Exploring Deep Learning Methods for Embedding Agent States in Multi-Agent Reinforcement Learning

---

Simon Hampp

September 12, 2025

Department of Computer Science
and Mathematics
Munich University of Applied Sciences

Master's Thesis

Exploring Deep Learning Methods for Embedding Agent States in Multi-Agent Reinforcement Learning

Simon Hampp

ID: 06808023
Focus: VCML

Reviewer Prof. Dr. David Spieler
Department of Computer Science and Mathematics
Munich University of Applied Sciences

2. Reviewer Prof. Dr. Markus Friedrich
Department of Computer Science and Mathematics
Munich University of Applied Sciences

September 12, 2025

Simon Hampp
Exploring Deep Learning Methods for Embedding Agent States in Multi-Agent Reinforcement Learning
Master's Thesis, September 12, 2025
Reviewers: Prof. Dr. David Spieler and Prof. Dr. Markus Friedrich
Focus of studies: Visual Computing and Machine Learning
Immatriculation Number: 06808023

Munich University of Applied Sciences
Department of Computer Science and Mathematics
Lothstr. 34
80335 München

Abstract

This thesis investigates permutation-invariant and agent-count invariant methods for centralized critics in multi-agent reinforcement learning (MARL). Conventional approaches typically concatenate all agent observations to form the critic's input. A simple strategy, but neither permutation-invariant nor adaptable to variable agent numbers. As a result, such critics are unsuitable for dynamic environments with changing agent counts and do not exploit the potential sample efficiency gains of Permutation Invariant (PI) critics. To address these limitations, critic architectures were developed based on prior work in DeepSets, GNNs, and transformers. These methods were benchmarked and tuned across multiple environments and evaluated in terms of performance, scalability, and generalizability. The results demonstrate that permutation-invariant critics can match, and in some cases exceed, the performance of the concatenation baseline, which remains a strong reference point. The scalability analysis shows that invariant critics handle increasing agent counts more efficiently in terms of parameters and FLOPs, with transformer-based methods scaling particularly well. Generalization experiments confirm that invariant critics can extend to unseen agent counts, provided training covers a sufficiently broad agent range. The developed methods can handle different agent counts without retraining the critic from scratch, whereas concatenation requires retraining for each configuration, which hinders its application for dynamic environments with varying agent counts during training. Sample efficiency gains were not apparent under the popular Centralized Training Decentralized Execution (CTDE) framework, but could be observed using Centralized Training Centralized Execution (CTCE). Overall, this thesis establishes that permutation- and agent-count invariant critics offer clear advantages for MARL in dynamic and large-scale environments. Among the tested methods, transformer-based architectures proved consistently robust and competitive, highlighting invariant critics as a promising foundation for scalable and generalizable MARL.

Acknowledgement

I would like to sincerely thank my professor and supervisor Prof. Dr. David Spieler for his invaluable guidance, constructive feedback, and for giving me the opportunity to carry out this thesis.

I am also grateful to Johannes Binder and Marcel Hampp for their contributions and continuous encouragement throughout this work.

Contents

1 Introduction

Machine learning as a whole is becoming increasingly popular, transforming industries and changing the way we approach problem-solving. Reinforcement Learning (RL) is one of the three main paradigms of machine learning besides supervised and unsupervised machine learning [1]. In RL, an agent learns to make decisions by interacting with an environment to maximize cumulative rewards [2].

While many well-known applications of RL involve single agents, such as game-playing AIs or robotic control systems, real-world systems often require multiple agents to operate simultaneously and interact with each other. Examples for multi-agent systems are a team of drones coordinating in a search-and-rescue mission, autonomous vehicles navigating traffic, or traders interacting in a financial market [3]. These systems are the domain of Multi-Agent Reinforcement Learning (MARL). In MARL, multiple agents interact with their environment and possibly with each other. These interactions can be cooperative, competitive, or mixed, leading to highly dynamic and complex learning scenarios. Thus, MARL is not a straightforward extension of RL, but rather it introduces new challenges that demand specialized methods and architectures [4].

Given the importance of systems that involve many autonomous actors, it is vital to develop MARL algorithms that can scale, generalize, and adapt to real-world environments. One active area of research is how to build learning systems that can efficiently handle scenarios where the number of agents may change over time, and where the ordering of agents is irrelevant to the underlying problem dynamics. These challenges are the focus of the work presented in this thesis.

Note on writing and coding assistance: The development of this text and associated implementations was supported by tools such as ChatGPT [5] and Grammarly [6], which were used to correct grammar, fix syntax errors, and improve overall clarity. All technical content, analysis, and conclusions are entirely my own.

1.1 Motivation

A key challenge in modern RL is effectively training agents to cooperate or compete within a shared environment to reach their goals. This is especially difficult in MARL because of the curse of dimensionality, which arises from the exponential growth of the joint observation and action space of the agents, as the number of agents grows. The increasing number of agents makes learning effective policies and value functions computationally challenging

[7]. Moreover, in many practical settings, the number of agents may not be fixed: agents can enter or leave the environment dynamically, requiring learning algorithms to be invariant to the number of agents present during both training and deployment [8].

A promising approach to addressing these challenges is to design models that are permutation-invariant to the ordering of agents and can generalize to varying numbers of agents. In MARL, particularly in the Centralized Training Decentralized Execution (CTDE) setting and actor-critic frameworks, critics evaluate the joint state or observations of all agents to compute value estimates of the state of the environment. If these critics use permutation-sensitive architectures, e.g., fully connected Neural Networks (NNs) that process agents sequentially, they may learn representations that do not generalize to different agent orderings and varying numbers of agents. For dynamic environments, this would require completely restarting the training process for all numbers of agents that are expected to occur, creating a resource-intensive training process. Hence, embedding agent observations in a permutation-invariant manner that can handle different numbers of agents is essential to ensure that the critic's output is consistent and generalizable. Permutation Invariance (PI) has the additional benefit of potentially speeding up the training process by increasing the sample efficiency of the methods [7].

Advances in deep learning have introduced several architectures that inherently enable permutation-invariance. These are often designed for tasks like point-cloud segmentation or classification, where the same challenges exist. For example, points in a point cloud have no clear ordering, requiring the models to be PI, and point clouds can also vary in size, just like the number of agents in an environment. An example of such a method derived from point clouds is DeepSets [9], which aggregates agent features via pooling operations. Another domain that requires PI and variable input size is computation on graph structures. Methods designed for that domain, like GraphSAGE (GSAGE) [10] or Graph Attention Network (GAT) [11], can also be used for MARL. Another notable field of deep learning that can be applied is transformer [12] architectures from Natural Language Processing (NLP), which can be constructed PI and handle variable-sized inputs.

These architectures offer a variety of trade-offs in terms of expressiveness, scalability, and computational complexity, potentially making them promising candidates for constructing effective and generalizable PI critics in MARL. This thesis builds on these ideas by investigating how different PI embedding methods perform in embedding agent observations for critics in Multi-Agent Proximal Policy Optimization [13] with CTDE [14]. By systematically comparing these methods, this work examines the models based on their generalizability, scalability, and overall performance, ultimately contributing to more robust and flexible MARL algorithms.

This motivates the main objective of this thesis: *designing permutation- and agent-count invariant critics ideal for generalizing across varying agent counts in MARL while being scalable and competitive with current baselines*. The thesis sets out to develop a Multi-Agent Proximal Policy Optimization (MAPPO)-based training pipeline incorporating several invariant embedding methods for the critic. The effect of the architecture of the embedding techniques and training hyperparameters will be explored in depth, and the

resulting models will be systematically compared based on performance, scalability, and generalization to varying agent numbers.

Ultimately, the goal is to provide empirical insights and best-practice recommendations for building generalizable MARL algorithms, which addresses a gap in the existing literature.

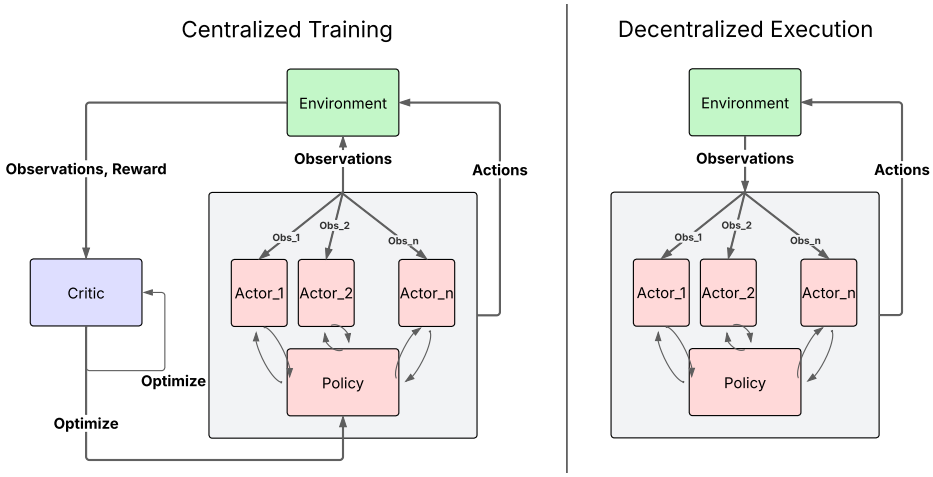


Fig. 1.1.: Visualization of CTDE with the actor-critic framework.

1.2 Problem Description

In MARL, learning effective value functions that can evaluate joint states or observations of multiple agents is crucial for training robust policies. In CTDE frameworks, this challenge is addressed by training a centralized critic that has access to the observations of all agents during training, while the decentralized actors rely solely on their local observations during execution. Figure 1.1 illustrates CTDE used in this thesis. During training, the centralized critic evaluates the joint observations of all actors. Meanwhile, each actor independently selects actions based only on its local observation of the environment. The actors share the same policy but perform independent actions. The critic is used to estimate the expected returns given the joint observations of all agents, and its gradients are used to optimize the policy, encouraging actions that lead to higher expected returns depending on the current state. This way, the critic improves the agents' decision-making by guiding them toward actions expected to be more successful in the current situation. After training, the critic is no longer needed. The agents rely on their shared policy, while acting exclusively on their observations [14].

A critical limitation arises when designing the critic: the joint observations of all agents are inherently unordered and potentially variable in size. If the critic architecture is permutation-sensitive, i.e., its outputs depend on the ordering of agent observations, then the learned value estimates may not generalize to different agent orderings or to environments with a different number of agents. This is especially problematic in real-world applications such as robot swarms [8], which are often dynamic. Another benefit of being able to

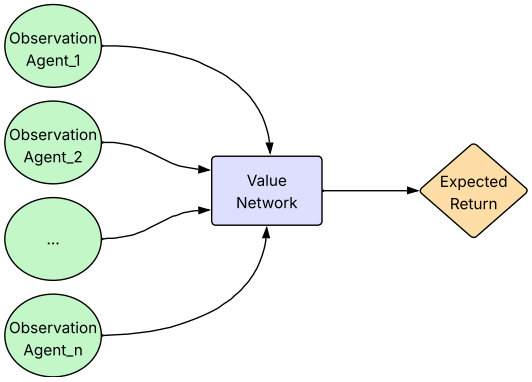


Fig. 1.2.: Visualization of the critic's value network.

generalize to a different number of agents is that it creates the possibility of training with fewer agents, which is computationally easier, and then rolling out the general policy to settings with many more agents.

The other issue with permutation-sensitive models is the curse of dimensionality. For a global state of n agents, there exist $n!$ permutations with the same information, e.g. $[\{o_1, o_2, o_3\} \equiv \{o_1, o_3, o_2\} \equiv \{o_3, o_1, o_2, \} \equiv \dots]$. Therefore, a PI function reduces the joint action space by a factor of $\frac{1}{n!}$, because all permutations produce consistent outputs, as the ordering is irrelevant [7].

The core problem addressed in this thesis is therefore how to create critic network architectures that are scalable, generalizable, and permutation-invariant in MARL with CTDE. The input and output of the value network of the critic is visualized in Figure 1.2. The network must embed the information from each agent's observation in a way that is invariant to the permutation of agents and to variations in the number of agents present. The specific actor-critic algorithm used is MAPPO [13], which represents a state-of-the-art MARL algorithm that can be modified for this thesis.

To formalize this challenge, see Figure 1.1, which presents a high-level overview of the actor-critic architecture used in this thesis. Each actor independently observes its local environment and outputs an action. Meanwhile, the critic receives all agent observations, embeds them using a PI and agent-count invariant network, and outputs a value estimate of all combined observations of the agents. The goal is to assess how different embedding architectures affect the critic network's ability to generalize to varying agent configurations (see Figure 1.2) and how efficiently they scale as the number of agents increases.

This comparative analysis will form the basis for deriving conclusions about the strengths and weaknesses of each embedding strategy. In particular, the evaluation focuses on identifying which methods offer the best trade-offs between sample efficiency, computational scalability, and generalization. The code to reproduce all experiments and results in this thesis is available on GitHub[1].

[1] `https://github.com/SimonH12/deep-rl-agent-embeddings`

1.3 Research Questions

The following research questions will be answered in this thesis:

1. How do different permutation and agent count invariant embedding methods perform in MAPPO?

2. Which critic architectures scale most efficiently to larger agent counts, balancing performance and computational cost?

3. How well do the embeddings generalize to different numbers of agents?

4. Are there best practices on how to train generalizable methods?

5. Is the theoretical sample efficiency gain of PI methods apparent in the experiments?

1.4 Research Contribution

This thesis contributes to the field of MARL by designing, implementing, and evaluating permutation- and agent-count invariant critics for the centralized critic in MAPPO. While the commonly used concatenation strategy provides a simple and strong baseline, it lacks flexibility and scalability. To address these shortcomings, this work investigates alternative critic architectures that incorporate invariance by design.

The key contributions of this thesis are:

- Development and implementation of nine different critic architectures based on DeepSet, GNN, and transformer, adapted to centralized critics in MAPPO.

- Systematic evaluation of these architectures in the Vectorized Multi-Agent Simulator (VMAS) benchmark environments, analyzing performance (RQ1), scalability (RQ2), generalizability (RQ3 & RQ4), and sample efficiency (RQ5).

- Comprehensive hyperparameter tuning in one of the scenarios to ensure fair and reproducible comparisons across methods.

- Detailed analyses of scalability, generalization limits, and alternative training regimes (e.g., CTCE), highlighting trade-offs between architectures in terms of parameters, Floating Point Operations (FLOPs), training time, and memory usage.

1.5 Thesis Outline

The thesis begins with chapter 2, which reviews existing research in MARL and invariant representations, covering traditional MARL architectures, challenges in multi-agent learning, permutation-invariant methods, set- and graph-based embeddings, transformer-based approaches, and benchmark environments, while identifying the research gaps addressed in this work. In chapter 3, theoretical foundations are introduced, of MARL, the MAPPO algorithm, and invariant neural network architectures, including DeepSets, graph neural networks, and transformers. Subsequently, chapter 4 presents the design and implementation of the proposed invariant critics, their integration into MAPPO, and the experimental methodology, including the environments, metrics, and evaluation setup. The experiments are detailed in chapter 5, showcasing the results on performance of different hyperparamerter choices, scalability, and generalizability and providing a thorough analysis of each. Chapter 6 interprets the findings, relates them to the research questions, and discusses their connection to existing work while addressing limitations and implications. Finally, chapter 7 summarizes the main contributions and findings of this thesis and outlines directions for future research.

2 Related Work

2.1 Introduction to Related Work

Multi-agent systems developed from single-agent RL frameworks. Initial work focused on including two-player games, which often had competitive objectives, before extending to true multi-agent systems with numerous agents. Now, MARL has many practical applications, ranging from stock trading [15, 16], networking [17, 18], social sciences [19, 20], robotics [21, 22, 23] to strategy games [24, 25].

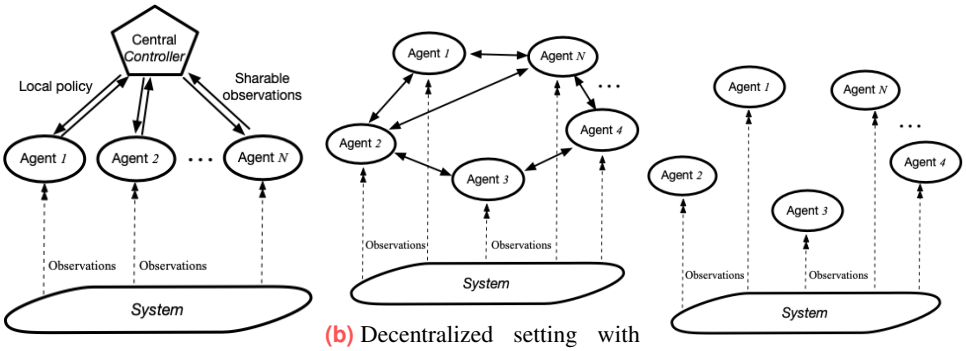
Competitive and cooperative agents. Since the field of MARL is so diverse, MARL techniques are often classified based on how agents interact with each other. Agents can be *fully competitive* with each other, meaning each agent has its own distinct objective and reward function. Examples of this are classic multi-player games like chess, go, or poker. In competitive scenarios, the return of the agents typically sums to zero, as there are winners and losers. Contrary to this are *fully cooperative* scenarios, in which the agents act towards a common goal with a shared reward function. Between these extremes are *mixed* methods, which have characteristics of both fully competitive and fully cooperative environments.

Like single-agent RL, MARL methods can also be grouped into value, gradient, or policy-based methods. However, since this is not specific to MARL and this thesis focuses on the architectural aspects of MARL networks rather than the algorithmic classes, a detailed discussion of these distinctions will not be further explored. For a thorough review and detailed explanation of typical RL techniques, see [2], and for a focus on MARL see [26].

2.1.1 Traditional MARL Architectures

Information structure. The way agents access and exchange information influences the design of the MARL methods. There are three common types of information structures in MARL: the centralized setting, the decentralized setting, and the decentralized setting with networked agents [3].

In the centralized setting, a central controller exists, which can collect and share information from the agents. The controller can also manage the policy of the agents. The centralized paradigm enables the popular CTDE approach to the training and rollout phases. During training, the centralized critic has access to all observations managed by the central controller. In the execution phase, each agent acts independently, only having access to its



(a) Centralized setting **(b)** Decentralized setting with networked agents **(c)** Fully decentralized setting

Fig. 2.1.: Three common information structures in MARL: centralized setting, decentralized setting with networked agents, and fully decentralized setting. In the centralized setting, a central controller aggregates agent information and coordinates policies. The decentralized setting with networked agents lacks a central controller, agents receive their observations from the environment and can communicate to inform their decision-making. Lastly, when agents cannot communicate and only receive their own observations, the setting is called fully decentralized [3]. Figures taken from [3].

observation. This setting is grounded in the Decentralized Partially Observable Markov Decision Process (Dec-POMDP) framework (see paragraph 3.1.1). Importantly, the *decentralized* in Dec-POMDP refers to the agents' independent policies and not the absence of a centralized controller. The policy can also be made centralized, utilizing the observations of all agents, becoming CTCE. A key limitation of the centralized approach is the reliance on a simulator or environment capable of providing complete global information, which can be difficult to construct in real-world applications.

In contrast to the centralized setting, a decentralized setting is another common possibility, where no central controller exists. The agents receive their observations directly from the environment. This kind of information structure is called *decentralized setting with networked agents*, if the agents can communicate with one another. Here, agents can exchange information and base their actions on their observations as well as the received information from other agents in the environment. If the agents can not communicate and base their actions exclusively on their own observations, the setting is *fully decentralized* [3].

Heterogeneous and homogeneous agents. Agents in an environment can either be homogeneous or heterogeneous. Homogeneous agents are indistinguishable from one another. They share the same reward structure, action space, and observation space. This enables *parameter sharing*, i.e., using a single neural network for each agent [27]. Parameter sharing can improve learning efficiency, while still enabling the agents to act independently at execution, based on their local state. On the contrary, heterogeneous agents have different roles, capabilities, observation, and action spaces. This makes parameter sharing impossible or only very limited. In general, homogeneous agents simplify the RL process, especially for the cooperative setting [3].

Simulator design. Many multi-agent environments are based on visual simulations. Deep RL methods are suited for such settings, as they can process high-dimensional visual input efficiently. For example, the actor and critic networks can be designed to receive frames of a game. Such architectures are common in Atari games [28] or the popular StarCraft learning environment [29]. However, visual input introduces additional challenges, particularly for scalability and computational resource needs, as high-fidelity visual simulators require significant resources. Additionally, video game scenarios only have limited applications to real-world robotic scenarios [30].

To address these shortcomings, vectorized simulators are an alternative. Instead of processing visual data, these environments represent agent observations as numerical vectors. This type of simulator can be very efficient and can scale to many agents. They also offer greater flexibility, making it possible to design new environment simulations more easily [30]. Examples of vectorized environments include Multi-Agent Particle Environment [31, 32] and VMAS [30]. VMAS is a project that ported all of the Multi-Agent Particle Environment (MPE) environments while being up to 100 times more efficient than MPE.

2.1.2 Challenges of MARL

Although the field of MARL is very diverse, several key challenges in MARL settings are often encountered. While not exhaustive, this section highlights several critical aspects of MARL.

Credit assignment problem. Unique to MARL is the credit assignment problem. In cooperative environments with a globally shared reward, it becomes difficult to determine how much each agent contributed to the shared reward. For example, one agent may have acted optimally and been responsible for the majority of the achieved reward, but since the reward is shared, the other agent who did not act optimally will still receive the same reward. One proposed method to mitigate the credit assignment problem is *value decomposition*. In these approaches, the global value function is decomposed into separate, agent-specific, value functions [33, 34]. Another method is Counterfactual Multi-Agent (COMA) [35], which uses a centralized critic to estimate what would have happened if one agent acted differently while keeping the other agents fixed. This allows for each agent's individual contribution to be estimated. In CTDE with fully cooperative agents, credit assignment can be solved implicitly if the centralized critic provides enough information through the policy gradients, and if exploration is maintained during training [36].

Moving target problem The moving target problem describes the non-stationarity of multi-agent systems. In MARL, each agent updates its policy during the learning phase, which causes the environment to change from the perspective of every other agent. Simultaneously, the other agents are also learning and adapting their policy, thereby again influencing one another, creating a feedback loop. This mutual influencing leads to a

constantly changing training dynamic. Developing algorithms that are both general and reliable in the face of changing conditions is a complex task and remains a central focus of current research [26].

Curse of dimensionality. As the number of agents increases, scalability becomes a main concern. Since each agent has observations and actions, the joint action and observation space increases exponentially with the number of agents. The scalability has to be considered in the design of the RL algorithm chosen and in the simulator used. A consequence of the exponential increase in dimensionality is poor sample efficiency of many MARL algorithms, which means the training becomes even more complex. A straightforward approach to handling multi-agent observations is to concatenate the observations of the agents into a single vector and pass this vector through a network. While simple, this technique does not scale well with the number of agents, as the input of the network has to scale with the number of agents. Nevertheless, many popular MARL algorithms, like MAPPO [13] or Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [37], use concatenation of the observations in their default implementation. For environments with homogeneous agents, using PI methods is a possible solution to this issue. For a global state of n agents, there are $n!$ permutations that have semantically the same information. A function that is PI yields the same output to all $n!$ permutations of the global state, therefore, a PI function reduces the state space by a factor of $\frac{1}{n!}$.

2.1.3 Achieving Permutation Invariance (PI)

Since PI is one of the main solutions to the curse of dimensionality, it is worth exploring further how PI can be achieved in MARL. There are primarily two methods to create PI MARL algorithms, via data augmentation or by using inherently PI deep learning techniques and adapting them to MARL [7].

Permutation Invariance with data augmentation. One technique to enforce PI is with data augmentation, which entails creating more data during the training phase. This can be achieved by shuffling the data of the n agents to create $n!$ permutations. During training, the model is encouraged or constrained to produce the same output for each permutation, to construct a PI model [38]. The shuffling of the data during the training process introduces significant computational overhead, particularly for environments with many agents, which is a major drawback of this technique.

Permutation Invariance with invariant architectures. An alternative way to create PI models is by using deep learning architectures that are naturally PI and incorporating them directly in the MARL algorithms. These architectures can for example be set-based [9, 23, 39], GNNs-based [40, 41, 42] or transformer-based [43, 44, 45, 41]. Such architectures handle unordered inputs and enable scalability to large agent populations. Still, this method

of achieving PI presents its own challenges, because of the use of shared embedding layers and pooling operations that can limit their expressive power if they lack sufficient dimensionality [46].

