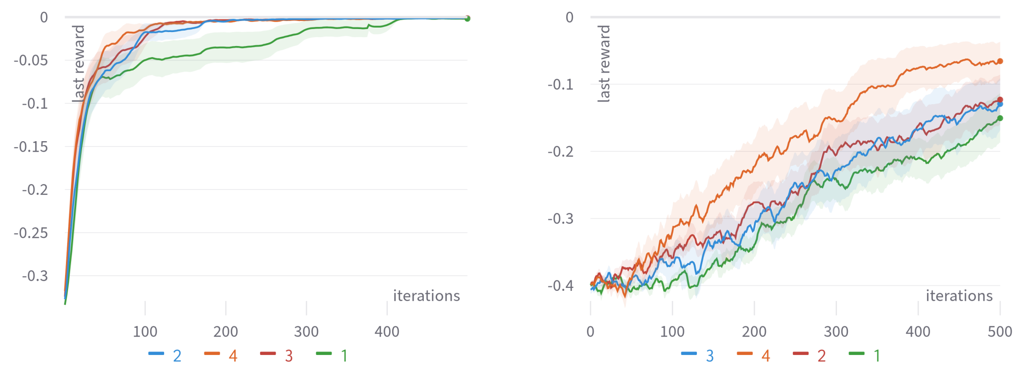
Figure 6.2.: Here we filtered the results for a latent dimension of 8 on the left and 16 on the right.

If we investigate the behavior for the different latent dimensions, we can see a large difference in performance shown in Figure 6.2. For a latent dimension of at least 16, the difference between the number of hops is quite small, as can be seen on the right side. If we filter the results for a latent dimension of 8, the general performance is expectedly worse. But the difference between the number of GNN hops is much more pronounced. This shows that, in larger constraints, the ability to process more of the observation data leads to a larger gain. As doubling the latent dimension also roughly doubles the total time needed, we conclude that more hops can be more efficient than a higher latent dimension.

6.2. Ablation

In these experiments we want to examine how good our architecture was in ablation tests. In other words, how good it would be able to abstract the task at hand, while using a different amount of agents to train than to evaluate. Overall, we chose to use the same number of timesteps and the same torus-size then the proof-of-concept. We used a latent dimension in the range of 8 to 32 and 1 to 4 number of hops. We trained on 5,10, or 20 agents and evaluated on 5,10, or 20 agents.

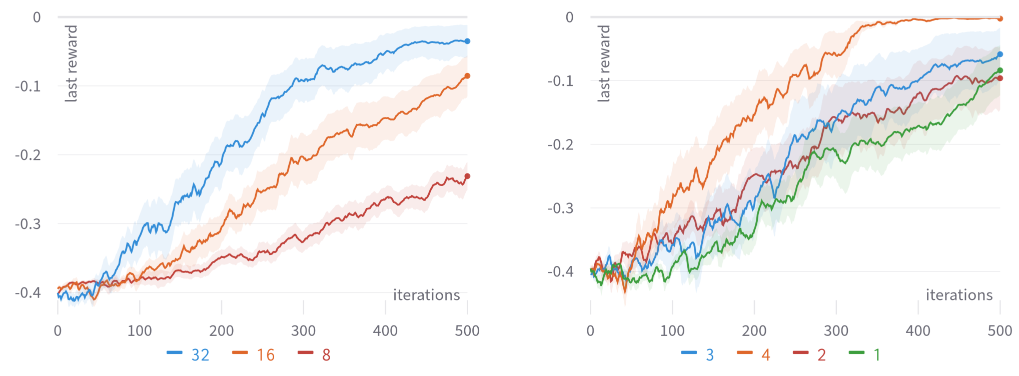
Figure 6.3 examines the two extreme cases of our ablation test. Learning on 10 agents did not prove any interesting results. On the left side we see no discernable difference between 2 to 4



(a) Training: 5 Agents, Evaluation: 20 Agents (b) Training: 20 Agents, Evaluation: 5 Agents