2.2 Set-Based Representations in MARL

In environments with homogeneous agents, the collection of agents in an environment can be modeled as a set. A set is made up of unordered elements without duplicates. This applies to the agents, too. Since they are identical, there is no trivial ordering to the agent set, and each agent appears only once in the environment [47].

The domain of machine learning models having a set as input, thus requiring PI architecture, is well established, most notably in the field of point cloud classification. In the context of point clouds, permutation-invariant networks are necessary as there is no inherent ordering of points in space. These models must also be computationally efficient to handle varying and often large numbers of points. Moreover, one classification network must generalize to other point clouds with different amounts of points. All these challenges also exist in MARL, where agents replace points, and their observations replace the spatial coordinates of the points [48].

Another parallel is the need to capture both local and global features. In point cloud tasks like part segmentation, a model must learn the local role of the point, but the point also has to be interpreted within the global structure of the point cloud [48]. Similarly, in MARL, the agent has to infer its local importance as well as the global impact on the reward its actions have [49].

2.2.1 DeepSet and PointNet

PointNet [48] played a foundational role in the development of deep learning for point cloud data. It was the first deep neural network architecture specifically designed to process unordered point sets directly, without requiring intermediate representations like voxel grids or meshes.

DeepSets [9], which was published around the same time, presented a general framework for creating models for set data. Both PointNet and DeepSets share the same principal concept: applying a shared function $\psi()$ to each element in a set X, whose results are then aggregated and passed through another function $\phi()$:

$$f(X) = \phi\left(\sum_{x \in X} \psi(x)\right) \tag{2.1}$$

This framework is PI and scalable to the number of set elements n. The DeepSets principles were adapted to MARL by [23], further referred to as SwarmRL in this thesis. In SwarmRL, the framework is integrated with Trust Region Policy Optimization (TRPO) to model homogeneous multi-agent environments.

2.2.2 SetTransformer

Another method originally developed for visual tasks, such as shape recognition and image classification, is SetTransformer [39]. In contrast to DeepSets [9], which uses simple permutation-invariant pooling operations, SetTransformer works on sets by using multi-head attention mechanisms [12]. Like DeepSets, it is PI and specifically designed to be a scalable solution, considering that scalability is often a limitation with self-attention, which scales quadratically with the size of the input set. Due to these characteristics, SetTransformer is well-suited for MARL and has already been explored in model-free and model-based offline MARL settings [44].

2.3 Graph-Based Representations in MARL

Graphs have many of the key characteristics found in MARL settings. In the graph context, agents in an environment can be modeled as nodes in a graph. Like agents, nodes in a graph can vary in number and are unordered.

2.3.1 GNN as Agent Embeddings

GNNs are particularly suited for MARL because they often incorporate communication protocols that closely align with the information structures encountered in multi-agent systems. For example, GNNs can model environments where agents communicate only with their nearest neighbors, which is often applicable in real-world scenarios [50]. To ensure scalable models, aggregational GNNs, like Graph Convolutional Networks (GCNs) [51, 52], are especially useful, as they efficiently summarize neighborhood information. Due to these factors, GNN methods have already been successfully applied to MARL [42].

2.3.2 Advanced GNN Frameworks

Classic GCN methods often struggle with generalizing to unseen nodes. To address this limitation, other GNN frameworks like GSAGE [10] were developed, which are explicitly made for inductive learning and already have been applied to MARL [53, 54]. Similarly,

GNN techniques using attention mechanisms, e.g. GAT [11] and its successor Graph Attention Network Version Two (GATv2) [55] were used in MARL [50, 41].

GAT was incorporated by [50] in a cooperative MADDPG setting to process the feature representation of the agents before passing them to the Q-network. Their experiments demonstrated that the MADDPG using GAT can significantly outperform plain MADDPG and QMIX [34]. GATv2 [55], an evolution of GAT, promises more expressiveness of the underlying model by changing the computation order of attention weights. Whether or not GATv2 outperforms GAT depends on the specific environment both are used in. For example, in [41], both techniques performed very similarly.

For a comprehensive overview of GCN, GSAGE, and GAT methods in MARL see [56]. Given the relevance and efficiency of GNN methods in addressing the challenges seen in MARL, GSAGE, GAT, and GATv2 will be employed in this thesis.

2.4 Transformer-Based Representations in MARL

Although less common in MARL, there are several examples of transformer-based architectures applied in MARL. Since attention blocks are inherently PI, when positional encoding is omitted, and capable of handling variable-sized inputs, they are particularly well suited for MARL. One example is the work by [57], which uses multi-headed attention in a shared actor-critic network in conjunction with Proximal Policy Optimization (PPO). Their approach also employs agent-wise fully connected networks to further encode each agent's observations, similar to the DeepSets encoder network [9] and the row-wise feedforward network in SetTransformer [39]. However, instead of adopting the widely used CTDE paradigm, they utilize CTCE. Notably, the authors did not provide code for their method. Nevertheless, the fundamental idea of employing attention mechanisms within the critic network serves as inspiration for the methods proposed in this thesis.

Another relevant work using attention mechanisms in actor-critic MARL with cooperative agents under the CTDE framework is [58]. In contrast to using a single centralized critic, [58] learns an individual critic for each agent, which leverages the observations of other agents. While the multi-headed attention mechanism within the critic employs parameter sharing, each agent uses a unique encoder and decoder Multi Layer Perceptron (MLP), allowing the architecture to support heterogeneous agents. Since this thesis focuses on homogeneous agents, the architectural complexity required for heterogeneous settings is not necessary. However, due to the architectural similarities between the approaches of [58] and [57], both are used as guiding references for the attention-based methods developed in this thesis.

2.5 MARL Benchmarks

As this thesis aims to compare a variety of embedding methods within a common scenario, it is essential to consider the existing benchmarks conducted by other researchers. For instance, in the MPE *Navigation* environment, [44] compared DeepSets, MLP, GCN, and a SetTransformer architecture. They used model-free and model-based offline RL. Their analysis focused on average rewards and highlighted the different results when training with a small number of agents ($n = 3$) versus many agents ($n = 30$). Models that performed well with small numbers of agents, such as GCN and MLP, showed worse performance when scaling the number of agents in the environment up. This emphasizes how important it is to analyze the approaches in different conditions to test them properly.

Another relevant paper [41] compared GNN approaches. In it, researchers propose an evolution of GAT and GATv2 using cross-attention. Their model was competitive with other embeddings like GAT, GATv2, GSAGE, and GCN. However, the cross-attention mechanism they propose relies on an expressive graph topology. In the environments proposed in this thesis, the graph is fully connected, and an informative topology would have to be artificially added.

A limitation of both benchmarks [44, 41] is that they rely on comparing the average rewards of the methods. While performance is crucial, the complexity and resource needs of the methods must also be evaluated to better contextualize their results.

2.6 Research Gaps

Based on the literature review, four key research areas have been identified that require further analysis:

1. Generalizability of Embedding Approaches
2. Training Best Practices for Generalizable Architectures
3. Limits of Generalizability
4. Complexity Analysis of Invariant Embedding Methods

2.6.1 Generalizability

A notable research gap is the lack of analyzing the generalizability of critic architectures in the research.

Generalizability of Embedding Approaches Specifically, the method's ability to perform well across environments with a varying number of agents. While architectures such as DeepSets [9] are inherently PI and invariant to input size, others like SetTransformer [39] can be modified to handle a varying set size. As an example, [23] investigates methods that allow a varying input size, but do not assess the generalizability of the suggested approaches. Similarly, the transformer-based approaches [58, 57] showcased strong performance in cooperative environments, but their generalizability was not analyzed. Testing the generalizability could be achieved by training these models on a set size different from the evaluation set size. Consequently, identifying models best equipped to handle different set sizes is one of the central research objectives of this thesis.

Training Best Practices for Generalizable Architectures. Due to the limited attention given to generalizability, there is no clear consensus on efficient training strategies for achieving generalization in the architectures. It remains unclear whether training on a range of agent set sizes leads to better generalization than training solely on a fixed agent configuration.

Limits of Generalizability. Another insufficiently researched area is the practical limits of generalizability. Specifically, it is unclear at what number of agents the model fails to perform well, and if these limits vary across different embedding techniques.

2.6.2 Complexity Analysis of Invariant Embedding Methods

As remarked by the RL overview in [47], there is a need for increased transparency regarding the computational demands of MARL algorithms. Resource demands can vary significantly between methods. For instance, attention mechanisms typically use more parameters and operations than traditional MLPs. To quantify and compare the complexity of the embedding approaches, metrics such as FLOPs or model parameters can be employed. These metrics can be efficiently computed in Python using packages such as *ptflops* [59]. In addition, runtime metrics such as training duration and memory usage should also be considered to provide a comprehensive evaluation of each method's resource demands. How the metrics change with increasing numbers of agents also has to be taken into account.

Background

3

3.1 Reinforcement Learning

This section serves as an overview of the essential concepts used throughout this thesis. The basics of RL are assumed to be prior knowledge. To get a deeper and more detailed understanding of RL, the following resources are recommended [60, 2, 61, 62].

3.1.1 Markov Decision Process

Markov Decision Process (MDP). In RL, a MDP represents the formal framework for modeling the interaction of an agent and its environment. The notation and concepts in this section are based on the lecture by David Silver [62] and the book "A Concise Introduction to Decentralized POMDPs" [60]. Formally, MDPs are defined as the tuple:

$$\text{MDP} := \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$$

An agent can be in a state s from the set of states $\mathcal{S}$ in the environment and take an action from the set of actions $\mathcal{A}$. These actions influence the environment, leading to transitions between states. The transition probabilities are captured by the state transition probability function $\mathcal{P}$, which defines the chance of moving to a new state given the state and action at time step t, as such $\mathcal{P}^a_{ss'} = P[S_{t+1} = s' \mid S_t = s, A_t = a]$. To guide the agent's behavior in the environment, a reward function $\mathcal{R}$ is defined. This function defines the expected reward of an action in a state at a time step, with $\mathcal{R}^a_s = E[R_{t+1} \mid S_t = s, A_t = a]$.

Using the reward function, the behavior of the agent can be adjusted to perform actions leading to higher rewards. The cumulative reward an agent receives from step t onward is called the total return. The goal in the environment is defined through this reward model. The agent must maximize the total reward to perform its task. The final component of the MDP is the discount factor γ. The discount factor ensures a finite total return and can be adjusted to prioritize either immediate rewards, with γ closer to 0, or long-term rewards, with γ closer to 1.

A key concept in solving an MDP is the policy, denoted as π. A policy is a mapping from states to actions, defined as $\pi(a|s) = P[A_t = a \mid S_t = s]$. The agent acts based on the policy. The goal of RL is to find the optimal policy π^* that maximizes the expected total

return from each state. The optimal policy dictates the best action to take in each state to achieve the highest cumulative reward over time. Without a policy, the agent would not know how to decide on actions and would not learn to act successfully in the environment [62].

Partially Observable Markov Decision Process (POMDP). In the case where the agent does not have access to the full state of the environment, the MDP becomes a POMDP. The environment is no longer fully observable. Therefore, the agent does not know the true state $s \in \mathcal{S}$ but instead receives the observation $o \in \mathcal{O}$ that provides only partial information about the state. Therefore, the policy depends only on the observation, rather than the state. The probability of receiving an observation o, based on an action that leads to state s', is modeled with the observation function $\mathcal{Z}$ as follows: $\mathcal{Z}^a_{s'o} = P[O_{t+1} = o \mid S_{t+1} = s', A_t = a]$. A POMDP can then be defined as such [62]:

$$\text{POMDP} \coloneqq \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma \rangle$$

Dec-POMDP. Both MDP and POMDP are defined for a single agent acting in the environment. The Dec-POMDP extends the concepts of regular POMDPs to multiple agents acting in the environment. The term *decentralized* in Dec-POMDP indicates that each agent has access to its own observation and not the complete global state. Moreover, agents do not observe each other's observations and must act without direct communication between each other, making coordination inherently more difficult. To model a Dec-POMDP, the POMDP is extended as follows:

$$\text{Dec-POMDP} \coloneqq \langle \mathcal{N}, \mathcal{S}, \mathbb{A}, \mathbb{O}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma \rangle$$

New to the Dec-POMDP is the set of agents $\mathcal{N} = \{1, \ldots, n\}$. The set of actions and observations, i.e., $\mathcal{A}$, $\mathcal{O}$ respectively, have to be adjusted to the multi-agent environment. The set of actions becomes the set of joint actions $\mathbb{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$, with a *joint action* $\vec{a} \in \mathbb{A}$ being the set of all actions taken by each agent i at a step t, leading to $\vec{a} = \{a_1, \ldots, a_n\}$. Likewise, the set of observations becomes the set of joint observations $\mathbb{O} = \mathcal{O}_1 \times \cdots \times \mathcal{O}_n$. Each agent i only observes $o_i \in \mathcal{O}_i$.

The transition function $\mathcal{P}$, observation function $\mathcal{Z}$, and reward function $\mathcal{R}$ are defined analogously to those in the POMDP, but now operate over joint actions and joint observations. All agents receive the same reward for a joint action, requiring cooperation to achieve the highest return [60]. The Dec-POMDP forms the basis for the MARL algorithm used in this thesis.

Partially Observable Stochastic Game (POSG). The POSG framework is a generalization of the Dec-POMDP. The Dec-POMDP can only be used with cooperative agent systems, as it specifies a common shared reward function. A POSG does not have this restriction, instead, each agent has its own reward function. This enables competitive scenarios in which the agents have different or competing objectives. A POSG itself is a

generalization of a Stochastic Game (SG) where agents have access to the true state of the environment instead of only observations, analog to POMDP and MDP. Finally, SG is a generalization of an MDP where multiple agents can be defined [26]. Thus, a Dec-POMDP can be seen either as a specialization of a POSG with a shared reward, or as an extension of a POMDP to decentralized multi-agent setting.

Guarantees in Dec-POMDP. In finite-horizon Dec-POMDPs with multiple agents, finding the best possible joint policy is extremely challenging. The problem is NEXP-complete, harder than NP-complete [63], meaning that finding the optimal joint policy can require time exceeding exponential time in the worst case. Even finding an approximately good policy that is within some small value ϵ of the optimal policy is just as hard. For infinite-horizon problems, the situation is even worse, the problem is undecidable, thus no algorithm can guarantee a solution in all cases. Because of this, most exact or guaranteed approximation methods are not practical. Instead, heuristic methods are often used in practice. These can work well, but they don't offer any guarantees about how close their solutions are to the optimal one [60].

3.1.2 Proximal Policy Optimization

PPO [64] is a reinforcement learning algorithm designed to balance ease of implementation, sample efficiency, and stability. PPO belongs in the category of on-policy policy gradient algorithms, meaning it learns from data sampled using the current policy, and directly optimizes the parameters θ of a stochastic policy $\pi(a|s)$ by maximizing expected returns. PPO can be seen as a simplified and more practical alternative to TRPO [65]. It achieves similar benefits, such as improved sample efficiency and stable learning, while being easier to implement [64]. This makes it a good basis to test different embedding methods for this thesis.

Actor-critic. The specific variant of PPO used is the actor-critic style. This approach employs two separate neural networks: the *actor network*, denoted as π_θ, which is responsible for selecting actions based on the current policy, and the *critic* or value network, denoted as $V_\phi(s)$, which estimates the expected return from a given state s. The critic is used only during training to provide learning feedback for the actor, guiding policy updates by reducing variance in the gradient estimation. The critic network is parametrized by ϕ [64].

Clipped Surrogate Objective

The clipped surrogate objective is key to PPO [64], because it enforces stable policy updates during training. PPO's surrogate objective builds on Conservative Policy Iteration (CPI), which is used by TRPO. Here, the surrogate objective is defined as:

$$L^{\text{CPI}}(\theta) = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right], \quad \text{with } r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{3.1}$$

In Equation 3.1, $\pi_\theta(a_t \mid s_t)$ is the current policy's probability of taking action a_t in state s_t at time step t. Similarly, $\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ is the probability under the old policy before the current update. With both probabilities, the ratio $r_t(\theta)$ compares the new and old policy behavior at a given state-action pair. The estimated $\hat{A}_t$ at time step t reflects how much better or worse an action a_t is compared to the average action in state s_t. The concept of $\hat{A}_t$ will be further explained in subsubsection 3.1.2. This estimate is computed over a batch of samples and averaged, denoted by $\hat{\mathbb{E}}_t$, the hat indicating that it relies on empirical data.

The surrogate $L^{\text{CPI}}(\theta)$ encourages increasing the likelihood of actions with positive advantages but does not restrict the size of policy updates. Unconstrained updates can destabilize the training process. That is why PPO introduces the Clipped Surrogate Objective (CLIP) mechanism:

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

The clip function constrains the probability ratio $r_t(\theta)$ to the interval $[1 - \epsilon, 1 + \epsilon]$, with ϵ being a typically small hyperparameter. This clipping mechanism prevents the probability ratio from deviating too far from 1, thereby avoiding large, potentially destabilizing updates. The min operator ensures the objective $L^{\text{CLIP}}(\theta)$ does not increase beyond the clipping range, preventing large updates to the policy that could destabilize training.

Generalized Advantage Estimation

To reduce variance and improve learning stability, PPO uses Generalized Advantage Estimation (GAE) [66] to compute the advantage function $\hat{A}_t$. GAE balances bias and variance by mixing multi-step temporal-difference (TD) errors using a smoothing parameter $\lambda \in [0, 1]$. It is defined as:

$$\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l}, \quad \text{where} \quad \delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{3.2}$$

Here, r_t denotes the immediate reward received from the environment at time step t. $V_\phi(s)$ is the approximate value estimate produced by the critic network, γ is the discount factor, and δ is the TD error at time step t. By adjusting λ, GAE can trade off between high-bias/low-variance, with small λ, and low-bias/high-variance, with large λ, advantage estimates. Note that Equation 3.2 corresponds to the estimation used in the original PPO paper [64], other estimators can be used as well.

PPO Algorithm

PPO uses Stochastic Gradient Descent (SGD) with multiple epochs and minibatches, which allows for a more efficient use of collected data compared to classic policy gradient methods that perform only a single gradient update per data sample.

In each iteration of PPO, data is collected from the environment by the agent interacting with it using the current policy $\pi_{\theta_{\text{old}}}$. Then, the advantage estimates $\hat{A}_t$ for all timesteps are calculated. The critic network V_ϕ estimates the value of the agent's current state and is essential for computing the advantage function.

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

After computing the advantage estimates, the policy is optimized over multiple epochs using minibatches to maximize the total loss $L(\theta, \phi)$. This loss includes three parts, which are balanced by the coefficients c_1 and c_2:

1. The clipped surrogate loss $L_t^{\text{CLIP}}(\theta)$ that stabilizes updates by constraining policy changes (see subsubsection 3.1.2).

2. The value function loss $L_t^{VF}(\phi)$, which trains the critic by minimizing the squared error between its value prediction $V_\phi(s)$ and a target value V_t^{targ}. The target value estimates the expected return using either bootstrapped values, e.g., with TD or GAE, or actual observed returns with Monte Carlo methods.

$$L_t^{VF}(\phi) = (V_\phi(s_t) - V_t^{\text{targ}})^2$$

3. An entropy term $S[\pi_\theta](s_t)$. The policy entropy S encourages exploration by preventing premature convergence to deterministic policies [67].

Algorithm 1: PPO, Actor-Critic Style [10].

for *iteration* $= 1, 2, \ldots$ **do**
  for *actor* $= 1, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in the environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  Optimize $L(\theta, \phi)$ using K epochs and minibatch size $M \leq NT$
  $\phi_{\text{old}} \leftarrow \phi$
  $\theta_{\text{old}} \leftarrow \theta$

It is important to note that the line specifying *actor* $= 1, \ldots, N$ does not imply multiple agents in a single environment. Instead, it refers to N parallel or sequentially running environments, each collecting T steps of experience using the current policy $\pi_{\theta_{\text{old}}}$.

3.1.3 Multi-Agent PPO

MAPPO extends the original PPO algorithm to enable cooperative MARL, and is an example of a Dec-POMDP (see paragraph 3.1.1). Each agent $i \in \{1, \ldots, n\}$ follows its own policy $\pi^i_{\theta^i}$. Given that all agents in this thesis are homogeneous, parameter sharing is used. This simplifies the per-agent policy $\pi^i_{\theta^i}$ to the shared policy π_θ, as all agents use the same policy with shared parameters θ.

Centralized Training Decentralized Execution. Training follows the CTDE paradigm, where the decentralized policy is trained using a centralized critic $V_\phi(O)$ that takes as input the set of all observations $O \in \mathbb{R}^{n \times d}$, with each individual observation $\vec{o}_i \in \mathbb{R}^d$. Once trained, agents act only based on their current observation using the learned, shared policy. For a system of n agents indexed by $i = 1, \ldots, n$, the CLIP objective for MAPPO has to be adapted to multi-agent settings as follows:

$$L^{\text{CLIP}}_{\text{MAPPO}}(\theta) = \hat{\mathbb{E}}_{i,t}\left[\min\left(r^i_t(\theta)\hat{A}^i_t,\ \text{clip}(r^i_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}^i_t\right)\right]. \tag{3.3}$$

A notable adaptation from the standard PPO algorithm [64] is that instead of assuming a fully observable environment, MAPPO only requires partial observability, making it suitable for Dec-POMDPs. Therefore, agent observations are used during both training and rollout phases, rather than the true global state of the environment. When MAPPO is trained using CTDE, it is also referred to as Centralized Proximal Policy Optimization (CPPO). Alternatively, a decentralized critic can be used, resulting in the so-called Independent Proximal Policy Optimization (IPPO). Since this thesis focuses on the CTDE setting, IPPO will not be discussed further. From this point forward, any reference to MAPPO implicitly refers to the centralized-critic variant, i.e., CPPO.

CTCE. Although the CTDE paradigm is more commonly used in MARL, CTCE is also possible [57]. The difference from CTDE is that the policy and the critic have access to all agent observations during training and execution. Agents still act independently but are aware of the entire observation space. CTCE faces the same challenges as CTDE, with the curse of dimensionality being even more severe because of the increased input dimensionality for both the actor and the critic.

3.2 Equivariance and Invariance Principles

In general, methods that show aspects of exchangeability, meaning that their outcome is independent of the order of input instances x_i, are classified into two distinct types: permutation invariant (PI) and permutation equivariant (PE). In this chapter, the idea

behind PI and PE functions is explained. Understanding the principles of PI and PE is important to know why they need to be considered in the context of MARL. Moreover, these principles help to decide which approaches are suitable for this thesis.

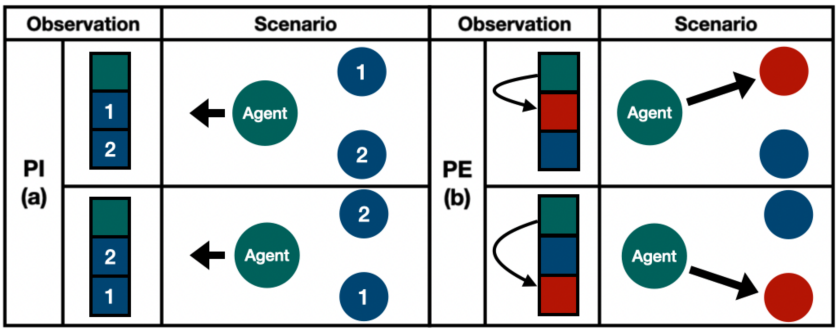


Fig. 3.1.: Concept of PI and PE [43]. (a) A MARL algorithm is PI if the order of agents in its observation does not influence the decision-making. I.e., the agent moves left independently of the ordering of agents one and two. (b) A MARL algorithm is PE if the order of agents changes the outcome in the same way the inputs were permuted. I.e., the agent interacts with the red agent independently of the position of the red agent in the observation space.

3.2.1 Permutation Invariance

A function $f_{\text{pi}} : X \rightarrow Y$ is PI if its output does not change based on permutations of the order of its input elements $x \in X$. The permutations are performed via a permutation function $\pi()$. With this, a PI function is defined as follows [68]:

$$f_{\text{pi}}(x_1, x_2, \ldots, x_n) = f_{\text{pi}}(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$$

Examples of PI functions include $\max(x_1, x_2, \ldots, x_n)$ and addition. In MARL, PI implies that an approach is not influenced by the ordering of agents. A critic's value estimate or an agent's decision-making is therefore invariant to the order of agents (see Figure 3.1) [68].