Figure 6.3.: Performance of different amount of GNN hops for ablation. The leftside shows successfull training with 5 agents and evaluation with 20 agents. Training with 20 agents was more difficult

GNN Hops. All of them were able to solve the task in roughly the same time. A single hop is shown as an outlier, which performed considerably worse. It took more than double the time to solve the task, compared to more hops. No real benefit for more than a single hop can be found for an upwards ablation of 5 training agents and 20 evaluation agents. The right side shows the downward ablation of 20 training agents and 5 evaluation agents. Given, that this is a harder task to learn, we can clearly see that the solutions still led to a gap between the agents. It therefore did not completely solve Rendezvous. Though noisy, 2-3 hops performed better overall than a single hop. However, a sizeable difference for 4 hops can be observed. This holds true, if we filter the results for 20 training agents and 20 evaluation agents. This might indicate, that the architecture with more hops is better suited for larger amounts of agents.



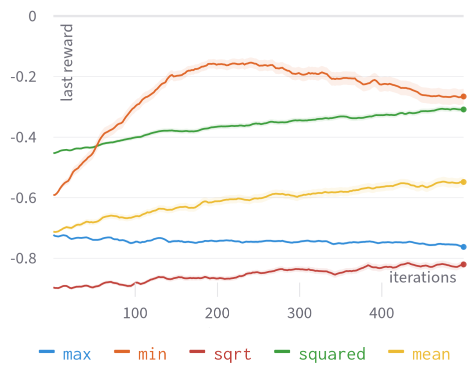
(a) Latent Dimension (b) Number of Hops without Latent Dimension 8

Figure 6.4.: Looking at the performance for the latent dimension shows a larger difference in the results on the leftside. Filtering out latent dimension of 8 shows a much stronger performance for 4 hops.

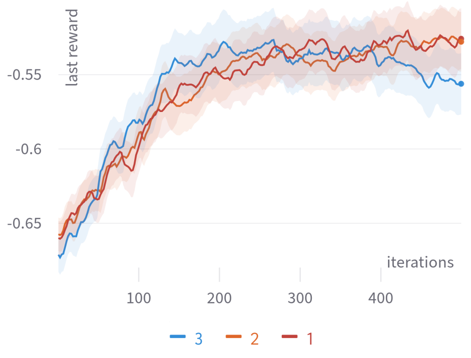
We wanted to examine the results for 20 training agents and 5 learning agents a bit more. If we take a look at the performance of different latent dimensions on the left side of Figure 6.4, we are able to see that a latent dimension of 8 clearly lags behind. Filtering for latent dimension 8, which is not shown here, shows us very similar results to Figure 6.2 in the proof of concept experiments. Because of this, we wanted to exclude latent dimension of 8 in the results. The right side clearly demonstrates a more pronounced difference for 4 GNN hops. It is the only setup, that was able to solve the task. With a singular constraint, the downsizing of information gathered by agents and the resulting information sparseness, the results were much more pronounced.

6.3. Different Aggregation Functions

For these experiments, we looked at the dispersion environment and different aggregation functions used by the architecture. The goal was, to examine the effect of aggregation functions to the learning effectiveness, under different reward types. Dispersion used very similar parameters to Rendezvous. We used a latent dimension in the range of 8 to 32 and 1 to 3 number of hops.



(a) Different Rewardtypes



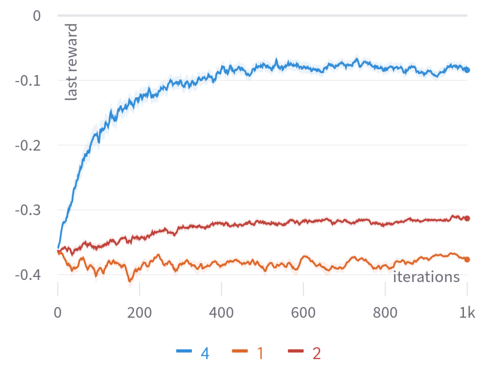
(b) Different Number of Hops

Figure 6.5.: Performance of different aggregation functions for the Dispersion task