3.2.2 Permutation Equivariance

The concept of PE requires that applying a permutation π to the input elements $x \in X$ results in the output elements $y \in Y$ being permuted in the same way by the PE function $f_{\text{pe}} : X \rightarrow Y$ [68]:

$$f_{\text{pe}}(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}) = [y_{\pi(1)}, y_{\pi(2)}, \ldots, y_{\pi(n)}]$$

Examples of PE functions include elementwise transformations applied independently to each input and the identity function. In MARL, permutation equivariance allows an agent to make decisions that depend on relationships with other agents. For example, if an agent

must act on a specific red agent, its behavior should be consistent regardless of the order in which the red agent appears in its observation (see Figure 3.1). PE is often overlooked in MARL approaches [68]. However, in settings with limited interactions and homogeneous agents, PE is less critical and therefore not required for the approaches explored in this thesis.

3.3 Pooling

Requirements. Pooling in MARL for agent states must satisfy two properties:

- Permutation invariance (see subsection 3.2.1)
- Support for variable-sized inputs

In the MARL setting, this means that the ordering of agents should not affect the output, and the number of agents in the environment may vary. A secondary but important consideration is handling variable agent populations. Pooling methods should enable agent swarms to be dynamic and scale to large numbers without leading to excessive growth in time or resource complexity.

Encoder-decoder architecture. Since agents are unordered, the logical data structure to represent them is a set. A common architecture for PI functions is the encoder-pooling-decoder framework [9]:

$$f(\{x_1, x_2, \ldots, x_n\}) = \phi\big(\oplus(\{\psi(x_1), \psi(x_2), \ldots, \psi(x_n)\})\big) \tag{3.4}$$

$\psi()$ and $\phi()$ are transformations, typically layers in a NN. The encoder $\psi()$ is applied independently to each element x_i of the set. The decoder $\phi()$ then takes the aggregated, encoded features and produces the desired output. The pooling operation $\oplus$ is responsible for converting the variable-sized input set of encoded elements into a fixed-size representation [39]. Common examples of pooling operations include mean or max pooling.

3.3.1 Pooling Methods

This section introduces several common pooling operations that are both permutation invariant and invariant to the number of elements in the input set X. Such operations are fundamental in set-based and graph-based models, where the input order is random and the number of elements can vary.

The first method is *mean pooling*, which computes the element-wise average over the input set [69]:

$$\oplus_{\text{mean}}(x_1, \ldots, x_n) = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Mean pooling can be extended to *weighted mean pooling* by assigning a weight w_i to each input element x_i, with the constraint that the weights sum to 1. This gives each input a varying importance and provides more flexibility [70]. Weighted mean pooling generalizes mean pooling: if all weights are equal, i.e., $\forall_{i \in \{1,\ldots,n\}} : w_i = \frac{1}{n}$, it becomes mean pooling.

$$\oplus_{\text{w\_mean}}(x_1, \ldots, x_n) = \sum_{i=1}^{n} w_i x_i \quad \text{with} \quad \sum_{i=1}^{n} w_i = 1$$

Another widely used pooling method is *max pooling*, which selects the element-wise maximum across the input set. This operation highlights the most dominant features present in the input:

$$\oplus_{\max}(x_1, \ldots, x_n) = \max\{x_1, \ldots, x_n\}$$

3.3.2 Pooling in MARL

The encoder-pooling-decoder formulation from Equation 3.4 has to be adapted because of its fixed output size. In MARL, the output needs to scale with the number of agents n. For example, when a critic uses an encoder-decoder structure, it must produce one value estimate for each agent i in the environment. To achieve this, the function can be applied once per agent. In this setup, each agent receives a tailored value estimate. To personalize the output for each agent, the decoder $\phi()$ takes as input both the agent-specific input x_i and the shared, fixed-size representation produced by the aggregator method. In the context of MARL, the agent-specific input x_i corresponds to the agent's observation $\vec{o}_i$. This architecture is expressed as:

$$f(\vec{o}_i, \{\vec{o}_1, \ldots, \vec{o}_n\}) = \phi\big(\vec{o}_i, \oplus(\{\psi(\vec{o}_1), \ldots, \psi(\vec{o}_n)\})\big) \tag{3.5}$$

For a visual illustration of the information flow, see Figure 3.2. This architecture is commonly used in PI algorithms and appears in similar forms in [23, 48]. Note also that Equation 3.5 is conceptually equivalent to the basic formulation of a GNN, as shown in Equation 3.6.

Global State Embedding. The aggregation of encoded features will be referred to as the Global State Embedding (GSE). This fixed-size vector represents the pooled and embedded global state of the environment.

$$\text{GSE} = \oplus(\{\psi(\vec{o}_1), \ldots, \psi(\vec{o}_n)\})$$

There are two options for creating the GSE: either including or excluding the local observation $\vec{o}_i$ of agent i. If $\vec{o}_i$ is omitted and not aggregated with the others, each agent receives a custom GSE, as the aggregation is performed over a distinct subset of observations for each agent. In contrast, if $\vec{o}_i$ is included in the aggregation, a single shared GSE can be computed once and reused by all agents, which improves computational efficiency. Both types of GSE are common in set-based algorithms employing the encoder-decoder architecture. For instance, PointNet [48] and GAT [11] include all input elements in the GSE, whereas GraphSAGE [10] and SwarmRL [23] exclude the local representation from the aggregation.

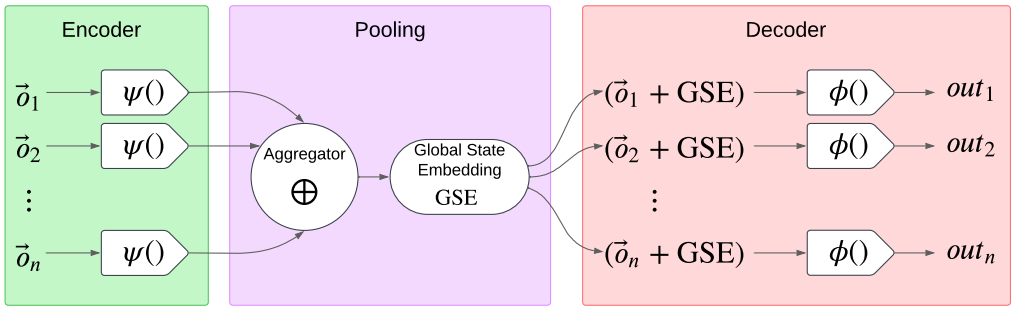


Fig. 3.2.: Visualization of the *encoder-pooling-decoder* architecture, the figure was created by the author. Inputs, e.g., observations of agents, are independently encoded by $\psi()$ and aggregated using a pooling method $\oplus$, resulting in the GSE. The GSE is then concatenated with the input observation of the agent for which the embedding is computed. This combined representation is passed through a decoder network $\phi()$, producing individually embedded representations for each input element.

3.4 Deep Learning

In this section, concepts of deep learning are introduced, with a focus on architectures relevant to multi-agent systems. Techniques for formatting and augmenting input data are discussed, as well as the basic principles of neural networks, followed by their adaptation to graphs and transformers.

3.4.1 Neural Networks

NNs are a machine learning method inspired by the structure of the human brain and its neurons [71]. Neural networks serve as a foundational model for more advanced machine

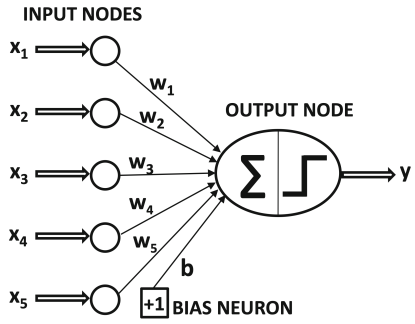


Fig. 3.3.: Visual representation of an artificial neuron [71].

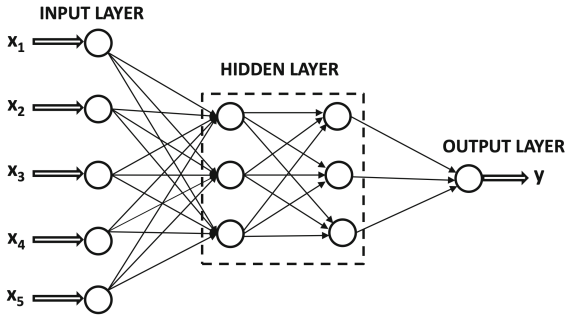


Fig. 3.4.: Feed-forward architecture with one output and two hidden layers [71].

learning techniques, it is therefore essential to have a good understanding of the concepts of NNs.

Artificial neurons. Artificial neurons are the building blocks of neural networks. Similar to biological neurons, they receive multiple input signals and produce an output signal (see Figure 3.3). Each neuron receives inputs $x_1, x_2, \ldots, x_n$, each associated with a corresponding weight $w_1, w_2, \ldots, w_n$. A bias term b is also introduced. The neuron's output is computed by applying a nonlinear activation function σ to the weighted sum of the inputs plus the bias [71].

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

The function σ, known as the activation function, is typically a Sigmoid, Tanh, or ReLU [71] (see Figure 3.5).

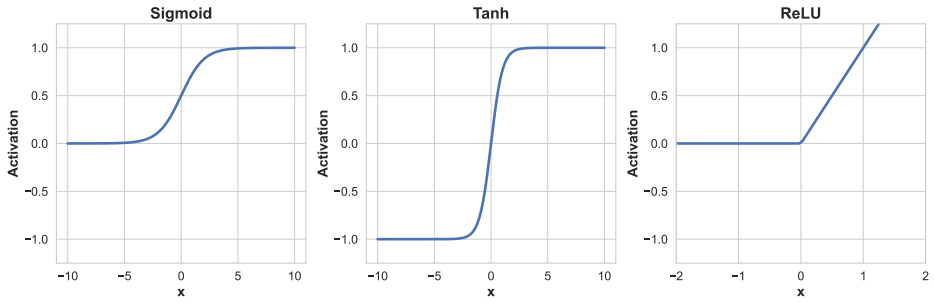


Fig. 3.5.: Overview of activation functions. Figure created by the author.

NN architecture. A typical architecture for neural networks is the feed-forward network [71]. It contains an input layer that receives the initial data and an output layer that produces the final result. The number and configuration of neurons in the output layer can be adapted to specific tasks. For example, regression tasks may require a single output neuron, while

classification tasks typically use one neuron per class. The number and size of the hidden layers, those between the input and output layers, are flexible and determine the complexity of the model. Figure 3.4 illustrates a simple feed-forward network with two hidden layers of size 3 and one output neuron.

Learning and training. Training a neural network generally involves two phases: *forward propagation*, where inputs are passed through the network to compute the output, and *backward propagation*, where the error between the predicted and target outputs is used to compute gradients via the chain rule [72]. These gradients are used to optimize the network's learnable parameters. Optimization techniques such as SGD enhance this training process by improving how weight updates are calculated and applied [73]. These concepts form the basis of deep learning. However, a discussion in greater detail is omitted as this would go beyond the scope of this thesis.

3.4.2 Graph Neural Networks

Graphs offer a powerful way to represent data found in diverse domains such as social networks, protein structures, and transportation systems [74]. In contrast to grid-structured data, e.g., images or sequences, graphs are non-Euclidean. Nodes can have varying numbers of neighbors and are unordered. This irregularity poses challenges for traditional machine learning methods, which typically require fixed-size, ordered inputs.

A graph is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges. Each node $i \in \mathcal{V}$ is associated with a d-dimensional feature vector $\vec{h}_i \in \mathbb{R}^d$, representing its initial input in the GNN.

Conventional operations, such as convolutions or multilayer perceptrons, rely on fixed-size, ordered inputs. Thus, they cannot be directly applied to graph-structured data [75]. To address these limitations and harness the representational power of graphs, GNNs have been developed.

GNNs operate by iteratively aggregating and transforming information from each node's local neighborhood in a way that respects both the graph's topology and permutation invariance. A general formulation, adapted from [74], is:

$$\vec{h}_i' = \phi\left(\vec{h}_i, \oplus_{j \in \mathcal{N}_i}\left(\psi(\vec{h}_j)\right)\right). \tag{3.6}$$

Here, $\vec{h}_i'$ denotes the updated embedding of node i. This embedding is computed by aggregating information from the local neighborhood $\mathcal{N}_i$ of $\vec{h}_i$ and combining it with the node's original representation $\vec{h}_i$. The function $\psi()$ applies a learnable transformation to

each neighbor's embedding $\vec{h}_j$, this can be implemented, for example, as a neural network layer that encodes the information of each neighbor of node i.

The operator $\oplus$ is a permutation-invariant aggregation function, such as summation or averaging, ensuring that the order of neighbors does not affect the result. The function $\phi()$ then combines the aggregated information with the node's current features, typically using a NN. Equation (3.6) provides a general framework underlying many GNN architectures, which extend it with mechanisms such as graph convolutions [76] or attention mechanisms [11].

Relation to MARL. In MARL, the set of agents can be modeled as nodes in a graph, either as a fully connected graph in a centralized setting or with communication protocols enforced by restricting each agent's neighborhood. Each agent i corresponds to one node in the graph, and its associated $\vec{h}_i$ is the observation $\vec{o}_i$ of agent i. Notice how similar the basic GNN paradigm in Equation 3.6 is to the encoder-pooling-decoder architecture shown in Equation 3.5.

3.4.3 Transformers

The concept of transformers was first introduced in the paper *"Attention Is All You Need"* [12], marking a breakthrough in NLP applications. This advancement also reached the general public through chatbots such as ChatGPT by OpenAI [5] and Gemini by Google [77].

The core mechanism of the transformer is the *self-attention* mechanism [12]. Self-attention enables transformers to process input elements in parallel while capturing dependencies and relationships between them. Given an input matrix $\mathbf{X} \in \mathbb{R}^{n \times d_x}$ consisting of n elements or tokens of dimensionality d_x, self-attention first computes three projections of the input, the query $\mathbf{Q}$, key $\mathbf{K}$, and value $\mathbf{V}$, using learnable weight matrices $\mathbf{W}$:

$$\mathbf{W}_Q \in \mathbb{R}^{d_x \times d_k}, \quad \mathbf{W}_K \in \mathbb{R}^{d_x \times d_k}, \quad \mathbf{W}_V \in \mathbb{R}^{d_x \times d_v}$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q \in \mathbb{R}^{n \times d_k}, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K \in \mathbb{R}^{n \times d_k}, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V \in \mathbb{R}^{n \times d_v}$$

The final attention output is computed using scaled dot-product attention:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \in \mathbb{R}^{n \times d_v} \tag{3.7}$$

Here, d_k is the dimensionality of the key vectors. Scaling by $\sqrt{d_k}$ helps to prevent too large dot products, which can destabilize training. Equation 3.7 enables the model to compute

attention scores for all positions in parallel, letting each input element attend to every other element. This allows learning global dependencies, regardless of sequence order.

Intuition behind self-attention. A useful way to understand self-attention intuitively is through the query-key-value analogy [78, 79]: Any element in the input, for example, a word[1] in a sentence, is trying to determine how much attention it should pay to the other words to understand its meaning in the context of the sentence. Each element generates a query vector to "ask" questions about the importance of other inputs, while simultaneously producing a key vector holding information about itself. The attention score between a query and a key can be seen as a relevance measure, or informally: "How relevant are you to me?"

Attention as a weighted sum. The output of scaled dot-product attention, as defined in Equation 3.7, can be interpreted as a weighted sum of the value vectors. The term $\text{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)$ computes attention weights that express the relevance of each key to the corresponding query. The softmax ensures these weights are positive and sum to 1 [12]. This mechanism enables the model to focus on the most relevant information in the input [79].

Multi-head attention. As an extension to self-attention, transformers can use multiple sets of queries, keys, and values, referred to as attention heads. Self-attention is computed independently in each head, enabling the model to capture different types of relationships between input elements. The outputs of all heads are then concatenated and passed through a learned output projection matrix $\mathbf{W}^O$. Formally, for m heads, each head i computes:

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i)$$

The final multi-head attention output is then:

$$\text{MultiheadAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_m)\mathbf{W}^O.$$

This final transformation allows the model to simultaneously consider the information captured by each attention head [12]

Transformer for MARL. In this work, self-attention is leveraged in its adaptations to sets (subsection 3.5.4) and graphs (subsection 3.5.3), to compare their respective performance in learning from agent environments. Instead of words in a sentence, MARL deals with agents in an environment. In the original transformer paper [12], positional encoding was introduced to provide word order in sequences, which is necessary for NLP tasks.

[1] In practical NLP applications, inputs are typically tokenized into subword units or tokens rather than full words. However, for simplicity and readability, the term "word" will be used throughout this thesis.

However, in a MARL setting, positional encoding is intentionally omitted to preserve the permutation-invariant properties of self-attention, since agent identity or order should not impact the outcome.

Positional encoding. Neural networks typically treat input data as sets of independent elements, which poses challenges when processing structured or sequential data, like text or images, where the order of elements carries meaning. To address this, *positional encoding* techniques embed information about the position or order of each element within a sequence. These encodings are added to or combined with the input representations, enabling the network to learn and incorporate sequential dependencies [80]. In contrast, agents in an environment are unordered, and permutation invariance is required. Introducing positional information directly contradicts the principle of permutation invariance. Therefore, positional encodings must be avoided entirely in PI MARL settings.

3.5 PI Embedding Methods

This section presents PI embedding methods that serve as the basis for embedding agent observations in a centralized critic network. Each method is introduced based on the structure and logic of its original publication, however, the notation is adapted to remain consistent with the conventions used throughout this thesis. Furthermore, certain symbols are modified to better reflect the MARL setting, for instance, inputs are directly denoted as agent observations where appropriate.

Default embedding – concatenation The most straightforward approach to processing centralized agent observations is by concatenation. Each agent's observation $\vec{o}$ is concatenated with those of the other agents in the environment. For example, in an environment with n agents, where each agent has a feature vector of size $1 \times d$, this results in a global observation matrix of size $n \times d$. Therefore, the size of the global input grows linearly with the number of agents n. Since the dimensionality of the concatenated vector directly depends on the number of agents, a corresponding MLP must be specifically designed for that particular agent count. Any change in the number of agents requires retraining the network, making this approach dependent on the number of agents [23].

Another drawback of concatenation is its inability to exploit the permutation-invariance, which is a key property in the homogeneous multi-agent environments explored in this thesis. As a result, concatenation is generally unsuitable for scenarios with a large or dynamic number of agents [23]. Regardless of these limitations, concatenation remains widely used in MARL whenever a centralized actor or critic is trained [81]. However, for homogeneous settings involving a large and possibly varying number of agents, this approach becomes impractical. For this reason, alternative embedding methods are explored in this chapter. Concatenation serves as a baseline against which these alternative methods are compared. The goal is to identify embeddings that match or exceed the performance of concatenation

while offering permutation invariance and the ability to generalize to varying numbers of agents.

3.5.1 DeepSets

DeepSets [9] is an architecture for neural networks designed to process sets using deep neural networks. The DeepSet architecture achieves PI of the set elements by first encoding each element $x \in X$ independently with an encoder NN $\psi()$, and then aggregating the encoded elements by summing them. After the summation, $\phi()$ represents another NN that produces the final output based on the embedded representation:

$$f_{\text{DS}}(X) = \phi\left(\sum_{x \in X} \psi(x)\right) \tag{3.8}$$

It is important to note that while the summation operation is invariant to the order of elements, it is sensitive to the number of elements in the set, as the magnitude of the output increases with the number of elements. This is a key consideration in settings where the number of agents may vary. In the original work [9], deep neural networks were employed for both the $\psi()$ and $\phi()$, and training was performed using the Adam optimizer. DeepSet approaches (DS) has been successfully applied across a range of domains, including point cloud classification, population statistics estimation, set expansion, and outlier detection in sets [9].

DeepSets in MARL In the paper "Deep Reinforcement Learning for Swarm Systems" [23], the authors extended the DS framework [9] for use in MARL. In this setting, the input elements x correspond to the local agent's observation $\vec{o} \in O$ within the environment. Instead of summing the embedded agent states, a mean operation is used. In their experiments, mean embedding outperformed max pooling. Mean and max embeddings also have the advantage of being invariant not only to the order of agents, but also to the number of agents, in contrast to summation.

$$f_{\text{DS\_mean}}(O) = \phi\left(\frac{1}{|O|}\sum_{\vec{o} \in O} \psi(\vec{o})\right)$$

A further optimization for MARL involves explicitly adding each agent's local observation to the embedded global state representation. After computing the mean embedding of the global state, agent i's local observation is concatenated with the global embedding as follows:

$$f_{\text{SwarmRL}}(O, \vec{o}_{\text{i}}) = \phi\left(\left(\frac{1}{|O|}\sum_{\vec{o} \in O} \psi(\vec{o})\right) \,||\, \vec{o}_i\right) \tag{3.9}$$

This structure follows the common encoder-pooling-decoder architecture, which is explained in more detail in paragraph 3.3 and subsection 3.3.2. In the following, for brevity, the approach described in "Deep Reinforcement Learning for Swarm Systems" [23], as shown in Equation 3.9, is referred to simply as SwarmRL. This architecture allows each agent to access both its own observation and a shared, permutation-invariant representation of the entire agent environment.

3.5.2 GraphSAGE

GSAGE is an extension of classical GNN and GCN architectures. It addresses the challenge of learning representations for nodes in an *inductive* setting, i.e., generalizing to unseen nodes and unseen graph structures. This is unlike traditional GCN models, which are typically designed for a fixed graph and struggle to generalize to unseen nodes or graph structures. As a result, GSAGE is suitable for dynamic, previously unseen graphs, as commonly found in real-world applications. It achieves competitive performance and outperforms methods such as DeepWalk [82] on various benchmark datasets.

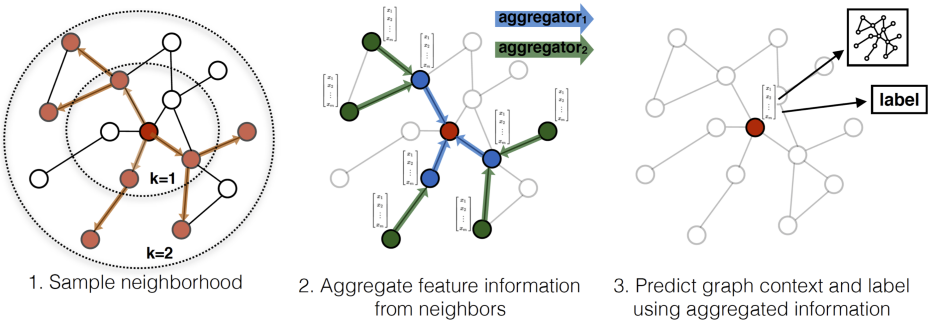


Fig. 3.6.: Visual depiction of GSAGE. 1. A fixed-size neighborhood of each node is sampled. The depth and number of sampled neighbors can be adjusted. 2. Each neighborhood is aggregated using an aggregator function, e.g., mean or max pooling. 3. The resulting embedding can then be used for downstream tasks such as node classification. Figure taken from [10].

Algorithm. GSAGE operates by recursively sampling fixed-size neighborhoods and aggregating node features. For each layer $k \in \{1, \ldots, l\}$ and each node $v \in \mathcal{V}$, the feature vectors of its neighbors $\mathcal{N}(v)$ are aggregated using a pooling function $\oplus_k$, resulting in an aggregated vector $\vec{h}^k_{\mathcal{N}(v)}$. This is concatenated with the node's own representation from the previous layer, $\vec{h}^{k-1}_v$, and passed through a feed-forward layer with learnable weights $\mathbf{W}^k$ and an activation function σ. The output is then normalized to be unit norm. The complete pseudocode is provided in algorithm 2, and a visual overview can be seen in Figure 3.6.

Algorithm 2: Embedding method of GSAGE [10], notation changed.