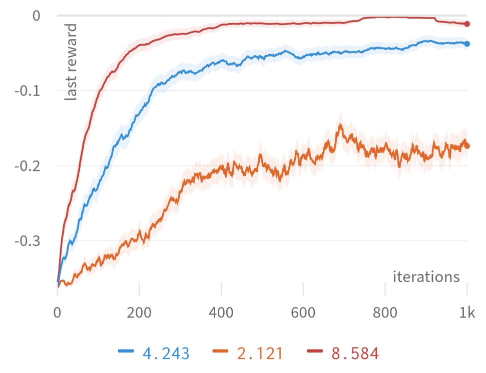
Figure 6.5 shows our results for dispersion. We can generally see, that the architecture was not able to learn for reward types mean, min and max. Furthermore, it shows that the number of hops did not make any difference in learning. If we filter for the different aggregation functions, we either observed no difference in the number of hops, or the setup did not learn at all. If we filter out mean, min and max, the results show no difference. This suggests that dispersion is either an outlier, not properly implemented, or shows that the GNN hops may not help in certain tasks.

6.4. Under Observation Constraints

The goal for these experiments was to determine, if the GNN hops had a larger impact, when we use a limited observation range for our agents. For this we looked at kNN and Euclidean distance culling of 50, 25 and 12.5% respectively. In the case of Euclidean distance, the percentage is relative to the world size. For kNN it denotes how many agents of the total number are visible. Tests were run from 8 to 64 latent dimension and 1 to 4 GNN hops.

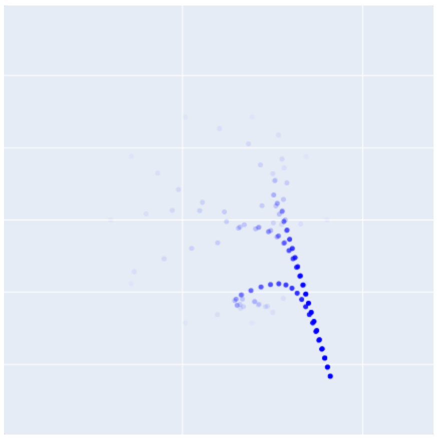


(a) kNN

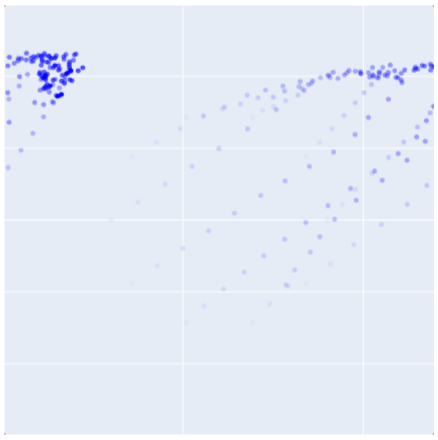


(b) Euclidean Distance

Figure 6.6.: Looking at the performance under different observation ranges. The left side shows different number of neighboring agents and the right side shows different observation distances.



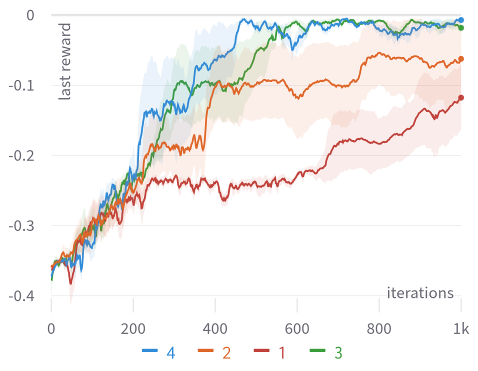
(a) kNN, 50%



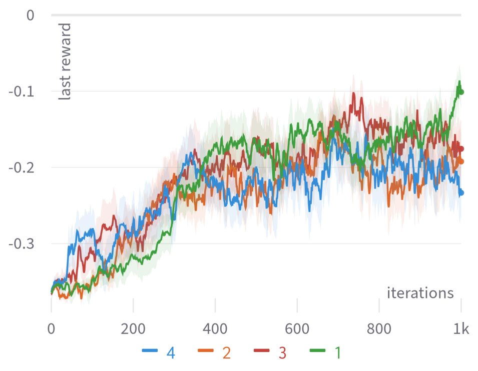
(b) Euclidean, 12.5%

Figure 6.7.: Two example episodes.