Input : Graph $(\mathcal{V}, \mathcal{E})$; input features $\{\vec{o}_v \mid v \in \mathcal{V}\}$; depth l; weight matrices $\mathbf{W}^k, \forall k \in \{1, \ldots, l\}$; non-linearity σ; differentiable aggregator functions $\oplus_k, \forall k \in \{1, \ldots, l\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\vec{h}'_v$ for all $v \in \mathcal{V}$

$\vec{h}_v^0 \leftarrow \vec{o}_v, \quad \forall v \in \mathcal{V}$

for $k \leftarrow 1$ **to** l **do**

  for $v \in \mathcal{V}$ **do**

    $\vec{h}^k_{\mathcal{N}(v)} \leftarrow \oplus_k \left(\{\vec{h}_u^{k-1} \mid u \in \mathcal{N}(v)\} \right)$

    $\vec{h}_v^k \leftarrow \sigma \left(\mathbf{W}^k \cdot \text{CONCAT} \left(\vec{h}_v^{k-1}, \vec{h}^k_{\mathcal{N}(v)} \right) \right)$

  $\vec{h}_v^k \leftarrow \vec{h}_v^k / \|\vec{h}_v^k\|_2, \quad \forall v \in \mathcal{V}$

$\vec{h}'_v \leftarrow \vec{h}_v^l, \quad \forall v \in \mathcal{V}$

Aggregator functions. The GSAGE paper [10] introduces the concept of *learnable aggregator functions* for combining information from a node's local neighborhood. However, most aggregation operations used, such as mean and max pooling, are non-parametric and therefore do not have trainable parameters. The learnable components are instead located in the feed-forward layers applied after aggregation. An exception is the Long Short Term Memory Network (LSTM) aggregator, which introduces trainable parameters as part of its sequence model. A challenge with using LSTMs is their lack of PI, as the ordering of neighbors influences the result. To mitigate this, GSAGE randomly permutes the order of neighbors during training, introducing a form of order invariance. For the mean aggregator, the model is further simplified by averaging over both the node's own features and those of its neighbors, without the concatenation step.

3.5.3 Graph Attention Networks

The GAT architecture [11] advances the concept of GNNs (see subsection 3.4.2) by introducing a self-attention mechanism. Instead of aggregating all neighboring node features uniformly, as is the case in GSAGE, the importance of each neighboring node is taken into account. This allows the model to focus more on relevant parts of the graph during training. Importantly, the attention mechanism used in GAT differs from the dot-product attention common in transformers [12]. Rather, GAT employs a learned attention mechanism specifically designed for graphs.

Algorithm. The input of the graph attention layer for N nodes with f features per node is the set of node features $H = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}, \vec{h}_i \in \mathbb{R}^f$. The attention mechanism operates on the neighborhood $\mathcal{N}_i$ of each node i. Each node feature is first linearly transformed by a shared weight matrix $\mathbf{W} \in \mathbb{R}^{f' \times f}$ into the latent space of dimension f'. The importance

of a neighboring node j to node i is computed using a shared feed-forward attention layer parameterized by $\vec{a}^\top \in \mathbb{R}^{1\times 2f'}$, followed by a LeakyReLU nonlinearity. This mechanism scores the concatenated feature pairs by computing a scalar value representing the importance of the neighbor. The complete equation to compute the importance of node j for node i is:

$$e_{ij} = \text{LeakyReLU}\left(\vec{a}^\top \left[\mathbf{W}\vec{h}_i \,\|\, \mathbf{W}\vec{h}_j\right]\right)$$

The attention coefficients e_{ij} are subsequently normalized across the neighborhood using a softmax function:

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k\in\mathcal{N}_i}\exp(e_{ik})}. \tag{3.10}$$

The updated embedding for node i is then computed as:

$$\vec{h}'_i = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}\mathbf{W}\vec{h}_j\right). \tag{3.11}$$

The graph attention layer's final output is the set of transformed node features $H' = \{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\}$. This process can be viewed as a weighted mean pooling of a node's neighborhood, where the weights are the learned attention scores α_{ij}. The function σ in Equation 3.11 is an optional nonlinearity.

Multi-head attention. GAT [11] can be extended with multi-head attention. With m attention heads, the final representation is computed by averaging over the outputs of each head. Concatenation can also be used instead, e.g., if it is not a final prediction layer:

$$\vec{h}'_i = \sigma\left(\frac{1}{m}\sum_{k=1}^{m}\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k\vec{h}_j\right). \tag{3.12}$$

GATv2. GATv2 [55] is an adaptation of GAT with the key alteration that the attention scores depend on both the source and target node features. In contrast to GAT, which applies the transformation $\mathbf{W}$ before feature combination, GATv2 first combines the raw features, applies $\mathbf{W}$, and then a LeakyReLU:

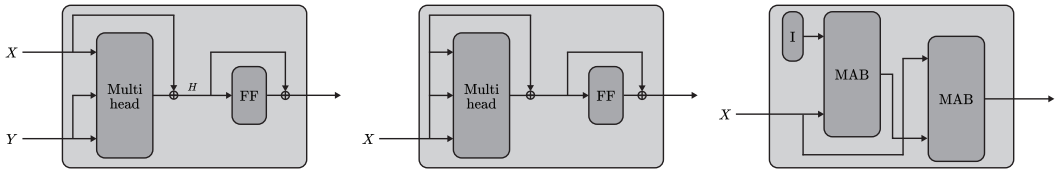
$$\text{GAT[11]:}\quad e_{ij} = \text{LeakyReLU}\left(\vec{a}^\top \left[\mathbf{W}\vec{h}_i \,\|\, \mathbf{W}\vec{h}_j\right]\right)$$

$$\text{GATv2[55]:} \quad e_{ij} = \vec{a}^{\top} \text{LeakyReLU}\left(\mathbf{W}\left[\vec{h}_i \,\|\, \vec{h}_j\right]\right) \tag{3.13}$$

This revised formulation prevents the attention scores from collapsing into a single-layer solution, potentially increasing the expressive power of the model. Meaning it can capture the relationships in the graph between neighbors better than GAT. Importantly, GATv2 maintains the same time complexity as GAT [55].

3.5.4 SetTransformer

The SetTransformer [39] architecture uses multi-headed self-attention specifically designed for learning from set structured data. It satisfies PI and handles variable-sized inputs by relying solely on self-attention mechanisms without any position-dependent features such as positional encoding. In the following, the key components of the SetTransformer will be outlined: Set Attention Block (SAB), Induced Set Attention Block (ISAB), and Pooling by Multiheaded Attention (PMA).



(a) Multihead Attention Block **(b)** Set Attention Block **(c)** Induced Set Attention Block

Fig. 3.7.: Architectural overview of compontents proposed by SetTransformer [39]. Figure taken from [39].

Set Attention Block. The SAB is designed similarly to the encoder architecture in [12], but lacks positional encoding and dropout layers.

$$\text{SAB}(\mathbf{X}) = \text{MAB}(\mathbf{X}, \mathbf{X}) \tag{3.14}$$

$$\text{MAB}(\mathbf{X}, \mathbf{Y}) = \text{Norm}(\mathbf{H} + \text{rFF}(\mathbf{H})) \quad \text{with} \quad \mathbf{H} = \text{Norm}(\mathbf{X} + \text{MultiheadAttention}(\mathbf{X}, \mathbf{Y}, \mathbf{Y})) \tag{3.15}$$

In SAB, Norm denotes layer normalization, and rFF is a row-wise feedforward network, typically a MLP with a ReLU activation. This means rFF is applied independently to each element in the input set. The function $\text{MultiheadAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ computes a standard scaled dot-product multihead attention, as shown in paragraph 3.4.3. The SAB applies self-attention to the input set $\mathbf{X} \in \mathbb{R}^{n \times d_x}$, where each element attends to all elements in the set, including itself. The SAB is implemented using a Multiheaded Attention Block (MAB) with $\mathbf{X}$ passed as both the query and the key/value inputs, i.e., $\mathbf{X} = \mathbf{Y}$. The main downside of SAB blocks is their quadratic complexity, which quickly becomes computationally expensive.

Induced Set Attention Block. To address the quadratic complexity of SAB, the ISAB introduces a set of m inducing points $\mathbf{I} \in \mathbb{R}^{m \times d}$. These inducing points are trainable parameters. The ISAB reduces the complexity to $O(nm)$, with $n > m$, while still capturing the global structure of the set through attention with the inducing points. The ISAB block is computed as follows:

$$\text{ISAB}(\mathbf{X}) = \text{MAB}(\mathbf{X}, \text{MAB}(\mathbf{I}, \mathbf{X}))$$

The inducing points $\mathbf{I}$ first attend to the input set $\mathbf{X}$. Then, another attention block is performed similarly to the conventional SAB, although here, $\mathbf{X} \neq \mathbf{Y}$ in Equation 3.15. This structure can be viewed as an autoencoder, where the input is first projected into a lower-dimensional space and then reconstructed. The inducing points embed the global structure of the set to represent the inputs $\mathbf{X}$. Both SAB and ISAB remain PI and PE [39].

Pooling by Multiheaded Attention. Instead of mean or max pooling (see section 3.3), SetTransformer uses Pooling by Multiheaded Attention. $\mathbf{Y}$ is some input set, similar to $\mathbf{X}$, but in practice PMA is used after self attention, thus $\mathbf{Y} \in \mathbb{R}^{n \times d}$. PMA uses k learnable "seed vectors" $\mathbf{S} \in \mathbb{R}^{k \times d}$ as queries. The output of PMA is a set of k vectors with dimension d. Having multiple seed vectors can be useful for generating multiple outputs, but in MARL, typically only one global pooled representation of the global state is required. The PMA is defined as:

$$\text{PMA}(\mathbf{Y}) = \text{MAB}(\mathbf{S}, \text{rFF}(\mathbf{Y}))$$

Since PMA uses an MAB, it relies on multiheaded self-attention. Self-attention can be seen as a weighted mean of the value vectors $\mathbf{V}$. In PMA, since $\mathbf{V} = \mathbf{Y}$, the weighted mean of $\mathbf{Y}$ is computed. This is similar to GAT discussed in subsection 3.5.3, though the attention mechanisms differ. While GAT uses a learned feedforward scoring function [11], SetTransformer employs the classic self-attention mechanism from [12].

Chaining the blocks together. By chaining the defined blocks, one can design an architecture resembling the encoder-pooling-decoder concept (see paragraph 3.3). For example, an encoder consisting of two ISAB blocks maps the input $\mathbf{X}$ to $\mathbf{Y} \in \mathbb{R}^{n \times d}$. A pooling layer with a PMA block ($k = 1$) then reduces $\mathbf{Y}$ to a single vector $\mathbf{Z}$ of shape $1 \times d$. Finally, the decoder, comprised of two SABs followed by a linear transformation rFF, transforms the output to the desired dimension d_{rff} [39]:

$$
\begin{aligned}
\text{SetTransformer}(\mathbf{X}) &= \phi_{sab}\left(\oplus_{\text{PMA}}\left(\psi_{isab}(\mathbf{X})\right)\right) \\
&\text{with} \\
\psi_{isab}(\mathbf{X}) &= \text{ISAB}(\text{ISAB}(\mathbf{X})) = \mathbf{Y} \in \mathbb{R}^{n \times d} \\
\oplus_{\text{PMA}}(\mathbf{Y}) &= \text{PMA}(\mathbf{Y}) = \mathbf{Z} \in \mathbb{R}^{1 \times d} \\
\phi_{sab}(\mathbf{Z}) &= \text{rFF}(\text{SAB}(\text{SAB}(\mathbf{Z}))) \in \mathbb{R}^{1 \times d_{rff}}
\end{aligned}
$$

4 Methodology

4.1 Multi-Agent PPO

The specific MAPPO implementation used in this thesis is adapted from the PyTorch implementation of MAPPO [83]. The main modification to the training loop involved removing unneeded components, such as the replay buffer, and integrating the custom embedding logic directly into the architecture.

Critic network. The embedding techniques for the critic network $V_\phi(\vec{o})$ are used for value estimation, needed to compute the advantage function $\hat{A}_t$ in the CLIP objective (see Equation 3.3). This critic network can be replaced with different architectures without requiring additional modifications to MAPPO, as long as the architecture implements a mapping from the observation set $O \in \mathbb{R}^{n \times d_o}$ to a set of value estimates:

$$V_\phi : \underbrace{\mathbb{R}^{n \times d_o}}_{n \text{ observations}} \rightarrow \underbrace{\mathbb{R}^{n}}_{n \text{ value estimates}}$$

The output is an n-dimensional vector because the critic computes one scalar value estimate per agent.

Actor network. The PyTorch MAPPO implementation uses a stochastic policy. To define it, the actor network must output the parameters of a distribution for each possible action. Specifically, a bounded Tanh-Normal distribution is used, parameterized by a mean and standard deviation [83]. Consequently, the output of the actor network has a dimensionality of twice the number of actions a. The input to the policy network is either a single observation or all observations, corresponding to CTDE or CTCE, respectively.

$$\text{CTDE:} \quad \pi_\theta : \underbrace{\mathbb{R}^{d_o}}_{\text{observation of the agent}} \rightarrow \underbrace{\mathbb{R}^{2 \cdot a}}_{a \text{ means and } a \text{ standard deviations}}$$

$$\text{CTCE:} \quad \pi_\theta : \underbrace{\mathbb{R}^{n \times d_o}}_{n \text{ observations}} \rightarrow \underbrace{\mathbb{R}^{2 \cdot a}}_{a \text{ means and } a \text{ standard deviations}}$$

CTDE and CTCE. Since CTCE is less common than CTDE, it is only used as an extension in the experiment chapter of this thesis. In the environments, the observation space of each agent remains the same regardless of the number of agents. With the proposed PI embedding methods, it is possible to create a centralized policy, just like the centralized critic, by using the CTCE framework. With CTCE, the policy's input depends on the number of agents in the system, like the critic.

Implementation details. The hyperparameters of the PPO settings were taken from the original PyTorch implementation [83] of MAPPO as they performed very well for the *CONCAT* baseline (see section 4.3) in VMAS. For a full list of these hyperparameters, see Table A.1. For optimization of the CLIP loss, the ADAM optimizer[1] is used.

4.2 Vectorized Multi-Agent Simulator

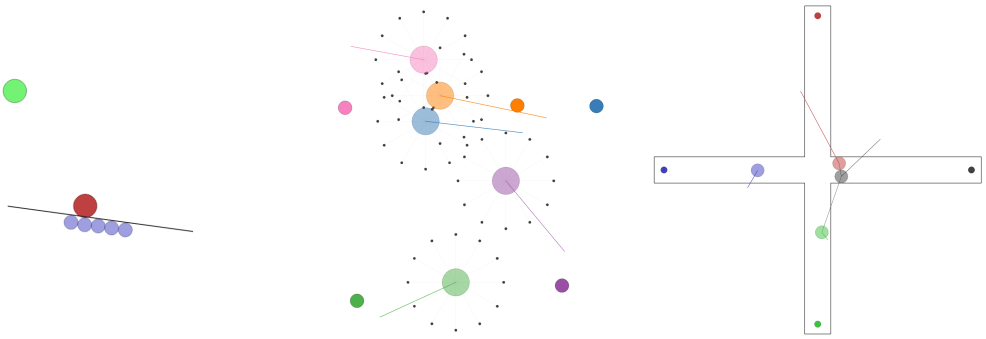
VMAS [30] is an open-source platform purpose-built for simulating multi-agent environments with a variety of different prebuilt MARL tasks. It can be used to model the behavior and interactions of multiple agents within a controlled and scalable environment. In this research, VMAS is utilized to model and test different actor-critic setups for MARL.

VMAS is especially useful for this thesis as it offers a lightweight, vectorized simulation environment for MARL. The framework's design is centered on scalability, flexibility, and ease of use, making it ideal for experimenting with various multi-agent systems scenarios. The scalability is achieved through efficient and parallelizable computations, made possible by the vectorized nature of VMAS. Since scalability is one of the aspects that are explored in this thesis, VMAS is well-suited for the task.

Ease of use of VMAS. The ease of use of VMAS is also an important aspect, with a common API that allows testing and iterating across different reward systems, observation models, and environment configurations. Changing the environment configurations is a central factor for this work, in order to try out different embedding methods of the agent observations. Multi-agent scenarios, all with the same underlying structure, can be seamlessly swapped out and tested in various experimental setups, making it easier to compare results and refine strategies. Written in PyTorch, VMAS is particularly useful for deep reinforcement learning tasks, as it supports efficient gradient-based optimization algorithms. One of the key strengths of VMAS is its ability to model decentralized policies for homogeneous agents, which is essential for this thesis. Additionally, VMAS's architecture makes it feasible to experiment with other multi-agent reinforcement learning algorithms. For example, algorithms like IPPO and CPPO can be tested alongside more conventional approaches like MAPPO.

[1] `https://docs.pytorch.org/docs/stable/generated/torch.optim.Adam.html`

Scenario diversity and vectorized representation. VMAS comes equipped with twenty one implemented multi-agent scenarios[2], each designed to test different aspects of multi-agent interactions. These scenarios provide a rich set of environments where agents can learn and adapt under different conditions, making VMAS a versatile tool for experimenting in different settings (see Figure A.1). As the name suggests, VMAS represents the agents as vectors. This is especially desirable for this work as it enables the use of different agent embedding types efficiently. While image-based representations, such as those used in the StarCraft Multi-Agent Challenge[29], might offer advantages in certain settings, VMAS's vectorized, continuous 2D space offers a more simplified and flexible framework for the goals of this thesis. Finally, VMAS has been well-tested and is a widely used codebase[3], further ensuring the robustness and reliability of the platform for the experiments. By using VMAS in this thesis, the simulations are both scalable and adaptable, allowing for prototyping, testing, and refinement of multi-agent reinforcement learning strategies. The platform's efficiency and the ability to scale to a large number of agents make it a suitable tool in the development and evaluation of MARL.



(a) Balance scenario **(b)** Navigation scenario **(c)** Multi-give-way scenario

Fig. 4.1.: The three VMAS environments used in this thesis. (a) In the balance scenarios, the blue agents try to balance a red dot on a line towards a green goal. (b) In the navigation environment, each agent has a goal which it tries to reach while avoiding collisions with other agents. The black dots represent lidar range data, enabling agents to detect nearby agents. (c) In the multi-give-way scenario, four agents must traverse an intersection while avoiding others to reach their goal. When agents are within a certain range, they communicate their positions with each other to avoid collision.

Action space. In VMAS, continuous action spaces can be used. For the scenarios chosen, the action is a two-dimensional force vector that determines the acceleration of an agent. Consequently, the actor network described in paragraph 4.1 requires an output dimension of 4, i.e, the distribution for each force direction defined by the mean and standard deviation.

[2] As of 27.04.2025.
[3] `https://github.com/proroklab/VectorizedMultiAgentSimulator`

4.2.1 Environments Suitable for Research Questions

The choice of environment is essential for answering the research questions formulated in section 1.3. To evaluate how well the embedding captures global and local relationships between agents, the selected environment must require cooperation among all agents.

Testing generalizability. To assess whether the policy can generalize across varying number of agents, the policy must be influenced by the agent count. For example, the *navigation* scenario requires the agent to avoid collisions while reaching their individual goals, which necessitates some cooperation. However, this cooperation is locally restricted to nearby agents only. Consequently, the learned policy remains largely the same regardless of the number of agents: avoid others and move toward the goal. Even the critic's value estimates are largely unaffected by agent count, as they depend only on collision avoidance and goal distance. Therefore, this environment is not well-suited for testing the generalizability of the actor or critic. The same limitation applies to other scenarios such as *discovery*, *dispersion*, and *transport*.

Variable agent count. Another requirement is that the environment must allow for a variable number of agents. Environments with a fixed agent count cannot be used to test aspects like generalizability or scalability of the methods. Additionally, some environments do not scale as well as others. It is also essential that the agents are homogeneous, and the environment is challenging to differentiate between approaches. The *balance* scenario meets all of these criteria best. It is a scalable and challenging environment in which agents must cooperate to balance an object on a line and move it toward a goal, starting from random positions. Since the agents must position themselves differently along the line, both the actor and critic are affected by the number of agents involved.

General performance evaluation. For general performance benchmarking, the *navigation* and *multi-give-way* scenarios are also used, as shown in Figure 4.1. The hyperparameters used for these environments are listed in Table A.2. Evaluating the methods across multiple environments increases the robustness of the experiment results.

Shared reward. In VMAS, it is possible to enable a shared reward by setting the hyperparameter $shared\_reward = True$. However, in VMAS this feature was not implemented as a truly shared reward. Instead, in some scenarios, such as *multi-give-way* and *navigation*, the reward is only partially shared. To address this, the rewards were adapted to ensure each agent receives the same shared reward. Specifically, the collision penalty of both scenarios was adjusted, as well as the passage collision penalty in the multi-give-way scenario, to ensure shared rewards in the environments.

4.2.2 Balance Scenario

In the balance scenario[4], a set of agents collaborates to transport a circle to a goal area located in the upper part of an environment. The agents spawn at random positions and need to balance a package, in the form of a red circle, on a horizontal line. Each agent obtains information from its point of view. The agent's observation is a vector with 16 dimensions, including its position ($2D$), velocity ($2D$), its relative position to the package ($2D$), and to the horizontal line ($2D$). The agent also knows the relative position of the package to the goal ($2D$) and the velocity of the package ($2D$). Regarding the horizontal line, the agent is provided with information about velocity ($2D$), angular velocity ($1D$), and rotation ($1D$). Based on this set of information, the agents need to synchronize their actions to maintain balance and guide the package, i.e., the red circle, which can roll of the horizontal line. The reward the agents receive is based on the distance of the package to the target, with all agents sharing the same reward. A positive reward is given if the package moves closer to the goal, and a negative one for being further away. There is also an additional penalty in case the package falls to the ground. The episodes terminate either if the package touches the floor, reaches the goal, or the maximum number of steps is reached.

4.2.3 Navigation Scenario

In the navigation scenario[5], a team of agents operates within a $2D$ space where each agent is assigned a goal position. The agents must move towards their respective targets while avoiding collisions with one another. At every timestep, each agent observes its state ($18D$), which includes its position ($2D$) and velocity ($2D$), as well as the relative position of its goal ($2D$). Agents also have access to lidar range data ($12D$), allowing them to sense nearby agents and obstacles. The reward is comprised of three components, and the final reward is shared across all agents:

- Position reward: The sum of individual agent rewards, where each agent is rewarded based on the change in distance to its goal. Agents receive a positive reward when moving closer and a negative reward when moving away.
- Final reward: A small bonus is granted when all agents have reached their goal area.
- Collision penalty: Agents receive a penalty if they collide with others, discouraging unsafe behavior. The sum of all individual penalties is the final collision penalty.

Episodes end when all agents have arrived at their goal positions, or when the maximum number of steps is reached. This way, the navigation scenario provides a challenging multi-agent setting for exploring goal-driven navigation and collision avoidance. The navigation

[4] https://github.com/proroklab/VectorizedMultiAgentSimulator/blob/main/vmas/scenarios/balance.py
[5] https://github.com/proroklab/VectorizedMultiAgentSimulator/blob/main/vmas/scenarios/navigation.py

scenario is a suitable environment for evaluating a method's performance, as it delivers a distinct setting from the primary balance scenario, used in testing. Since navigation requires only basic collision avoidance and no explicit cooperation between agents, it serves as a valuable test of the method's ability to adapt to different dynamics and objectives beyond the main environment used for evaluation.

4.2.4 Multi-Give-Way Scenario

In the multi-give-way scenario[6], four agents are positioned in a walled-in intersection. Each agent has a fixed goal located on another side of the intersection, requiring agents to coordinate their movements through the intersection without collisions. At each timestep, agents observe their state ($13D$), including position ($2D$), velocity ($2D$), relative position to their goal ($2D$), and the distance to their goal ($1D$). Additionally, agents observe the relative positions of other agents within a communication range, allowing limited situational awareness for coordination. This feature of the environment was intended by the creators of the environment, but was not implemented. Instead, it is implemented by the author of this thesis, giving an agent the relative position of the agents in the specified communication range ($6D$). If no agents are in range, the positions are zero.

The reward function encourages agents to reach their respective goals efficiently and safely, with the same three reward components as the navigation scenario. A positional reward incentivizes efficient movement to the respective goal, and a final reward is granted when all agents have reached their goal. Moreover, a collision penalty is imposed if agents collide. In addition to these rewards, a passage collision penalty was implemented by the author of this thesis to encourage the agents not to collide with the walls of their passage. In contrast to the other environments used, the reward components making up the complete reward were tuned to ensure the agents reach their goal while trying not to collide with the walls and each other. The initial set of reward parameters caused the agents to get stuck in deadlocks and did not incentivize reaching the goal sufficiently.

This scenario challenges agents to implicitly coordinate yielding and moving decisions in a constrained shared space, simulating interactions in real-world traffic. Unlike the navigation scenario, which focuses primarily on collision avoidance, the multi-give-way scenario requires cooperative behavior to prevent deadlocks and collisions in the environment, testing a method's ability to manage multi-agent interactions under spatial constraints and implicit communication.

[6] https://github.com/proroklab/VectorizedMultiAgentSimulator/blob/main/vmas/scenarios/multi_give_way.py

4.3 Concatenation Baseline

To provide a baseline for evaluating the proposed embedding strategies, a simple concatenation-based approach is used, referred to as *CONCAT* in this thesis. As elaborated in paragraph 3.5, this method constructs a global state representation by concatenating all agent observations into a single vector, which is then processed by a shared NN. The embedding function is defined as:

$$f_{\text{CONCAT}}(O) = \phi\left(\text{concat}(O)\right), \tag{4.1}$$

where O is the set of n observations $\vec{o} \in \mathbb{R}^{1\times d_o}$. The function $\phi()$ is a fully-connected NN that takes as inputs the concatenated vector produced by $\text{concat}(O) \in \mathbb{R}^{1\times(n\cdot d_o)}$ and outputs a single scalar value estimate. Internally, this scalar is then replicated n times to produce n value estimates, one per agent. Each agent, therefore, receives the same global value estimate, since the global state is equal for all agents, as it would be arbitrary to reorder the global state.

The *CONCAT* strategy is straightforward to implement and has shown strong empirical performance in various MARL tasks [81]. However, it exhibits two critical limitations: It is neither invariant to the number nor to the order of agents. The input to ϕ has a fixed dimensionality determined by the number of agents and the size of each agent's observation.

Despite these limitations, *CONCAT* serves as a strong and commonly used baseline in the literature. For the experiments, the size and depth of the MLP used in *CONCAT* will be tuned to fit the balance scenario best. This way, achieving comparable or better performance using methods that are PI and scalable to a varying number of agents represents a meaningful improvement over this baseline.

4.4 DeepSets-Based Invariant Embedding Methods

Application of SwarmRL. This thesis adopts the mean embedding architecture from SwarmRL [23], which is based on DeepSets [9] (see subsection 3.5.1). Experiments are conducted to evaluate whether mean embedding outperforms max embedding in the considered setting, as was observed in SwarmRL. Additionally, the effects of varying layer sizes and network depths are systematically explored.

Embedding Architectures This work compares three distinct network architectures inspired by DeepSets and SwarmRL for constructing critic networks in MAPPO. These

variants differ primarily in how the global state embedding is constructed and how local agent information is incorporated. The three approaches evaluated are called: *DS*, *DS_LOCAL*, and *DS_GLOBAL*.

Setup. The methods vary in their treatment of the local agent state and the global embedding:

- *DS* computes only the mean embedding over all agents, including the local agent, and does not concatenate the local state separately. This is the same as the originally proposed DeepSets method [9], only with mean embedding instead of summing the outputs of the encoder, as described in Equation 3.8.
- *DS_LOCAL* differs by computing the mean embedding over the other agents' states only, explicitly excluding the local agent's state from the pooled embedding, before concatenation. This way, each agent has a different GSE.
- *DS_GLOBAL* corresponds to the SwarmRL approach defined by Equation 3.9, where the local agent state is concatenated to the mean embedding of all agents.

The formal definitions are summarized in Table 4.1.

Embedding	Definition
DS	$f(O, \vec{o}_i) = \phi\left(\frac{1}{\lvert O\rvert}\sum_{\vec{o}\in O}\psi(\vec{o})\right)$
DS_GLOBAL	$f(O, \vec{o}_i) = \phi\left(\left(\frac{1}{\lvert O\rvert}\sum_{\vec{o}\in O}\psi(\vec{o})\right)\Vert\vec{o}_i\right)$
DS_LOCAL	$f(O, \vec{o}_i) = \phi\left(\left(\frac{1}{\lvert O\rvert}\sum_{\vec{o}\in O}\psi(\vec{o})\right)\Vert\vec{o}_i\right)$, where $\vec{o}_i \notin O$

Tab. 4.1.: Definitions of embedding approaches. Here, O denotes the set of agent states, $\vec{o}_i$ the agent's observation for which the value estimate is computed, ψ the embedding network applied to each agent state, and ϕ the decoder network. The symbol $\Vert$ denotes vector concatenation.

Evaluating the performance differences between these approaches provides insight into the importance of explicitly concatenating the local state with the pooled global embedding.

Moreover, the results can show the impact of excluding the local state from the global embedding (*DS_LOCAL*), which produces an embedding tailored to each agent's neighbors, whereas including the local state (*DS_GLOBAL*) creates a global state representation shared by all agents. However, excluding the local state could also have drawbacks, since the completeness of the global state is limited. An immediate disadvantage of excluding the local state is that the GSE has to be computed for every agent individually. For *DS_GLOBAL* and *DS*, it is sufficient to run $\frac{1}{\lvert O\rvert}\sum_{\vec{o}\in O}\psi(\vec{o})$ once and reuse it for each agent, representing a considerable optimization, which might help these methods to scale better than *DS_LOCAL*.

For all three methods, the encoder network ψ is shared and applied independently on each agent's observation. The decoder network ϕ differs only in its input dimensionality: for *DS*, the input dimension is the output dimension of the decoder (d_ψ), since it lacks the concatenated local observation. For *DS_GLOBAL* and *DS_LOCAL*, the input dimension is $d_\psi + d_o$ due to the concatenated local observation. Here, d_ψ denotes the hidden output dimension of the encoder network ψ, and d_o denotes the dimensionality of an individual agent's observation. The decoder ϕ outputs a scalar value estimate, like the *CONCAT* method. The value estimates vary per agent, for *DS_LOCAL* and *DS_GLOBAL*, since the function f is evaluated with each agent's local observation $\vec{o}_i$. *DS* operates like *CONCAT* in the sense that it has the same value estimate across all agents at a given time step, because $\vec{o}_i$ does not influence the embedding.

4.5 GNN-Based Invariant Embedding Methods

As described in subsection 3.4.2, GNN embeddings provide permutation- and agent-count-invariant techniques suitable for the MARL setting explored in this thesis. In total, three graph-based embedding methods are used, based on GraphSAGE and GAT, referred to as: *GSAGE*, *GAT*, and *GATv2*.

GSAGE, GAT, and GATv2 are all graph-based embedding methods operating on node features. The input observations $\vec{o} \in O$ of the agents are first transformed into initial node embeddings by the encoder network:

$$\vec{h} = \psi(\vec{o}), \tag{4.2}$$

before any processing occurs in the GNN framework.

This reflects the fact that these methods do not operate directly on the observation set $O \in \mathbb{R}^{n \times d}$, but rather on a transformed feature matrix $H \in \mathbb{R}^{n \times d'}$, derived from O by an MLP with output feature dimension d'.

All three embedding methods are implemented in Python using the PyTorch Geometric (PyG) package [84], which includes the basic NN layers corresponding to the algorithms behind GraphSAGE [10], GAT [11], and GATv2 [55]. This package differs slightly from the original formulations presented in the respective papers. These differences, along with specific changes in implementation, are described in this section. PyG is used due to its popularity, efficiency, and adaptability to the requirements of this thesis.

4.5.1 PyTorch Implementation – GraphSAGE

For GraphSAGE, there are minor deviations from the original algorithm (see algorithm 2). In the original formulation, the node embedding is computed as:

$$\text{Paper [10]:} \quad \vec{h}_i' = \mathbf{W}_2 \cdot \text{mean}\left(\mathcal{N}_i \cup \{\vec{h}_i\}\right) \quad \text{or} \quad \vec{h}_i' = \mathbf{W}_2 \cdot \text{concat}\left(\max\left(\mathcal{N}_i\right), \vec{h}_i\right) \tag{4.3}$$

For a more detailed formulation, refer to algorithm 2. The PyG implementation introduces a second weight matrix and treats each respective node $\vec{h}_i$ separately from its neighbors. Moreover, mean and max pooling no longer alter the computation, which simplifies the approach as there is no additional concatenation step for max pooling, and no extra union for mean pooling:

$$\text{PyG [84]:} \quad \vec{h}_i' = \mathbf{W}_1\vec{h}_i + \mathbf{W}_2 \cdot \text{mean}(\mathcal{N}_i) \quad \text{or} \quad \vec{h}_i' = \mathbf{W}_1\vec{h}_i + \mathbf{W}_2 \cdot \max(\mathcal{N}_i) \tag{4.4}$$

PyG also offers an option to project node features $\vec{h}_i$ and apply an activation before aggregation directly in the GSAGE layer. However, this feature is not used here. Instead, to maintain more control over the projection, the inputs to the *GSAGE* embedding are first processed through an MLP, i.e., the encoder network ψ. The size, depth, and necessity of this MLP are evaluated experimentally. This encoder acts as an explicit feature transformation, processing each input independently. Another difference from the original formulation is that, rather than concatenating, the aggregated neighborhood vector is summed with the local feature vector $\vec{h}_i$. Additionally, an output MLP, referred to as the decoder network ϕ, is applied after the GraphSAGE layer and the ReLU activation σ. The complete computation function for the *GSAGE* embedding used in this thesis is given as:

$$f_{GSAGE}(\mathcal{N}_i, \vec{h}_i) = \phi\left(\sigma\left(\text{Normalize}\left(\mathbf{W}_1\vec{h}_i + \mathbf{W}_2 \cdot \oplus_{j\in\mathcal{N}_i}\vec{h}_j\right)\right)\right) \text{ where } \vec{h}_i \notin \mathcal{N}_i, \tag{4.5}$$

In the equation, $\oplus$ denotes an aggregation function applied over the neighbors, such as mean or max pooling. The result of the linear transformations on both the node's features and the aggregated neighborhood features is normalized and passed through a non-linear activation function σ. The normalization step is performed using LayerNorm[7], in order to stabilize and scale the combined feature representation.

4.5.2 PyTorch Implementation - Graph Attention Network

GAT. The GAT implementation in PyG closely follows the original formulation introduced in [11], see subsection 3.5.3 for a detailed description. The primary difference lies in the computation of the attention coefficients. In PyG, the attention weight vector $\vec{a}^\top$ is split into two separate vectors corresponding to the source and target nodes. The same splitting is

[7] https://docs.pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html

Embedding	Definition
GSAGE	$f(\mathcal{N}_i, \vec{h}_i) = \phi\left(\sigma\left(\text{Normalize}\left(\mathbf{W}_1\vec{h}_i + \mathbf{W}_2 \cdot \oplus_{j\in\mathcal{N}_i}\vec{h}_j\right)\right)\right), \quad \text{where } \vec{h}_i \notin \mathcal{N}_i$
GAT	$f(\mathcal{N}_i, \vec{h}_i) = \phi\left(\sigma\left(\frac{1}{m}\sum_{k=1}^{m}\sum_{j\in\mathcal{N}_i}\alpha^k_{\text{GAT},ij}\mathbf{W}^k\vec{h}_j\right)\right), \quad \text{where } \vec{h}_i \in \mathcal{N}_i$
GATv2	$f(\mathcal{N}_i, \vec{h}_i) = \phi\left(\sigma\left(\frac{1}{m}\sum_{k=1}^{m}\sum_{j\in\mathcal{N}_i}\alpha^k_{\text{GATv2},ij}\mathbf{W}^k\vec{h}_j\right)\right), \quad \text{where } \vec{h}_i \in \mathcal{N}_i$

Tab. 4.2.: Formal definitions of the GNN-based embedding approaches used in this thesis. Each method defines an embedding function $f(\mathcal{N}_i, \vec{h}_i)$ that maps a node and its neighborhood to a learned representation. *GSAGE* aggregates neighborhood features via a mean or max pooling operation, followed by a linear transformation, normalization, non-linearity, and a final decoder MLP. In contrast, *GAT* and *GATv2* use multi-head attention mechanisms to weigh neighbor relevance with m heads. The key difference between *GAT* and *GATv2* lies in the placement of the non-linearity within the attention coefficients α_{ij}. All variants share a similar encoder-decoder structure, but differ in how neighborhood information is incorporated and weighted. The GNN methods operate on the node embeddings, which are computed from the observations of the agents with the encoder network, as $\vec{h}_i = \psi(\vec{o}_i)$.

applied to the weight matrix. Furthermore, instead of concatenating, the transformed node features are summed. The attention coefficient e_{ij} between node i and node j is computed as:

$$\text{PyG [84]:} \quad e_{ij} = \text{LeakyReLU}\left(\vec{a}_s^\top\mathbf{W}_s\vec{h}_i + \vec{a}_t^\top\mathbf{W}_t\vec{h}_j\right)$$
$$\text{Paper [11]:} \quad e_{ij} = \text{LeakyReLU}\left(\vec{a}^\top\left[\mathbf{W}\vec{h}_i \,\|\, \mathbf{W}\vec{h}_j\right]\right)$$

GATv2. GATv2 [55] modifies the attention mechanism by applying the non-linearity before the attention weight vector multiplication, allowing for more expressive attention functions. Similar to the GAT implementation, the weight matrix is split into source and target components, and the results are summed rather than concatenated. For GATv2, the attention coefficient e_{ij} is computed as:

$$\text{PyG [84]:} \quad e_{ij} = \vec{a}^\top\,\text{LeakyReLU}\left(\mathbf{W}_s\vec{h}_i + \mathbf{W}_t\vec{h}_j\right)$$
$$\text{Paper [55]:} \quad e_{ij} = \vec{a}^\top\,\text{LeakyReLU}\left(\mathbf{W}\left[\vec{h}_i \,\|\, \vec{h}_j\right]\right)$$

Final embedding. To construct the final critic embedding, encoder and decoder networks are used in the same way as in *GSAGE*. Additionally, the multi-head variant of GAT and GATv2 is used, as described in Equation 3.12, resulting in the final graph attention embeddings shown in Table 4.2. An additional normalization layer for the graph attention techniques is omitted, as it did not impact performance and the attention coefficients are already normalized via the softmax function in α (see Equation 3.10).

4.5.3 GNN Architecture Consideration

Adaptation for MARL. The GAT and GSAGE architectures are naturally PI and can handle varying numbers of agents, eliminating the need for additional adaptations in the context of multi-agent systems. In this work's setting, the critic operates in a centralized manner, with access to the observations from all agents. This allows the construction of a fully connected graph, where each node represents an agent and the node's features correspond to the agent's observations. Since the graph is fully connected, information from all agents can be aggregated in a single message-passing step. Therefore, a single layer of GAT, GATv2, GSAGE is sufficient for this setting.

Architecture experiments. The configuration of the encoder network, in terms of depth and output dimension d', will be explored in the experiment chapter. The decoder adheres to the same MLP structure used in the MLP-based approaches. For GSAGE, the embedding algorithm simplifies in the case of fully connected graphs with completely sampled neighborhoods. It becomes almost identical to the *DS_Local* method. Here, $\mathcal{N}_i$ refers to the neighborhood of $\vec{h}_i$, excluding the node itself, resembling O in Table 4.1. The differences lie primarily in the additional normalization applied in GSAGE, and the configuration of the non-linearity σ and weight matrix $\mathbf{W}$. Both methods use mean or max embeddings and concatenate the local observation to the aggregated embedding. In GSAGE, however, the local observations are preprocessed through the encoder network before being concatenated and pooled. GAT and GATv2 differ only in their attention mechanisms, and it will be informing to compare their performance in the same setting. Unlike GSAGE, these methods do not explicitly concatenate the embedded global state with the local observation. Instead, the neighborhood $\mathcal{N}_i$ implicitly includes the local observation in the attention-weighted aggregation. This enables the model to learn the importance of local observations for the prediction task by adjusting its attention weights accordingly. For both GAT and GATv2, the optimal number of attention heads will also be investigated in the experiments.

GNN as encoder-pooling-decoder. As shown in Table 4.2, the GNN methods still follow the encoder-pooling-decoder architecture. However, the message-passing aggregation of the graph techniques builds a custom GSE for each node, similar to *DS_LOCAL*, by incorporating pooling and constructing the GSE directly within the message-passing framework.

4.6 Transformer-Based Invariant Embedding Methods

The three transformer-based embedding architectures explored in this thesis will be referred to as: *SetTransformer* (see subsection 3.5.4), *SABTransformer*, and *ISABTransformer*. The *SABTransformer* and *ISABTransformer* make use of self-attention and induced self-attention, like the *SetTransformer*, but in a simplified architecture. Most notably, they do not

have a separate pooling mechanism but solely use self-attention or induced self-attention for the embedding.

Attention mechanism. The multi-headed attention mechanism in the SetTransformer differs from approaches like GAT. Instead of replicating the input dimension for each attention head, the SetTransformer-based methods split the input dimension evenly across the specified number of heads. Consequently, the model dimension must be divisible by the number of heads. For example, with a model dimension of 32 and two attention heads, each head operates on 16 dimensions. The outputs of all heads are then concatenated to form the final attention output.

4.6.1 Agent-Invariant SetTransformer

A requirement in this thesis is that the model's output should scale with the number of agents, i.e., for each agent, an individual output must be produced. However, the standard *SetTransformer* produces a fixed-size output no matter the input set size. To address this, the architecture is adapted to follow the encoder–pooling–decoder pattern, as in paragraph 3.3. Specifically, the local observation of an agent is concatenated with a globally pooled embedding derived from the entire agent set:

$$
\begin{aligned}
\text{SetTransformer}(O, \vec{o}) &= \phi_{sab}\big(\oplus_{\text{PMA}}(\psi_{isab}(O)) \parallel \vec{o}_i\big) \\
\text{where} \quad \psi_{isab}(O) &= \text{ISAB}(O) = \mathbf{Y} \\
\oplus_{\text{PMA}}(\mathbf{Y}) &= \text{PMA}(\mathbf{Y}) = \mathbf{Z} \\
\phi_{sab}(\mathbf{Z}) &= \text{rFF}(\text{SAB}(\mathbf{Z}))
\end{aligned}
$$

This computation is performed for each agent's observation $\vec{o}_i \in O$, aligning with the encoder–pooling–decoder paradigm (see paragraph 3.3), making it suitable for the MARL setting. A key benefit of this modular design is flexibility: the number of blocks in the encoder and decoder can be easily adjusted. For example, the encoder may comprise two or three stacked ISAB blocks. The impact of these design choices will be evaluated experimentally.

4.6.2 SAB and ISAB Transformer

Instead of relying on a pooling operation and a separate decoder, the SAB and ISAB blocks can be used as standalone solutions for the critic network, without requiring any additional pooling or decoding components. This is feasible because self-attention is

Embedding	Definition	Components (partial)
SetTransformer	$f(O, \vec{o}_i) = \phi_{sab}(\oplus_{\text{PMA}}(\psi_{isab}(O)) \parallel \vec{o}_i)$	$\psi_{isab}(O) = \text{ISAB}(O)$
SetTransformerOG	$f(O) = \phi_{sab}(\oplus_{\text{PMA}}(\psi_{isab}(O)))$	$\oplus_{\text{PMA}}(\mathbf{Y}) = \text{PMA}(\mathbf{Y})$
SABTransformer	$f(O) = \text{rFF}(\text{SAB}(\text{SAB}(O)))$	$\phi_{sab}(\mathbf{Z}) = \text{rFF}(\text{SAB}(\mathbf{Z}))$
ISABTransformer	$f(O) = \text{rFF}(\text{ISAB}(\text{ISAB}(O)))$	—

Tab. 4.3.: Definitions of the transformer embedding approaches. All variants are based on the SetTransformer [39]. In this table, the components of the *SetTransformer* are depicted with one ISAB and one SAB block, but the number of blocks making up the encoder and decoder can be varied. The same is true for the components making up the *SABTransformer* and the *ISABTransformer*. In addition to the three newly created transformer-based methods, the original SetTransformer is also depicted, as *SetTransformerOG*.

already permutation invariant and can process variable-sized input sets. Furthermore, self-attention captures interactions between elements, in this case, agent observations, allowing each embedding to incorporate information from all agents.

Based on this, two embedding architectures are proposed: the *SABTransformer* and the *ISABTransformer*, defined as follows:

$$\text{SABTransformer}(O) = \text{rFF}(\text{SAB}(\text{SAB}(O)))$$

$$\text{ISABTransformer}(O) = \text{rFF}(\text{ISAB}(\text{ISAB}(O)))$$

This architecture can be highly efficient, as it only needs to be applied once per set of agent observations, rather than once per agent. All pairwise interactions are modeled through the self-attention layer within the SAB or ISAB block. As with the *SetTransformer* embedding technique, the number of SAB or ISAB blocks is treated as a hyperparameter. This parameter will be varied in the experimental setup to balance computational complexity and performance. The row-wise feedforward network rFF is used solely to project the final embedding of each agent after the self-attention to the desired output dimension. For the critic network, this dimension is $d_{\text{rff}} = 1$ to produce a scalar value estimate for each agent. Notably, rFF is implemented as a straightforward linear layer without any additional activation or normalization functions.

4.7 Experiment Framework

This chapter describes the experimental setup used throughout this work. It provides a discussion of what environments are used and which settings were adopted. Moreover, it introduces the distinct MAPPO method employed and explains the evaluation metrics used to analyze these methods.

DeepSet	GNN	Transformer
DS	*GSAGE*	*SetTransformer*
DS_LOCAL	*GAT*	*SABTransformer*
DS_GLOBAL	*GATv2*	*ISABTransformer*

Tab. 4.4.: Grouping of embedding methods into three categories: DeepSet, GNN, Transformer. Because of page constraints, *DS_LOCAL*, *DS_GLOBAL* will occasionally be shortened to *DS_LO* and *DS_GL*. Similarly, *SetTransformer*, *SABTransformer* and *ISABTransformer* will be shortened to: *SET*, *SAB* and *ISAB*

Method groupings. This work explores various approaches to designing embedding architectures in MARL, specifically for MAPPO. To organize the evaluation of these approaches, the methods are grouped into three base categories with respect to their primary architectural characteristics: DeepSet-based embeddings, GNN-based embeddings, and transformer-based embeddings. Notably, certain methods can have features of multiple categories, for example, GAT uses graph structures and attention mechanisms. Nonetheless, it is classified under the GNN category rather than under the transformer category due to its graph-centric design. How the methods are grouped into the three categories and how they will be shortened is depicted in Table 4.4.

The categorization makes presenting the experiments and their findings straightforward and easy to follow. Several approaches have hyperparameters that are only present in their concepts and need explicit explanation. Each category will be introduced and fine-tuned individually, highlighting its unique characteristics and hyperparameter requirements. Afterward, the methods will be compared across categories to examine their relative performance, scalability, and generalization capabilities.

4.7.1 Environment Setting

From VMAS, three environments are used. Most importantly, the *balance* scenario is employed for hyperparameter tuning and the generalizability experiments. For general comparisons, the *navigation* and *multi-give-way* scenarios are also used alongside *balance*. For more information about the motivation behind choosing these environments, see section 4.2.

Scenarios. For *balance*, the default settings are modified. The primary change is increasing the maximum possible number of steps per episode to 200. Additional steps are necessary, since the balance scenario is complex and requires more timesteps for agents to complete the task of balancing the ball to its target. Leaving the maximum steps at the default value of 100 would mean that even an optimal policy cannot solve the task due to an insufficient number of steps. This was discovered during testing, as no combination of embedding strategies or models could achieve a satisfactorily high reward. The models would plateau at a low reward and fail to improve further. Additionally, the default number

of agents used in the hyperparameter tuning experiments is set to 5, to balance training efficiency and performance. In the *navigation* scenario, a maximum of 100 steps is sufficient to achieve good results. In the *multi-give-way* environment, 200 steps are used for the same reasons as in *balance*. The other environment settings remain unchanged, the complete parameter configurations can be found in Table A.2.

4.7.2 Key Performance Indicators

To evaluate how well the proposed methods perform, several metrics are used:

1. Training time (s)
2. Inference time (s)
3. Memory usage during training (MB)
4. Mean reward
5. Loss
6. Floating Point Operations per forward pass (millions)
7. Parameters (thousands)

Runtime and Resource Usage Metrics

The time-dependent metrics, *training time* and *inference time*, are not entirely reliable, as they depend on both implementation details and the current state of the local machine. For benchmarking, a MacBook Air with an M2 chip[8] is used. Particularly, inference time can vary significantly depending on background processes, which are difficult to control. Similarly, *memory usage* depends on how optimized the implementation is. However, memory usage has the benefit of being precisely measurable and relatively isolated from other tasks that might be running in parallel.