The first Figure 6.6 shows that in general, the architecture struggled a lot with learning kNN culling. The Euclidean distance is symmetrical, if a sees b then b sees a. But kNN is highly asymmetrical. This leads to a directed graph between agents. So, information flow is very restricted and even locally incomplete. Some observations might never reach some of the agents. The example episodes shown in Figure 6.7 clearly demonstrate this behavior. It demonstrates how the agents converge on local groups first, followed by each group trying to find the other groups and converging globally. This breaks for small k in kNN. For 12.5% and 25% culling in kNN, no learning was achieved.



(a) 25 % Distance, Latent Dimension of 8



(b) 12.5 % Distance

Figure 6.8.: Comparison of 25% and 12.5 % distance culling for different number of GNN hops.

For distance culling, 50% was generally not different than to our proof-of-concept. This might be, because of our specific evaluation setup. The agents always spawn in a circle, with the radius being half the world size. 50% is just enough so that each agent can see each other at the start of the environment. In Figure 6.8 we examined distance culling in a little bit more detail. The right side shows that the number of hops did not make a difference, as the architecture struggled to learn under these constraints. For 25%, which is shown on the left side, we observed larger differences between the number of hops for a latent dimension of 8. For higher latent dimension this was still the case, but not as pronounced. This seems to suggest that, if the constraints are even more severe and the task itself is solvable for the number of iterations provided, the gain for multiple GNN Hops is large. Though with more than 3 hops, no real benefit is shown.

6.5. Pursuit Results

We originally intended to also run our experiments on the Pursuit environment. This included observation tests and examining both heterogeneous aggregation types. We hoped, that a more complex task could lead to even more pronounced evidence for the advantage of multiple GNN Hops. Unfortunately, due to the aforementioned hardware issues and time constraints, we were not able to produce adequate results. We found, that even simple Single-Evader Pursuit test cases oscillated heavily. We tried running multiple experiments with increasing difficulty. Our investigations showed, that when the evader has a randomized starting position, the experiments break. Even with extensive debugging, we were not able to find a logical or mathematical error in either the environment itself, or the heterogeneous graphs. We additionally tried to extend our architecture by having one MLP per Node-, or Edge-Type in the heterogeneous modules. This showed a more stable solution in the resulting policy, which still did not learn the task at hand. At the point of writing, it is unclear if the problem exists within our implementation of the environment, or the heterogeneous GNN.

Chapter 7.

Conclusion and Future Work

7.1. Conclusion

Generally, we have shown the overall validity of our approach and architecture. Furthermore, under different constraints multiple homogeneous GNN hops can improve upon the results. Firstly, when having a small latent dimension. Here, more hops did help and were more efficient than increasing the latent dimension. This suggests a higher observation efficiency than without using GNNs. Secondly, in the ablation tests, the benefit was also much higher. In observation constrained environment the architecture performed better, if the setup was reasonable solvable. If it wasn't, more hops did not improve the result. Here in low latent dimension and low observation radius, the benefit was most pronounced. Which leads to the conclusion that the high processing capabilities of mulitple hops did improve upon learning.

7.2. Future Work

More can be done to expand on the work already finished in this bachelor thesis.

Firstly, while our results showed promising results, additional experiments would be needed to show the limits and real benefits of our approach. Some of the experiments may produce different results if they ran longer, like a latent dimension of 8. Also, tighter observation radii could use more iterations. Especially kNN did not produce real results. Furthermore, some of the parameters our architecture is capable of we were not able to test. This includes the different neighbor aggregation types for heterogeneous graphs, or the node-wise values and graph-wise value functions. Likewise working results for Pursuit are needed to show our architecture in the heterogeneous case.