These sensitive runtime metrics are therefore primarily used to evaluate the overall complexity of the methods, rather than to conduct exact comparisons. Ideally, all embedding techniques would have similar complexity. Otherwise, it would be difficult to conclude that, for example, method A is better than method B, but A requires significantly longer training times. These Key Performance Indicators (KPIs) help identify when a method introduces substantially more complexity, and thus help contextualize its performance.

[8] `https://en.wikipedia.org/wiki/Apple_M2`

Profiling memory usage. In this thesis, the PyTorch profiler[9] is used to measure memory consumption. A limitation of profiling is that it affects overall performance. Running the profiler approximately doubles the execution time. Moreover, the profiler impacts different NN architectures differently. To address this, after completing the main experiment, an additional run is executed using the same configuration, with the profiler enabled for the first 10 iterations. The cumulative memory usage from these iterations is then used as the reported memory usage. Since memory usage is stable across runs, repeating the profiler measurements would introduce unnecessary computational overhead.

Limitations of the Python profiler. The Python profiler (see subsection 4.7.2) used to measure memory usage during training did not fully support the attention mechanisms employed in the transformer models. Additionally, it lacked compatibility with the "MPS" GPU backend. As a result, memory-related KPIs during training could not be reliably measured for the transformer approaches and are therefore not included in the evaluation.

Model Complexity Metrics

To better understand the complexity of the proposed techniques, machine-agnostic complexity metrics are required. For example, while training time can indicate model complexity, it also reflects how well the underlying software packages are optimized. To solve this, machine-independent metrics are used: the number of Floating Point Operations required for a single forward pass, and the number of learnable parameters. Both FLOPs and parameters are measured using the Python package `ptflops` [59]. FLOPs are reported in millions, while the number of parameters is reported in thousands.

Performance metrics. In addition to complexity metrics, performance metrics such as *Mean Reward* and *Loss* are recorded. The mean reward is computed as the average *episode reward* across all agents and all environments in the batch at a given training iteration. The *episode reward* is defined as the sum of all rewards received at each timestep of an episode. For simplicity, the term "mean reward" is often referred to as just "reward" throughout this thesis. The mean reward serves as a reliable metric for comparing different methods, as it is machine-agnostic and largely independent of the underlying implementation. The same holds true for the computed *Loss* values of the networks. Both metrics are therefore essential tools for fine-tuning and evaluating the various embedding approaches.

Confidence intervals. Each experiment is typically repeated 10 times. For all reported metrics, both the mean and the 90% confidence interval are provided. If an experiment is not repeated, it will be explicitly stated in the corresponding section. For FLOPs and the number of parameters, no confidence interval is stated, since these stay the same for all

[9] `https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html`

runs. As described in paragraph 4.7.2, the memory usage is also recorded only once, which is why no confidence interval can be specified.

4.8 Experiment Types

This section describes the types of experiments conducted in the experiments chapter of this thesis.

4.8.1 Hyperparameter Tuning

Training parameters. Training parameters such as the number of epochs and iterations used in the MAPPO algorithm are one category of parameters to tune. These parameters will be optimized specifically for the baseline method *CONCAT* to which the proposed architectures will be compared. This ensures that *CONCAT* benefits from well-tuned settings, which makes any performance improvement by a PI embedding method over *CONCAT* a meaningful result.

Method parameters. The first set of experiments focuses on tuning the hyperparameters of the embedding architectures. This tuning is organized according to the three method categories: DS, GNN, and Transformer. Each category shares similar hyperparameters within itself, whereas the hyperparameters can differ significantly between categories. All methods will be tuned using the balance scenario (see subsection 4.2.2), as it serves as the primary evaluation scenario throughout this thesis.

4.8.2 Cross-Method Evaluation

Since hyperparameter tuning is performed within method groupings, it is also important to compare the methods across groups as well as against the baseline architectures. In addition to evaluating performance in the balance scenario, methods will also be tested in the alternative scenarios introduced in section 4.2, i.e., the navigation and multi-give-way scenarios.

Another cross-method experiment investigates the impact of the pooling operation, specifically mean against max pooling (see subsection 3.3.1). Up to this point, mean pooling will be used for the DS methods and the GSAGE method. This choice is based on the findings of [9], who reported superior performance using mean pooling, and [10], who found that mean and max pooling achieved comparable results. It remains an open question how the choice of pooling method impacts the performance in the VMAS scenarios.

4.8.3 Empirical Scalability Analysis

Visual analysis. To assess the scalability of the different embedding methods, each relevant metric is plotted against the number of agents on both a linear and a logarithmic scale. The linear plots help illustrate the absolute magnitude of differences between methods and scenarios. In addition, metrics are visualized using *log-log plots*, in which both the x-axis (number of agents) and the y-axis (metric value) are logarithmically transformed. This type of visualization is useful for identifying whether a power-law relationship might exist between metric values and agent count.

If a metric M scales as $M(N) \propto N^a$, where N is the number of agents and a is the scaling exponent, then the log-log plot of the metric will appear as a straight line. In this case, the slope of the line corresponds to the exponent a, which indicates how the metric changes as the number of agents increases [85]. To give an example, say for a method, the training time increases quadratically with the number of agents. In the linear plot, this results in a quadratic graph, while it appears as a straight line of slope 2 in the log-log-plot. Meanwhile, a curved line in the log-log plot suggests that the metric does not follow a simple power-law scaling. For more examples of power-law relationships plotted in log-log form, see [86].

Linear regression and model fitting. To quantify the observed scaling behavior, a simple linear regression is performed on the log-transformed data points. The slope of the resulting regression line corresponds to the exponent a [87], and the coefficient of determination R^2 is used to assess how well the data fits a linear model in log-log space.

In this thesis, a threshold of $R^2 \geq 0.98$ is used to determine whether the relationship is sufficiently linear to interpret the slope as a power-law exponent. This threshold ensures that at least 98% of the variance in the log-transformed data is explained by the linear model, representing a very strong fit. The choice reflects a practical balance between statistical robustness and tolerance for minor empirical noise. It also aligns with visual inspection: in all examined cases, lines exceeding this threshold appeared visually linear.

The value of the regression in the log-transformed data, i.e. a, is then used to interpret the scaling pattern:

- $a \approx 1$: Approximately linear scaling of the metric with the number of agents. To account for minor deviations, slopes between 0.95 and 1.05 will be considered roughly linear, while slopes between 0.99 and 1.01 will be classified as linear.
- $a < 1$: Sublinear scaling, suggesting better than linear efficiency as the number of agents increases (e.g., square-root scaling for $k = \frac{1}{2}$).
- $a > 1$: Superlinear scaling, implying increasing resource demands (e.g., quadratic growth for $a = 2$).

Nonlinear trends and deviations. If the data deviates from a straight line in the log-log plot, i.e., $R^2 < 0.98$, this suggests that the scaling behavior is not well captured by a single exponent a. For example, a downward-curving line indicates that the rate of increase in the metric slows as the number of agents grows, which can be interpreted as improved efficiency at scale for that particular metric. In contrast, an upward-curving line suggests that the rate of increase accelerates with more agents, indicating reduced scalability. Such curvature may hint at exponential growth behavior, although the limited dataset makes it difficult to characterize this precisely.

Finally, for metrics where a slope and R^2 value could not be computed, for example, because all observed values are 0 or constant, it can imply a constant complexity. These cases result in flat lines in both linear and log-log plots. No exponent is computed in these instances, and the metric is considered unaffected by the number of agents.

Limitations. The results of this analysis should be interpreted with caution. Both the slope and R^2 values are derived from a limited number of empirical data points, without repeated trials or formal statistical testing. As a result, the estimated scaling behaviors are approximate and intended to illustrate general trends rather than to serve as rigorous theoretical claims. The relationships between the metrics and number of agents are also likely more complex than a simple power law relationship, therefore, the results of the empirical analysis are only approximations for the true scaling behavior. More robust conclusions would require broader sampling and statistical validation, which are considerations beyond the scope of this thesis.

4.8.4 Generalizability

A central question in this thesis is how well the proposed critic architectures generalize to agent counts not seen in the training phase. The generalizability results must be interpreted in the context of the specific scenarios used. Since most scenarios require only limited coordination among agents, testing generalization is challenging. However, the *balance* scenario (subsection 4.2.2) does facilitate different behaviors depending on the agent count and is therefore suited for evaluating how well a method generalizes to new agent counts in this setting.

Training data composition. One aspect under investigation is how the training process should be structured to achieve the best generalization. Specifically, whether it is more effective to train on a diverse set of agent counts or to focus on a narrow set of agents. These experiments aim to identify best practices for balancing variability in training with generalization performance.

Train low – test high. Another important question concerns the limits of generalization. That means, how well do policies trained on a small agent count perform when acting in environments with significantly more agents than those seen during training? This experiment tests the robustness and scalability of the learned representations.

Training mostly low. This aspect explores how well models perform that were trained for the majority of the training phase on a few agents in the environment, and only in some iterations on higher agent counts. This has the potential of yielding high rewards while keeping training times manageable.

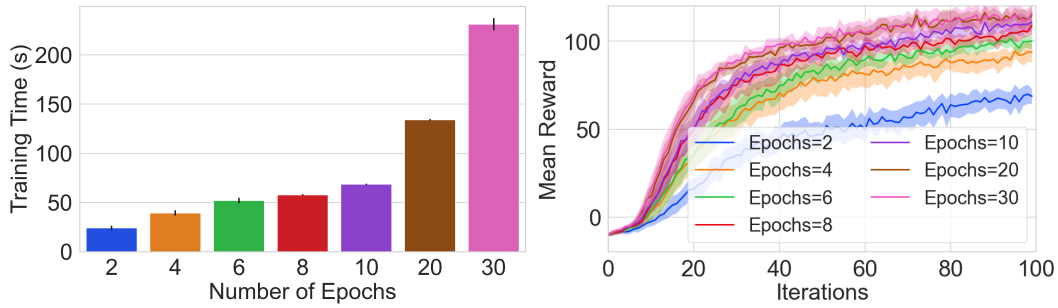
4.8.5 MAPPO with centralized execution

How the models fare when trained with CPPO instead of MAPPO will be analyzed in this experiment class. Instead of training and testing using the CTDE framework, the agent has access to the global observations of all agents in the environment during training and execution, meaning it will implement CTCE.

5 Experiments

5.1 Baseline – *CONCAT*

This chapter presents the hyperparameter tuning for the *CONCAT* baseline used to compare all other methods (section 4.3). Moreover, the MAPPO training parameters will be analyzed to optimize the rewards.



(a) CONCAT training times with varying epochs **(b)** CONCAT rewards with varying epochs

Fig. 5.1.: Tuning the number of epochs in MAPPO for the *CONCAT* strategy.

Training parameters for *CONCAT*. To balance training time and performance, the number of epochs per iteration in MAPPO is examined. Each epoch optimizes the neural networks used in MAPPO in each iteration. In the original training loop [83], 30 epochs were used, which led to high training times. Figure 5.1 shows how the number of epochs affects both training duration and achieved rewards. More epochs per iteration allow for faster learning, resulting in higher rewards that plateau later. However, for complex embedding methods, using more than 10 epochs becomes impractical, as optimization becomes the bottleneck rather than the environment rollouts.

Since the goal is not to achieve the highest possible reward, but to compare embedding methods fairly against *CONCAT*, epochs = 8 is set. This configuration achieves results comparable to those with a higher number of epochs. After 80 iterations, a mean reward of 100.93 ± 4.09 is achieved. Moreover, 8 epochs aligns better with the range of epochs used in the original PPO paper, which experimented with values between 3 and 15 [64].

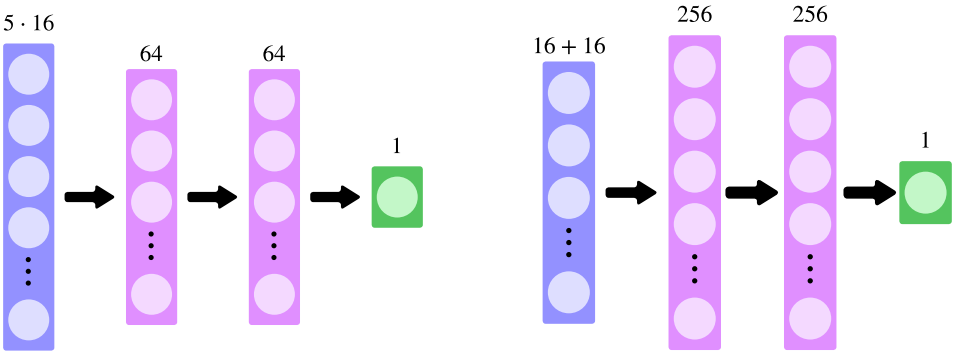
By setting epochs = 8 and iterations = 80, *CONCAT* achieves competitive results without excessive computational demands. Additionally, selecting a configuration in which *CONCAT* reaches a reward around 100 makes comparisons intuitive: if another method

exceeds or falls short of the 100 benchmark, it is immediately clear whether it outperforms or underperforms relative to *CONCAT*.

Epochs	Training Time	Memory Usage	Inference Time	Mean Rewards	Loss
2	24.03 ± 2.29	330.00	$5.86 \cdot 10^{-4} \pm 8.84 \cdot 10^{-5}$	68.59 ± 4.21	0.41 ± 0.37
4	39.33 ± 2.58	661.00	$5.91 \cdot 10^{-4} \pm 8.81 \cdot 10^{-5}$	93.65 ± 5.61	−0.19 ± 0.27
6	51.95 ± 2.72	991.00	$5.18 \cdot 10^{-4} \pm 6.68 \cdot 10^{-6}$	100.03 ± 4.91	−0.39 ± 0.19
8	57.73 ± 0.63	1320.00	$4.55 \cdot 10^{-4} \pm 5.55 \cdot 10^{-6}$	108.91 ± 4.82	−0.93 ± 0.31
10	68.54 ± 0.64	1650.00	$4.65 \cdot 10^{-4} \pm 2.09 \cdot 10^{-5}$	110.64 ± 5.91	−1.17 ± 0.23
20	134.00 ± 0.67	3300.00	$4.77 \cdot 10^{-4} \pm 1.04 \cdot 10^{-5}$	114.53 ± 5.60	−1.58 ± 0.30
30	231.14 ± 6.28	4960.00	$5.55 \cdot 10^{-4} \pm 2.33 \cdot 10^{-5}$	116.05 ± 5.57	−1.83 ± 0.30

Tab. 5.1.: Tuning the number of epochs with *CONCAT* for 100 iterations. Note that all have the same decoder network $\phi()$ with $37.60 \cdot 10^6$ FLOPs and $18.80 \cdot 10^3$ parameters.

Model parameters for decoder ϕ. As defined in Equation 4.1, the basic *CONCAT* method uses a decoder MLP $\phi()$, whose architecture can be adapted as needed. The optimal layer size and depth, depend on the specific environment. To ensure that *CONCAT* provides a strong baseline, both layer size and depth were tuned for maximum reward. First, the depth was fixed at 2 layers while varying the number of hidden units across values $[16, 32, 64, 128, 256]$. A hidden layer size of 64 had the best results. Afterwards, the depth was varied between 1, 2, and 3 layers, with a depth of 2 achieving the highest reward. The complete evaluation metrics are reported in Table A.3 and Table A.4. The final tuned fully connected MLP has the following architecture: an input layer sized according to the number of agents multiplied by the size of each agent's observation, followed by two hidden layers of 64 units each, each followed by a Tanh activation function, which worked well in this thesis and the original MAPPO source [83]. A final linear output layer completes the network, without any additional activation function. A visualization of this network can be seen in Figure 5.2a.



(a) Decoder network ϕ for *CONCAT* **(b)** Decoder network ϕ for *DS_GLOBAL*

Fig. 5.2.: Depiction of the decoder networks for *CONCAT* and *DS_GLOBAL* with 5 agents in the environment and observations of size 16. The number above each layer is the size of the layer, i.e. the number of neurons. For *DS*, the input of the decoder is 16, instead of 32, since the agent's observation is not concatenated with the output of the encoder. For *DS_LOCAL*, the cell size is 128 instead of 256, due to better performance. For simplicity, activations and normalizations are omitted from the visualization.

5.2 DeepSet-based Approaches

Default decoder network ϕ. As the DS-based approaches include both an encoder ψ and a decoder ϕ, these networks can be designed independently to optimize performance. However, to streamline the tuning process, the focus is placed on the encoder network, as it represents the main addition compared to the *CONCAT* method. The decoder architecture remains largely consistent with the one used in *CONCAT*, since it serves the same purpose. Therefore, the decoder is also designed with a depth of 2, i.e., two hidden layers followed by an output layer (Figure 5.2b). However, setting the hidden layer size to 64, as in *CONCAT*, resulted in worse performance for the DS approaches. Using layer sizes of 128 or 256 with the same depth of 2 produced better rewards. This is likely due to the fact that, in DS approaches, the input to the decoder is considerably smaller as a result of the pooling operation. For instance, instead of an input size of $80 = 5 \cdot 16$, with 5 agents and observation vectors of size 16 each, the input is only the output dimension of the encoder. Consequently, a layer size of 64 is not expressive enough. Therefore, the DS experiments were conducted using decoder hidden layer sizes of 128 and 256.

5.2.1 Encoder Width and Depth

Width. The hidden layer size or width of the encoder network is a crucial hyperparameter. Depending on the complexity of the agent states and the task, different widths may yield better performance for the DS-based approaches. Four different widths were explored in the experiments: 16, 32, 48, and 64. A width of 16 was chosen as the lower bound, as it matches the size of each agent's observation vector in the balance scenario. To evaluate whether larger widths could extract more information and lead to improved rewards, the larger sizes were tested.

Depth. The encoder network's depth is another key design choice. As with the width variations described in paragraph 5.2.1, three different depths were evaluated for each DS approach. The first variant has no encoder at all (depth = None), in which case the agents' observations are pooled directly without further processing and then passed through the decoder. The second variant has a depth of 1, consisting of a single linear layer followed by a *tanh* activation function and a final output layer. For depth = 2, the encoder is composed of two linear layers, each followed by a *tanh* activation, with a final output layer.

For example, with a width of 64 and a depth of 2, the encoder architecture includes an input layer of size 16 (matching the observation dimension), followed by a *tanh* activation, a linear layer of size 64, another *tanh* activation, and a final linear layer of size 64. The resulting embedding vectors, each of dimension 64, are then aggregated using mean pooling. Architectures with widths of 16, 32, and 48 follow the same structural pattern.

Strategy	Width	Depth = 2	Depth = 1	Depth=None
DS	16	94.00 ± 3.32	88.25 ± 3.81	**98.95 ± 3.49**
	32	92.09 ± 3.12	91.68 ± 4.54	
	48	90.29 ± 4.73	*95.04 ± 4.36*	
	64	89.55 ± 10.24	90.72 ± 3.47	
*DS_LOCAL**	16	91.30 ± 5.98	*95.47 ± 5.04*	**96.51 ± 3.57**
	32	90.44 ± 3.94	92.74 ± 3.92	
	48	87.10 ± 9.58	90.89 ± 3.61	
	64	90.85 ± 4.46	90.62 ± 4.56	
DS_GLOBAL	16	**97.78 ± 5.33**	94.67 ± 2.78	94.23 ± 3.54
	32	*95.72 ± 2.77*	93.49 ± 3.01	
	48	90.45 ± 5.34	91.15 ± 3.17	
	64	91.72 ± 3.85	89.29 ± 5.72	

Tab. 5.2.: Mean rewards after 80 training iterations. The best results are indicated in bold, and the second best in italics. All results are close, especially considering the 90% confidence intervals. There is also no clear trend towards larger widths or depth. **DS_LOCAL* was run with 128 cells in the hidden layers, as this returned the best result for this approach.

Results – Encoder Width and Depth

In Table 5.2, the resulting *mean rewards* achieved for varying depths and widths of the encoder after training for 80 iterations are depicted. To provide a broader view of the training process, Figure 5.4 visualizes the mean rewards over all iterations. These graphs already suggest that encoder depth and width had only a minor influence on overall performance. All combinations of depth and width achieved relatively high rewards and similar learning speeds.

A closer look at the rewards after 80 iterations reveals the effects of the encoder's width and depth. Surprisingly, for both *DS* and *DS_LOCAL*, setting the depth to *None*, i.e., using no MLP for encoding, produced the best results. Additionally, a layer width of 16 consistently yielded strong performance. For *DS_GLOBAL*, the best result was achieved with the combination depth = 2 and width = 16, reaching a final reward of 97.78.

It is notable that *DS* outperforms the other embedding methods despite not explicitly passing the agents' local states to the decoder network. Including the local state appears to have no meaningful impact on either learning speed or final performance. However, using a wide encoder with 64 units performs particularly poorly for *DS*. Even more telling than the final rewards are the confidence intervals: for example, the setup with depth = 2 and width = 64 has a confidence interval of 10.24, nearly twice as high as any other configuration across all experiments. A similar pattern is seen for *DS_LOCAL* with depth = 2 and width = 48.

The strong performance of configurations with no encoder may result from the simplicity of the agent observations in the environment, as adding learnable parameters does not create additional insight for the models. In this task, a simple mean of the observations appears sufficient for the DS approaches. This outcome may differ for more complex environments.

These findings align with results from the SwarmRL paper [23], which concluded that shallow networks performed best in their MARL scenarios. The same holds true for the balance task used in this experiment: either no encoder or encoders with a small width performed the best.

Results – Resource Needs

The width and depth of the encoder also influence the computational and memory requirements of the methods. The absence of an encoder network results in the fastest training times and the lowest memory usage. This is illustrated in Figure 5.3, where no-encoder is labeled as depth = 0. Due to the relatively simple neural networks used in the DS approaches, training time shows only a slight upward trend with increasing embedding widths. However, for *DS_LOCAL*, training time increases faster with growing embedding widths. This is because the GSE must be computed independently for each agent. The relationship between memory usage and embedding width is even more pronounced, with the memory consumption increasing linearly with the embedding width. Although *DS_LOCAL* starts with lower memory usage, due to its smaller decoder network, it shows the fastest memory growth. This is because, during the forward pass, *DS_LOCAL* must store and process intermediate encoder outputs and construct separate global states for each agent. As embedding dimensions grow, these intermediate tensors become larger, leading to increased memory usage.

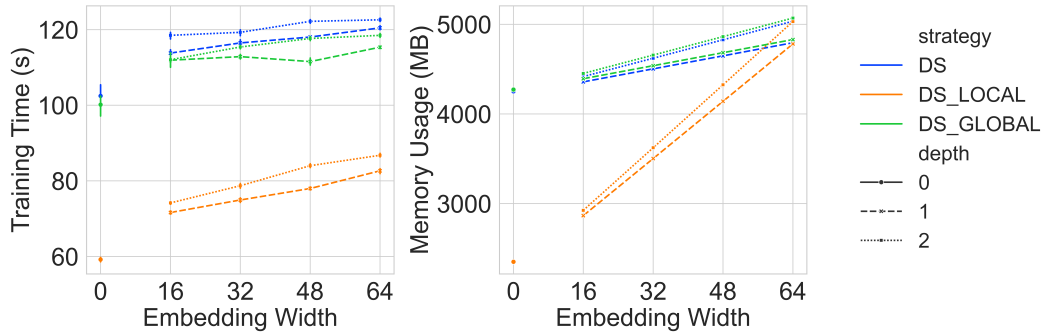
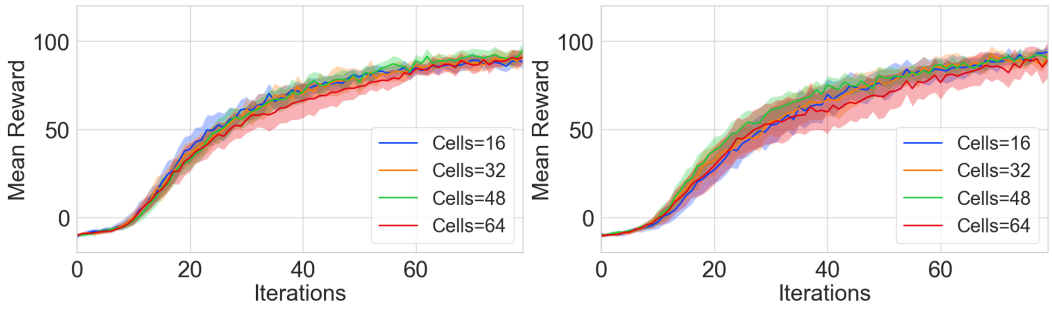
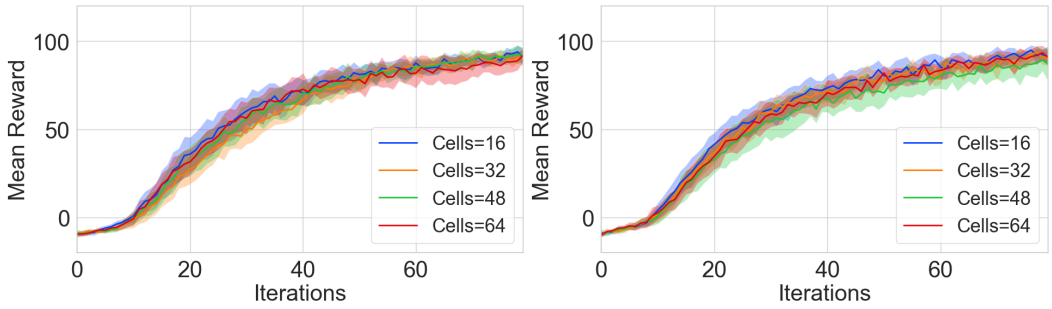


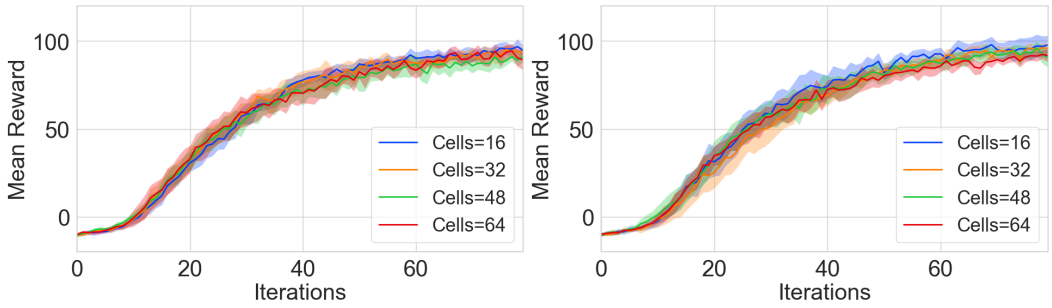
Fig. 5.3.: Training duration and memory usage of DS approaches. *DS_LOCAL* achieved the best results with a decoder network of width 128. Thus, the training times and memory usage are naturally smaller compared to *DS* and *DS_GLOBAL*, which use a decoder of width 256. Even so, the memory usage increases faster for *DS_LOCAL* due to the overhead of generating the global state for this method compared to the other DS techniques.



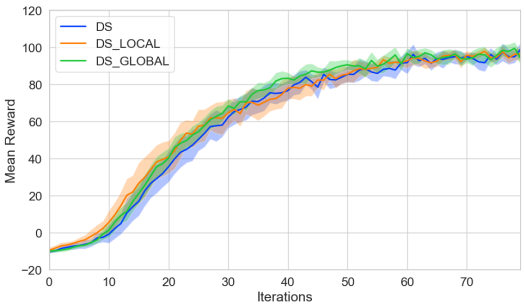
(a) *DS* depth 1. (b) *DS* depth 2.



(c) *DS_LOCAL* depth 1*. (d) *DS_LOCAL* depth 2*.



(e) *DS_GLOBAL* depth 1. (f) *DS_GLOBAL* depth 2.



(g) No depth variant*.

Fig. 5.4.: Plots of the mean rewards achieved by varying the depth of the encoding network. Subfigures marked with * correspond to *DS_LOCAL* models constructed with 128 core cells in the decoder, since the methods performed better with this configuration.

5.3 GNN-based Approaches

This chapter explores architectures for the critic network using GNN techniques. The methods include *GSAGE*, *GAT*, and *GATv2*.

5.3.1 Setup

Encoder network $\psi()$. The encoder serves the same purpose as in the DS methods but differs architecturally. When *Encode* = *True*, an MLP with a single hidden layer maps the input to dimension d', followed by a ReLU and output projection. This mirrors the setup in the DS-based approaches with depth = 1 (see subsection 5.2.1). When *Encode* = *False*, no nonlinearity is used. However, a linear layer still maps the input to d', similar to depth = None in previous experiments.

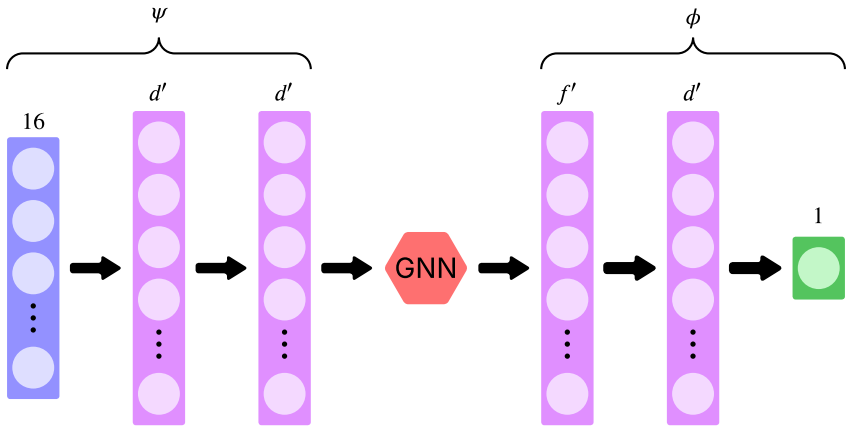


Fig. 5.5.: Architecture of the GNN models. Hidden layer width d' impacts the encoder $\psi()$, decoder $\phi()$, and internal GNN mechanism. For GSAGE, $f' = d'$, while for GAT and GATv2, $f' = m \cdot d'$, where m is the number of attention heads. The diagram reflects the case *Encode* = *True*. Activation and normalization are omitted for clarity. In practice, a ReLU activation follows the projection in *Encode* = *True*. For *Encode* = *False*, a single linear projection is applied from size 16 to d' without activation. For *GSAGE*, normalization is applied after the GNN step. See section 4.5 for further details.

Decoder network $\phi()$. The decoder, as in DS methods, maps internal embeddings to the scalar output, in the case of the critic. The decoder's primary role is dimensional adjustment, as the expressiveness of the model is handled by the GNN mechanism. Therefore, the decoder consists of one hidden layer from f' to d', followed by ReLU, and a final scalar output. Here, $f' = d'$ for *GSAGE*, and $f' = m \cdot d'$ for attention-based variants.

5.3.2 Hidden Layer Width Encoder

This section investigates the impact of hidden layer width d' on model performance. Complex MARL tasks may benefit from projecting the input into a higher-dimensional space ($d' > d$), while simpler tasks may perform better with $d' \approx d$. Widths of 16, 32, 48, and 64 are tested, all multiples of the input observation size of 16. Unlike the DS experiments in subsection 5.2.1, d' here affects the encoder, decoder, and internal GNN operations (see Figure 5.5).

Results. As shown in Table 5.3, *GAT* and *GATv2* both benefit from increased width. The performance difference across widths is also more pronounced than in DS-based models. For instance, *GATv2* achieves a mean reward of over 95 at widths 48 and 64, compared to just 81.08 at width 16. This suggests a stronger dependence on embedding size, particularly for attention-based GNNs.

GAT underperforms overall, with a maximum mean reward of only 77.78. As seen in Figure 5.6, its performance often stagnates or degrades after around 60 iterations, indicating potential convergence to a suboptimal policy. Loss plots (Figure A.4) show continued decrease in training loss, but this does not correlate with better performance, suggesting overfitting or unstable learning. Additionally, *GAT* exhibits higher variance across runs, further indicating its increased instability.

Strategy	Width	Encode = True	Encode = False
GSAGE	16	93.52 ± 2.69	91.88 ± 4.45
	32	**97.98 ± 2.74**	93.87 ± 3.37
	48	94.74 ± 3.73	92.16 ± 4.81
	64	93.96 ± 2.64	*96.78 ± 2.88*
GAT	16	72.59 ± 8.09	60.97 ± 7.00
	32	70.28 ± 8.26	63.75 ± 8.98
	48	*75.87* ± 6.60	68.79 ± 8.81
	64	**77.78 ± 8.99**	65.94 ± 10.07
GATv2	16	81.08 ± 9.40	85.90 ± 9.05
	32	94.78 ± 7.37	91.20 ± 2.49
	48	**96.08 ± 3.22**	*92.73 ± 4.41*
	64	*95.50* ± 4.38	92.25 ± 3.71

Tab. 5.3.: Mean rewards after 80 training iterations. Best results depicted in bold, second-best in italics. A width of just 16 performed worse than higher widths in most cases, otherwise it is difficult to draw conclusions about the hidden layer width. For all methods, *Encode = True* yielded the best performing model configurations.

5.3.3 Necessity of Encoder ψ

As in subsection 5.2.1, it is tested whether a separate encoder network is necessary. For DS models, omitting the encoder often yielded better results. In GNN-based models, *Encode* = *False* implies only a linear projection, whereas *Encode* = *True* includes a ReLU nonlinearity, as illustrated in Figure 5.5.

Results. All GNN models performed better with encoding. The improvement was most notable for *GAT*, while *GSAGE* and *GATv2* received smaller gains from the encoding. This contrasts with DS methods, where encoders often degraded performance. Thus, shallow encoding appears to be beneficial for the GNNs techniques.

5.3.4 Attention Heads

The number of attention heads is a key hyperparameter in attention-based GNNs. Each head can learn different relations between inputs, possibly leading to better performance at the cost of higher computational overhead. The number of attention heads is varied for the best-performing configurations: width 48 for *GATv2*, and widths 64 (*Encode* = *True*) and 48 (*Encode* = *False*) for *GAT*.

Results. For *GATv2*, the number of heads had little impact on final performance. One-head models trained slightly slower but achieved comparable rewards after 80 iterations. In contrast, *GAT* was very sensitive to the number of heads. With three heads and *Encode* = *True*, it reached a mean reward of 97.27, comparable to *GATv2*, but it performed significantly worse with fewer heads. Performance also dropped substantially without an encoder.

Figure 5.8 shows that training time and memory consumption roughly double when increasing from one to three heads, for both models, while *GAT* is slightly more efficient overall. Since one, two, and three attention heads performed similarly well for *GATv2*, but the resource usage is substantially better with just one head, this will be the default for *GATv2*. *GAT* will use three attention heads, as this increased performance greatly.

5.4 Transformer-based Approaches

In this section, the transformer-based methods are tuned. These are *SetTransformer*, *SABTransformer*, and *ISABTransformer*.

Layer normalization. In the original SetTransformer, layer normalization is performed after each attention block. However, this resulted in worse performance for the transformer methods, which is why it will be omitted. The experiments with normalization can be viewed in Figure A.5.

Original SetTransformer. The *SetTransformerOG* was tuned alongside the other invariant transformer methods to ensure that the modifications to the architecture did not negatively impact the performance of the models. The experiments show that the invariant SetTransformer approaches performed just as well or better than the original SetTransformer formulation.

5.4.1 Model Dimensions

The model dimension in the *SetTransformer* is similar to the concept of the hidden layer width in GNN techniques. The parameter model_dim sets the dimensionality of the embeddings inside the self-attention blocks. For example, with model_dim = 32, each token, i.e., agent observation, is transformed into a 32-dimensional vector inside the attention blocks. The model dimensions are varied between $16, 32, 48$, and 64. The model dimension influences the query, key, and value matrices in the attention mechanism, since model_dim = $d_q = d_k = d_v$ (see Equation 3.14 and Equation 3.7).

Results. Varying the model dimensions did not significantly affect the models' rewards after the full 80 iterations (see Figure 5.9). However, it did impact sample efficiency greatly. All SetTransformer-based methods learned slower with model_dim = 16. This outcome is similar to the hidden layer width experiment for GNN, where a layer width of 16 also performed worse than larger widths.

For the *SABTransformer* and *ISABTransformer*, a dimension of 64 performed best, and particularly for the *ISABTransformer*, the learning speed was the fastest at 64. At 80 iterations, the rewards for the *SABTransformer* drop slightly, which is why in Table 5.4 the model with 64 dimensions is marginally outperformed by the one with 48 dimensions. The plots showing the training process reveal that the model with 64 dimensions has more potential than the others (see Figure 5.9). Therefore, in the following experiments for both methods, the dimension will be set to 64. In the *SetTransformer* and *SetTransformerOG*, 64 and 48 performed very similarly, but because of faster training times, 48 will be chosen as the default setting. For the concrete results, see Table 5.4.

Strategy	Model Dimensions	Training Time	Mean Rewards
ISABTransformer	16	80.14 ± 2.97	92.29 ± 5.10
	32	98.75 ± 1.46	*99.06 ± 4.45*
	48	384.96 ± 2.38	96.71 ± 7.78
	64	395.85 ± 1.00	**102.84 ± 4.79**
SABTransformer	16	59.51 ± 0.10	89.75 ± 3.44
	32	197.28 ± 0.29	99.22 ± 5.46
	48	209.19 ± 0.55	**100.69 ± 4.27**
	64	219.32 ± 1.13	*100.67 ± 4.88*
SetTransformer	16	145.48 ± 0.08	94.31 ± 3.10
	32	172.67 ± 0.23	*98.77 ± 2.31*
	48	473.24 ± 0.59	**100.19 ± 3.56**
	64	500.61 ± 1.05	98.28 ± 3.70
SetTransformerOG	16	125.91 ± 3.55	92.29 ± 5.60
	32	314.27 ± 1.11	91.28 ± 8.64
	48	555.79 ± 0.61	*100.41 ± 3.53*
	64	586.37 ± 1.77	**100.68 ± 3.82**

Tab. 5.4.: Training time and mean rewards across model dimensions and transformer strategies. Best and second-best rewards are bolded and italicized, respectively. For the remaining performance metrics, refer to Table A.9.

5.4.2 Attention Heads

This experiment is analogous to the GNN experiment in subsection 5.3.4, where the number of attention heads used in the multi-headed attention mechanisms is tuned. In contrast to the GNN experiment, the number of attention heads is varied between 1, 2, and 4. This is because the embedding dimensions must be evenly divisible by the number of heads. Since the default model dimension for the *SABTransformer* and *ISABTransformer* is 64, which is not divisible by 3, three heads cannot be used. This was not an issue for the GAT methods, because in GAT, the attention mechanism works differently: instead of replicating the input dimensions across each head, they are split across heads in the transformer methods.

Results. Similar to *GATv2*, the number of attention heads did not meaningfully impact the rewards of the transformer methods. For the following experiments, the attention heads for all methods will be set to one, since a single head performed just as well or better than multiple heads (see Figure 5.10).

5.4.3 Inducing Points

The inducing points are an important aspect of the *SetTransformer* models. They handle the quadratic complexity of multi-head attention while remaining trainable parameters, allowing the model to maintain its expressive (see paragraph 3.5.4). In the paper defining

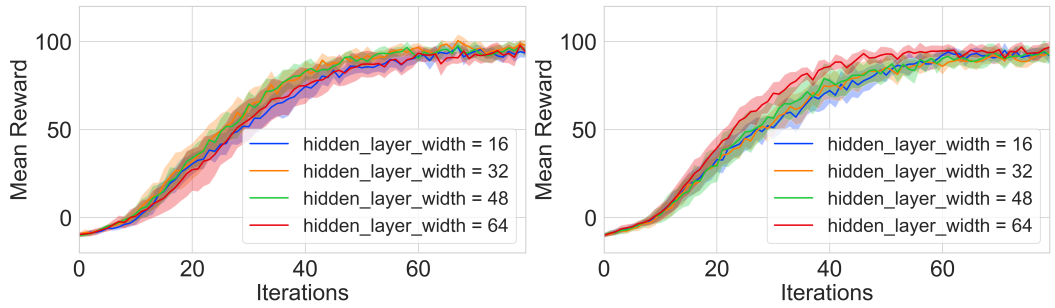
the *SetTransformer* [39], the only suggestion for setting the number of inducing points m is that $m < n$ to reduce complexity, where n is the number of agents in this context. With 5 agents in the environment, this gives $m \in \{1, 2, 3, 4\}$.

Results. Changing the number of inducing points did not have a pronounced effect on rewards (see Figure 5.11). Even with only one inducing point, the model's performance was promising. Training duration also did not change much for the different configurations. This is likely because inducing points do not substantially reduce the complexity of the overall parameters when using only 5 agents. Moreover, the differences between the tested numbers of inducing points are small. A slight trend indicates that more inducing points increase training time, though only marginally (see Table A.11). Because of the beneficial impact of using only one inducing point on training time and FLOPs, one inducing point will be used as the default parameter.

5.4.4 Attention Blocks

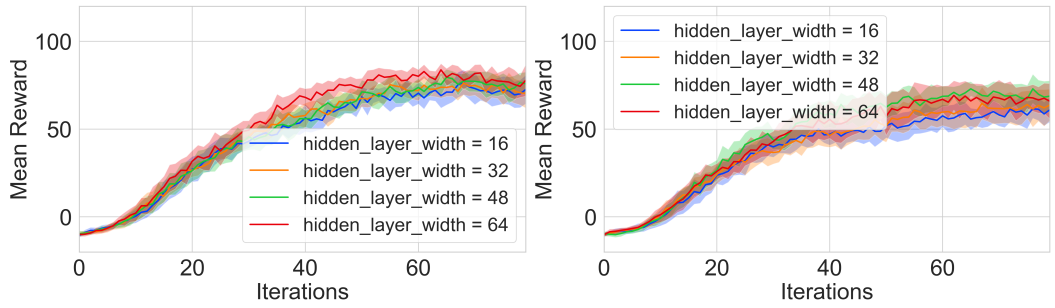
The number of attention blocks in the SetTransformer methods regulates model depth, similar to the encoder depth in the DS methods (subsection 5.2.1). For the *SABTransformer* and *ISABTransformer*, the depth impacts the entire architecture, as encoder, pooling, and decoder are fused within the attention blocks. For instance, a depth of 2 implies using 2 SAB blocks as the *SABTransformer*. For the *SetTransformer*, the number of blocks influences the encoder and decoder parts. Here, a block count of 2 means two SAB blocks in the encoder and two ISAB blocks in the decoder. The PMA block remains unchanged regardless of the number of SAB and ISAB components.

Results. Using only one SAB or ISAB block seems insufficient for the balance scenario. Both *SABTransformer* and *ISABTransformer* perform worse with only one block (see Figure 5.12). Even though the *ISABTransformer* with one block reaches comparable rewards after 80 iterations, its learning speed is slower than with multiple blocks. By contrast, the *SetTransformer* and *SetTransformerOG* perform well with count = 1, meaning they use one SAB and one ISAB block. This configuration will be chosen as the default for the methods. The *SABTransformer* and *ISABTransformer* will use count = 2 as their default parameter.c



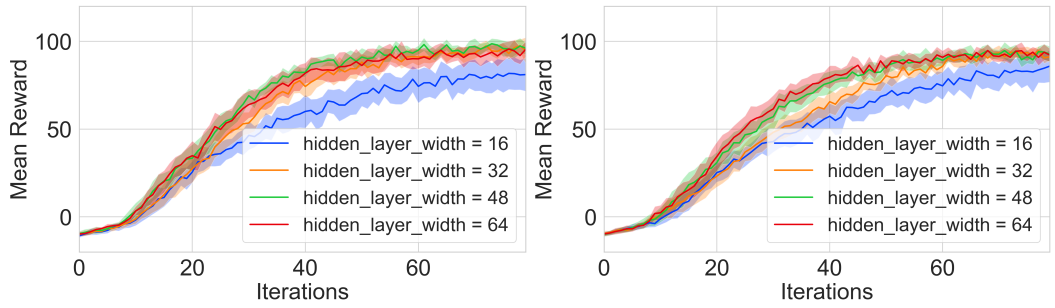
(a) *GSAGE* with *Encode* = *True*

(b) *GSAGE* with *Encode* = *False*



(c) *GAT* with *Encode* = *True*

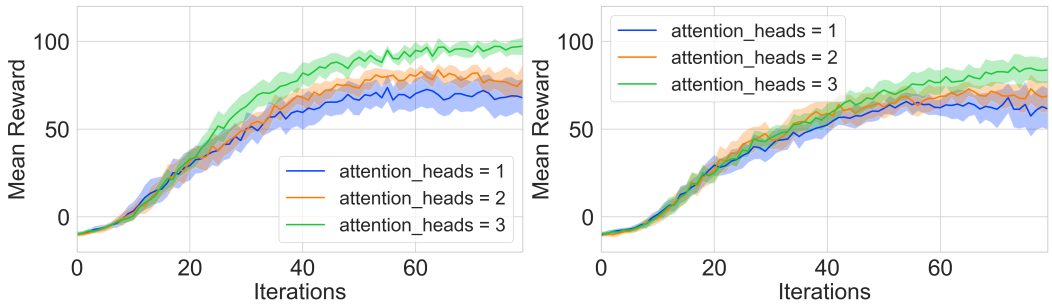
(d) *GAT* with *Encode* = *False*



(e) *GATv2* with *Encode* = *True*

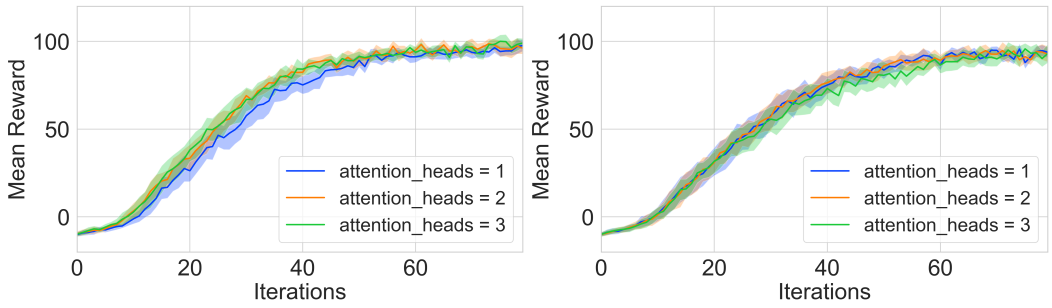
(f) *GATv2* with *Encode* = *False*

Fig. 5.6.: Training rewards for GNN variants with and without encoder networks for different hidden layer widths.



(a) *GAT* with *Encode* = *True*.

(b) *GAT* with *Encode* = *False*.



(c) *GATv2* with *Encode* = *True*.

(d) *GATv2* with *Encode* = *False*.

Fig. 5.7.: Effect of varying attention head count on training rewards.

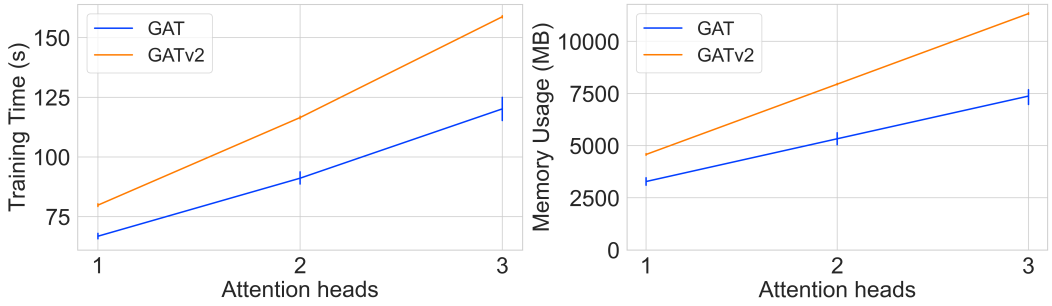
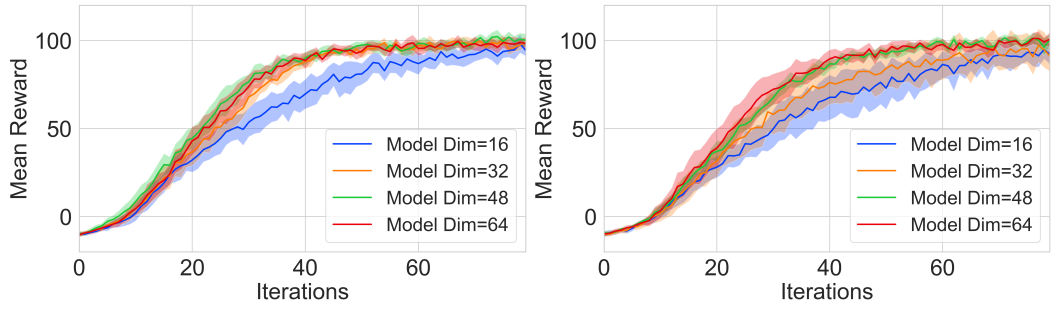
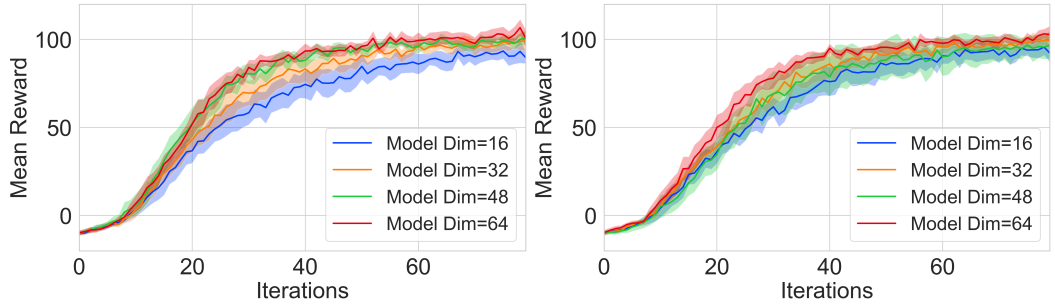


Fig. 5.8.: Training time and memory usage in relation to the number of attention heads for *GAT* and *GATv2*.