All of the experiments in this paper only considered using Proximal Policy Optimization (PPO) Schulman et al. (2017), as it is a very common baseline training algorithm. However, recent research shows, that other methods might lead to better results for Multi-Agent Reinforcement Learning. Specifically Trust Region Layers (PG-TRL) Otto et al. (2021) is an alternative to PPO. The results show that it is at least on par with PPO, while requiring less code-level optimizations. It is noted that in experiments with sparser rewards, the better exploration of TRL improves the results significantly. Ruede et al. (2021) explores Trust Region Layers (PG-TRL) (Otto et al., 2021) for multi-agent tasks. The author explains, that given a multi-agent cooperative task, the agents are rewarded as a group, which makes the reward more sparse. Therefore, creating correlation of a single agent's action and the group reward is harder. The hyper parameters used for TRL were based on searches for PPO and no extensive testing for TRL was done. Even then TRL was able to perform similar to PPO.
Creating further experiments based on the architecture established in this thesis would very likely benefit from TRL.

Furthermore Ruede et al. (2021) also used more complex multi-agent task than we used in our experiments. In Box Clustering there are agents and multiple boxes. Each box is assigned to one cluster. The goal is to move the boxes, so that the distance between the boxes in each cluster is minimal. Optimal solutions will require that the agents work together to move the boxes and that they split the work between them. In his thesis his approach worked well for two clusters of boxes but fell apart with three clusters. Then the agents were only able to move one cluster correctly. Only after increasing the batch size and environment steps per training steps tenfold, the agents were able to consider more than 2 clusters. As explained above, our approach is structurally similar to Ruede et al. (2021) as both can be described with message-passing of GNNs. It was shown that more complex tasks, especially with tight communication ranges, benefited hugely from multiple message passing hops. So, one can assume that we would be able to solve Box Clustering better.

As this thesis is an extension of basic ideas found in Ruede et al. (2021), both thesis are designed to work for a single group of homogeneous agents. It is stated in the architecture description, that in heterogeneous graphs the policy training method can only use one node type. It is always trained on the agent node features. It would be possible to parameterize the node type it trains on. In team-based tasks you can have multiple competing groups of agents. Our architecture could support two policies that can be trained on different node types and therefore groups of agents.

7.3. Acknowledgements

The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

Bibliography

M. Abdoos, N. Mozayani, and A. L. Bazzan. Holonic multi-agent system for traffic signals control. *Engineering Applications of Artificial Intelligence*, 26(5):1575–1587, 2013. ISSN 0952-1976. doi: https://doi.org/10.1016/j.engappai.2013.01.007. URL `https://www.sciencedirect.com/science/article/pii/S0952197613000171`.

M. Aurangzeb, F. L. Lewis, and M. Huber. Efficient, swarm-based path finding in unknown graphs using reinforcement learning. In *2013 10th IEEE International Conference on Control and Automation (ICCA)*, pages 870–877, 2013. doi: 10.1109/ICCA.2013.6564940.

B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch. Emergent Tool Use From Multi-Agent Autocurricula. *arXiv e-prints*, art. arXiv:1909.07528, Sept. 2019.

T. Chu, J. Wang, L. Codecà, and Z. Li. Multi-agent deep reinforcement learning for large-scale traffic signal control. *IEEE Transactions on Intelligent Transportation Systems*, 21(3): 1086–1095, 2020. doi: 10.1109/TITS.2019.2901791.

L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry. Implementation matters in deep policy gradients: A case study on PPO and TRPO. *CoRR*, abs/2005.12729, 2020. URL `https://arxiv.org/abs/2005.12729`.

N. Freymuth. Davis project. Part of the research of the ALR research group at the Karlsruher Institute of Technology (KIT), 2021.

M. Hüttenrauch, A. Šošić, and G. Neumann. Deep reinforcement learning for swarm systems. *Journal of Machine Learning Research*, 20(54):1–31, 2019. URL `http://jmlr.org/papers/v20/18-476.html`.

Y. Ishiwaka, T. Sato, and Y. Kakazu. An approach to the pursuit problem on a heterogeneous multiagent system using reinforcement learning. *Robotics and Autonomous Systems*, 43(4):245–256, 2003. ISSN 0921-8890. doi: https://doi.org/10.1016/S0921-8890(03)00040-X. URL `https://www.sciencedirect.com/science/article/pii/S092188900300040X`.

J. Jiang, C. Dun, and Z. Lu. Graph convolutional reinforcement learning for multi-agent cooperation. *CoRR*, abs/1810.09202, 2018. URL `http://arxiv.org/abs/1810.09202`.

A. Koestler. The ghost in the machine. 1968.

Y. Liu, W. Wang, Y. Hu, J. Hao, X. Chen, and Y. Gao. Multi-agent game abstraction via graph attention neural network. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):7211–7218, Apr. 2020. doi: 10.1609/aaai.v34i05.6211. URL `https://ojs.aaai.org/index.php/AAAI/article/view/6211`.

F. Otto, P. Becker, V. A. Ngo, H. C. M. Ziesche, and G. Neumann. Differentiable trust region layers for deep reinforcement learning. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=qYZD-AO1Vn`.

R. Ruede, G. Neumann, T. Asfour, and M. Huettenrauch. Bayesian and attentive aggregation for multi-agent deep reinforcement learning. *Autonomous Learning Robotics (ALR)*, 2021. URL `https://phiresky.github.io/masters-thesis/manuscript.pdf`.

A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. Battaglia. Learning to simulate complex physics with graph networks. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 8459–8468. PMLR, 13–18 Jul 2020. URL `https://proceedings.mlr.press/v119/sanchez-gonzalez20a.html`.

J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR. URL `https://proceedings.mlr.press/v37/schulman15.html`.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL `http://arxiv.org/abs/1707.06347`.

A. Servin and D. Kudenko. Multi-agent reinforcement learning for intrusion detection. In K. Tuyls, A. Nowe, Z. Guessoum, and D. Kudenko, editors, *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*, pages 211–223, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-77949-0.

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks, 2018.

F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger. Simplifying graph convolutional networks. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6861–6871. PMLR, 09–15 Jun 2019. URL `https://proceedings.mlr.press/v97/wu19e.html`.

M. Yogeswaran, S. Ponnambalam, and G. Kanagaraj. Reinforcement learning in swarm-robotics for multi-agent foraging-task domain. In *2013 IEEE Symposium on Swarm Intelligence (SIS)*, pages 15–21, 2013. doi: 10.1109/SIS.2013.6615154.

L. Zheng, J. Yang, H. Cai, M. Zhou, W. Zhang, J. Wang, and Y. Yu. Magent: A many-agent reinforcement learning platform for artificial collective intelligence. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. URL `https://ojs.aaai.org/index.php/AAAI/article/view/11371`.

W. J. Zhou, B. Subagdja, A.-H. Tan, and D. W.-S. Ong. Hierarchical control of multi-agent reinforcement learning team in real-time strategy (rts) games. *Expert Systems with Applications*, 186:115707, 2021. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2021.115707. URL `https://www.sciencedirect.com/science/article/pii/S0957417421010897`.

Z. Zhou, W. Zhang, J. Ding, H. Huang, D. M. Stipanović, and C. J. Tomlin. Cooperative pursuit with voronoi partitions. *Automatica*, 72:64–72, 2016. ISSN 0005-1098. doi: https://doi.org/10.1016/j.automatica.2016.05.007. URL `https://www.sciencedirect.com/science/article/pii/S0005109816301911`.

Appendix A.

Experiment Hyperparameters

Table A.1.: General PPO parameters (Schulman et al., 2017) and code optimizations (Engstrom et al., 2020)

Parameter	Value
batch size	256
number of rollout steps	2048
discount factor	0.99
epochs per iteration	5
entropy coefficient	0.0
value function coefficient	0.5
λ_{gae}	0.95
learning rate	$3.0e^{-4}$
l_2 norm	0.0
max gradient norm	0.5
value function clip range	0.2
clip range	0.2
reward normalization	True
observation normalization	True
MLP layers	2
MLP activation function	tanh()