(a) Rewards of *SetTransformer*.

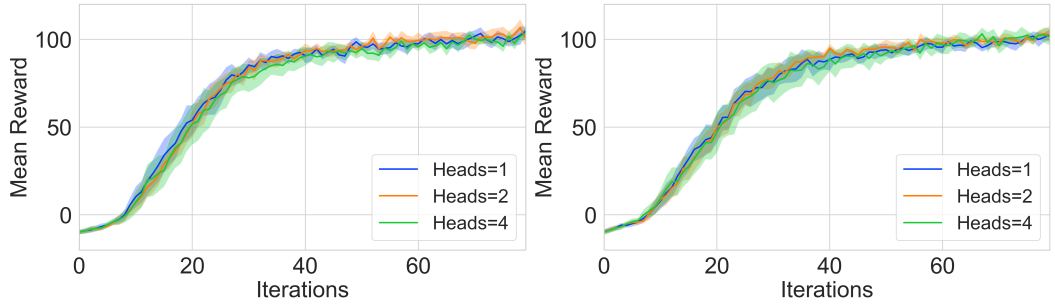
(b) Rewards of *SetTransformerOG*.



(c) Rewards of *SABTransformer*.

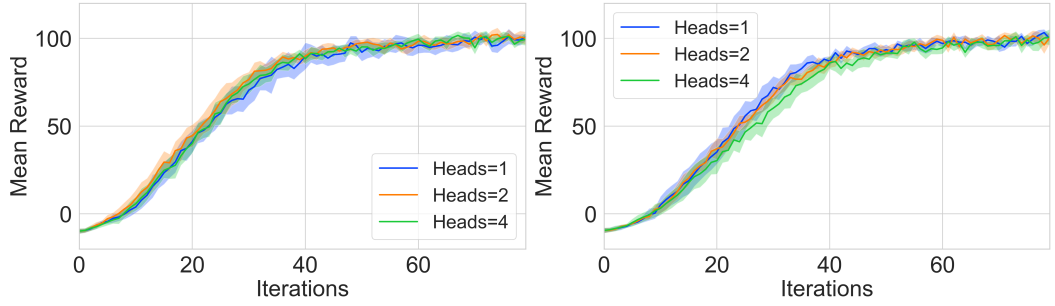
(d) Rewards of *ISABTransformer*.

Fig. 5.9.: Rewards of transformer methods with varying model dimensions.



(a) Rewards of *SABTransformer*

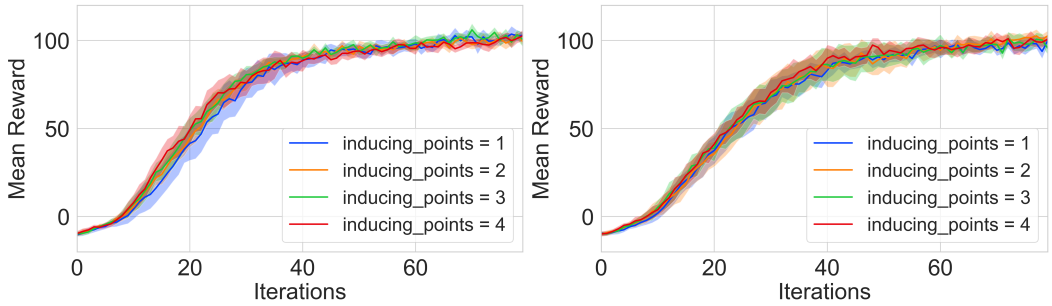
(b) Rewards of *ISABTransformer*.



(c) Rewards of *SetTransformer*.

(d) Rewards of *SetTransformerOG*.

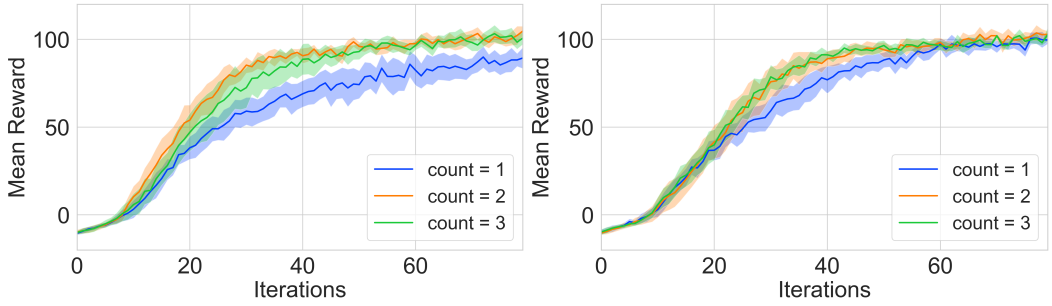
Fig. 5.10.: Varying number of attention heads for transformer techniques.



(a) Rewards of *ISABTransformer*

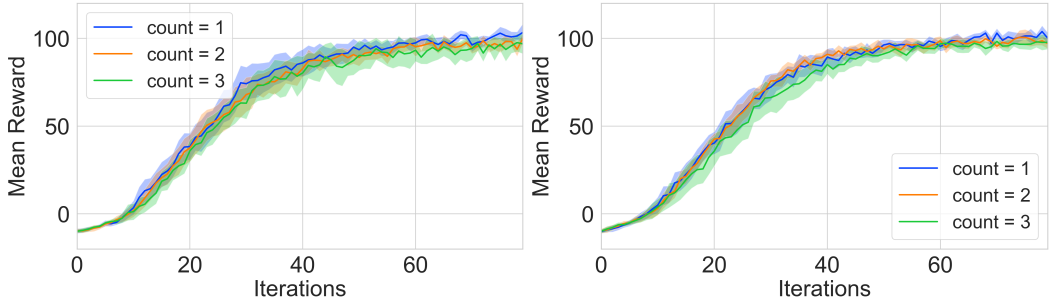
(b) Rewards of *SetTransformer*.

Fig. 5.11.: Varying number of inducing points.



(a) Rewards of *SABTransformer*

(b) Rewards of *ISABTransformer*.



(c) Rewards of *SetTransformer*.

(d) Rewards of *SetTransformerOG*.

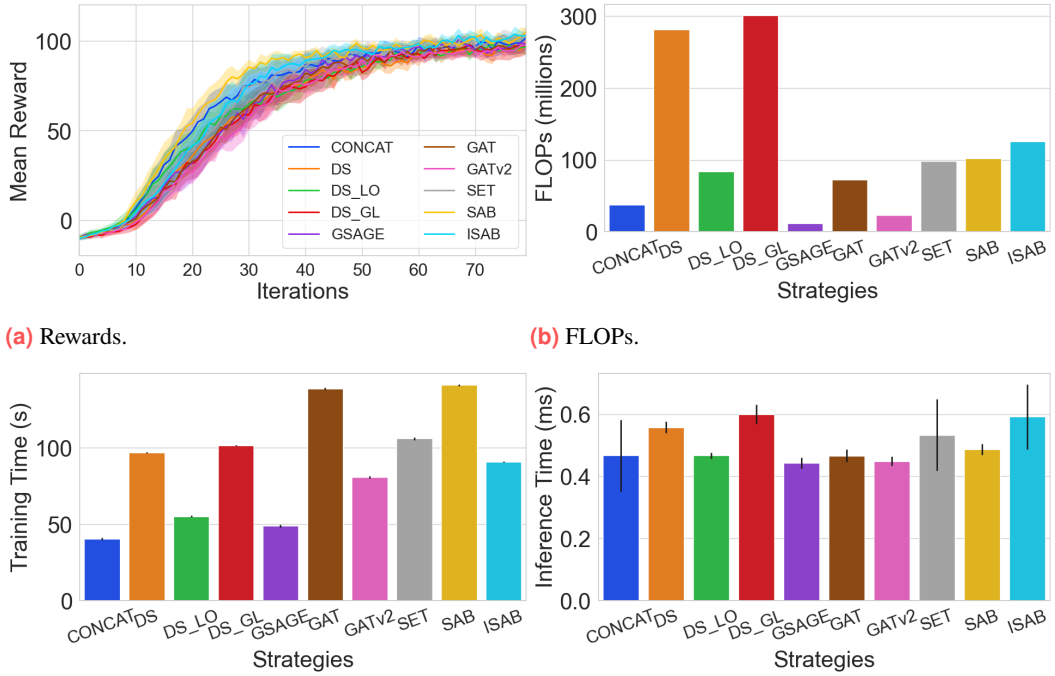
Fig. 5.12.: Varying number of transformer blocks for transformer techniques.

5.5 Cross Method Comparison

With the invariant approaches' hyperparameters tuned, they can now be compared with each other to determine their relative performance. They are also compared to the baseline *CONCAT* approach, to evaluate whether they manage to outperform it. The hyperparameters that resulted in the best performance are listed in Table 5.5. For the experiments with the other environments, these parameters will also be used.

Strategy	Decoder Width	Encoder Depth	Encoder Width	Encode	Heads	Inducing Points	Model Dims	Blocks
CONCAT	64	-	-	-	-	-	-	-
DS	256	None	-	-	-	-	-	-
DS_LOCAL	128	None	-	-	-	-	-	-
DS_GLOBAL	256	2	16	-	-	-	-	-
GSAGE	-	-	32	True	-	-	-	-
GAT	-	-	64	True	3	-	-	-
GATv2	-	-	48	True	1	-	-	-
SetTransformer	-	-	-	-	1	1	48	1
SABTransformer	-	-	-	-	1	-	64	2
ISABTransformer	-	-	-	-	1	1	64	2

Tab. 5.5.: Hyperparameters that led to the best performance for each method in the balance scenario.



(a) Rewards.

(b) FLOPs.

(c) Training time.

(d) Inference time.

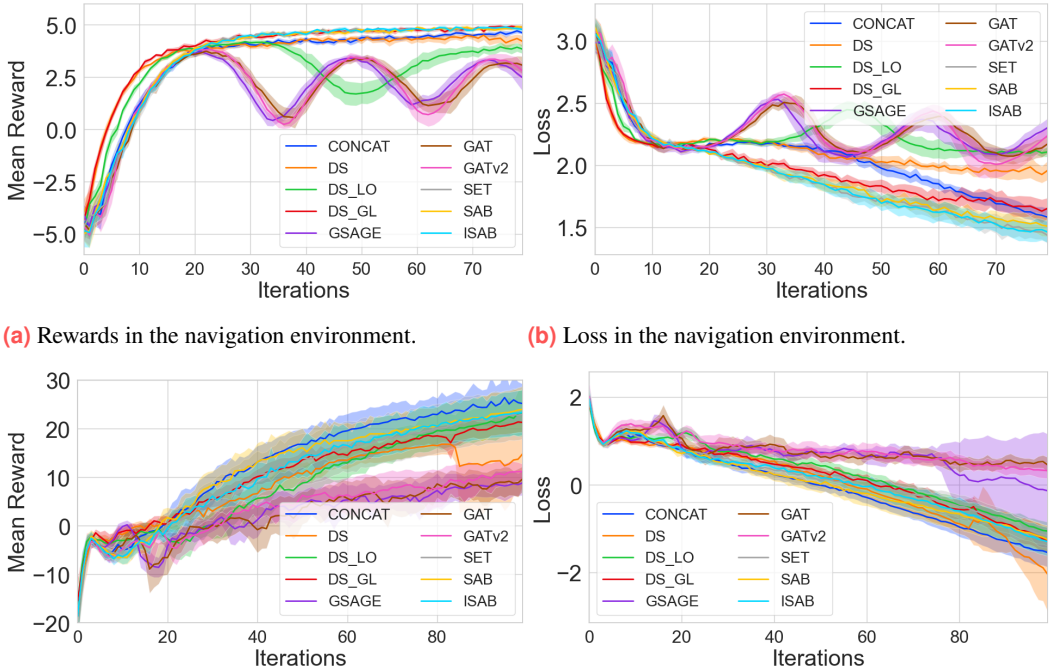
Fig. 5.13.: Comparison of selected KPIs of all methods in the balance environment.

Balance scenario. Plotting the rewards of the embedding techniques side by side with *CONCAT* shows that, overall, all techniques perform relatively similarly. *CONCAT* appears

to have better sample efficiency during the first 30 iterations, learning faster than the other approaches, apart from *SABTransformer*. However, after around 80 iterations, all methods achieve very similar rewards.

The training times of the methods reveal clearer differences. The fastest methods are *CONCAT*, *GSAGE*, and *DS_LOCAL*, each taking around 50 seconds or less to train the model for 80 iterations. Approximately double that time, around 100 seconds, is required for the other DeepSet and most transformer approaches. *GAT* and *SAB* stand out with the longest training times of nearly 150 seconds.

In contrast, the inference times are very similar across all models, ranging between 0.4 and 0.6 milliseconds. This shows that PyTorch is well optimized overall, despite the large differences in FLOPs. For instance, *DS* and *DS_GLOBAL* require around 300 million FLOPs, while *GSAGE* and *CONCAT* need only 11.5 and 37.6 million, respectively. The remaining methods average around 100 million FLOPs. The high FLOPs count for *DS* and *DS_GLOBAL* is due to the large decoder layers with 256 cells each, compared to 64 for *CONCAT* and 128 for *DS_LOCAL*. Interestingly, at 5 agents, *SABTransformer* has fewer FLOPs than *ISABTransformer*, due to the added MAB in each ISAB. However, as will be discussed in paragraph 5.6.2, the number of FLOPs for *SABTransformer* grows more quickly, and with 15 agents it requires more FLOPs than *ISABTransformer*. Full results are shown in Table B.1.



(a) Rewards in the navigation environment.

(b) Loss in the navigation environment.

(c) Rewards in the multi-give-way environment.

(d) Loss in the multi-give-way environment.

Fig. 5.14.: Cross method comparison in navigation and multi-give-way environments.

Navigation scenario. The results of training the different models in the navigation scenario are surprising (see Figure 5.14). The policy required here is conceptually simpler than in the balance scenario, as no actual cooperation is needed. Agents only need to reach their own goals while avoiding collisions (see subsection 4.2.3). For 5 agents, the maximum achievable reward is approximately 5, if all agents reach their goals in an episode, without colliding.

The training processes of the GNN methods and *DS_LOCAL* are very unstable. After around 20 iterations, the reward peaks, but then falls back to almost 0. The reward and loss oscillate strongly, and the rewards never reach the levels achieved by *CONCAT*. This oscillatory behavior can be explained by the CTDE setup with parameter sharing and homogeneous agents. While a centralized critic should stabilize learning, the shared policy must optimize across all agents simultaneously in a non-stationary environment. Policy updates that improve performance in some interaction patterns may worsen it in others. As training alternates between these phases of improvement and decline, the collective reward exhibits the observed oscillations. Nevertheless, the oscillations are highly regular, with improvement and deterioration phases of comparable duration and magnitude, suggesting that a deeper investigation would be needed to fully understand the underlying dynamics.

Other methods, such as *DS_GLOBAL* and the transformer-based approaches, achieve slightly higher rewards than *CONCAT*. Moreover, the DS methods also outperform *CONCAT* during the first 10 iterations, showing better sample efficiency in this scenario, which is not the case in the balance scenario. Training and inference times follow a similar pattern as in the balance scenario (see Figure 5.13) and are detailed in Table B.2.

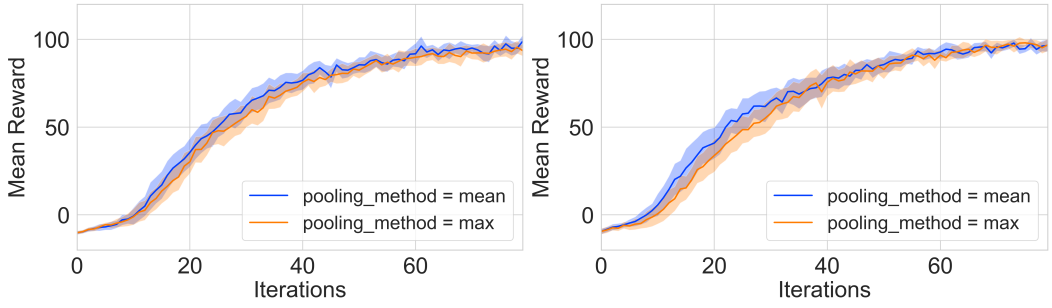
Multi-give-way scenario. In this scenario, training is more chaotic and unstable across all methods during the first 25 iterations. Rewards fluctuate without a clear pattern, unlike the oscillations observed in the navigation scenario. This behavior may be explained by the reward structure. The final reward for all agents reaching their goal is significantly larger than the intermediate rewards for not colliding. Once this terminal reward is discovered, optimization focuses on it. Early in training, however, substantial exploration is required, leading to unstable reward signals. Overall, the *CONCAT* method outperforms all other proposed techniques in this scenario. Transformer-based approaches perform second best, while GNN models perform worst in terms of reward, reaching only about 10 after 100 iterations. Notably, the *DS* method performed moderately well initially but experienced a sharp drop in performance around 85 iterations, leading to a collapse in its mean reward. This shows that even with MAPPO, which is meant to prevent the new policy from diverging too much, with the clipped objective, it doesn't entirely eliminate the risk of instability in these dynamic multi-agent systems.

5.5.1 Mean and Max Pooling

For the DeepSet methods and *GSAGE*, the type of aggregator function can be swapped between max and mean pooling:

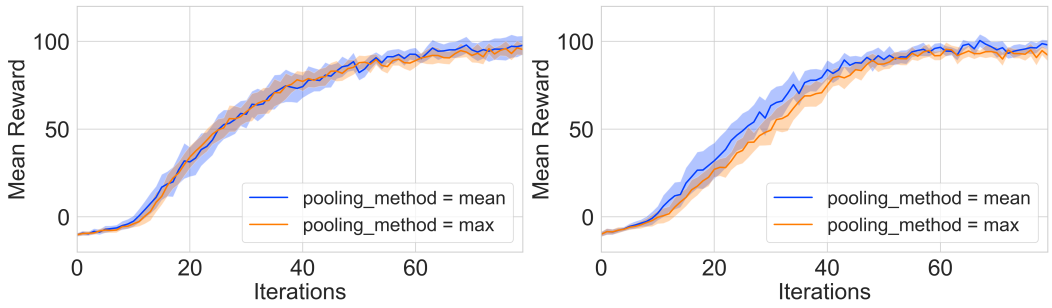
$$\text{Mean pooling: } y = \frac{1}{n}\sum_{X} x_i, \quad \text{Max pooling: } y = \max_{X}(x_i) \tag{5.1}$$

SwarmRL [8], which uses *DS_GLOBAL*, found that mean pooling worked best in their experiments. Consequently, all experiments using the DeepSets approach have used mean pooling as the default so far. *GSAGE* also supports different pooling aggregators. Their results showed that both mean and max pooling performed well and were faster than other methods, such as LSTM pooling. However, the differences in yielded rewards between pooling methods were marginal [10]. To investigate how the choice of aggregator function impacts DeepSet approaches and *GSAGE*, max pooling is used instead of the default mean pooling. The results are plotted in Figure 5.15. Similar to the findings in [10], the choice of pooling method had little impact on the models' rewards. The training and inference times for both pooling methods are also very similar (see Table B.4). The only noticeable difference among *DS*, *DS_LOCAL*, and *GSAGE* is in sample efficiency, with mean pooling achieving slightly higher rewards faster, although the final mean reward remains unchanged. The results of this experiment are very similar to the original SwarmRL [8] and GSAGE [10] paper, with mean pooling either slightly outperforming max pooling or both being very similar.



(a) Mean vs max pooling in *DS*.

(b) Mean vs max pooling in *DS_LOCAL*.



(c) Mean vs max pooling in *DS_GLOBAL*.

(d) Mean vs max pooling in *GSAGE*.

Fig. 5.15.: Comparison of rewards for mean and max pooling.

5.6 Scalability

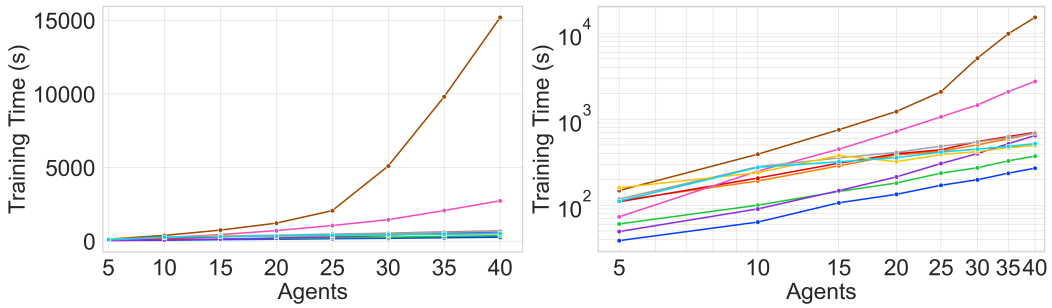
One of the main goals of this thesis is to identify models that scale efficiently without significantly increasing resource requirements. Therefore, in this chapter, the number of agents in the environment is increased, and the impact on the metrics is recorded. Due to the nature of scalability experiments, the training durations become too long to repeat 10 times for confidence intervals, as done in the hyperparameter tuning experiments. However, since this section focuses primarily on metrics that have shown low variance and small confidence intervals, this is not a concern for these experiments. The most important metrics in this analysis are the complexity metrics, training times and memory usage, which have been stable throughout this thesis, as well as FLOPs and the number of parameters, which are constant for a given model, further reducing the need for repeated measurements.

As explained in more detail in subsection 4.8.3, the metrics in relation to the number of agents are plotted on both linear and logarithmic scales to better understand how the metrics scale with the number of agents. A linear regression is computed on the log-transformed data, and the slope of the fitted linear model can be interpreted as the exponent of the relationship between the metric and the number of agents. For example, if a metric M scales quadratically with the number of agents N, i.e., $M(N) \propto N^2$, the exponent is 2, and the data will appear as a straight line in the log-log plots with a slope of 2. This way, the relationship can be approximated.

5.6.1 Runtime Metrics

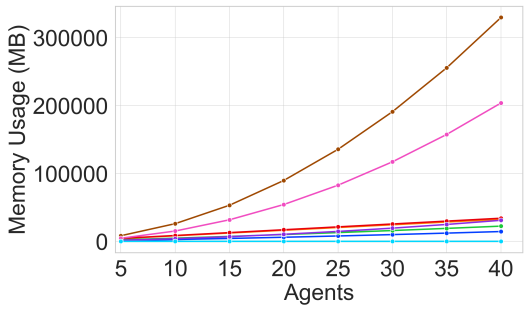
In this section, the impact of the number of agents on the main runtime metrics, training time and memory usage during training is examined. The graphs of the embedding methods with varying numbers of agents can be seen in Figure 5.16, and the results from the linear regression in Table 5.6.

Training time. As expected, training time increases with the number of agents. The *SABTransformer* embedding starts with the largest training duration at 5 agents, closely followed by *GAT* and the other transformer methods. However, the training times of *GAT* and *GATv2* increase faster than those of the other embedding methods. At 10 agents, *GAT* overtakes the transformer embeddings and becomes the method with the longest training time. At 15 agents, both GAT techniques have the longest training durations. In the log-log plot of training duration, *GAT* exhibits an upward curve, indicating that its training time grows faster than a simple power law relationship as the number of agents increases, exploding for 30 or more agents. Overall, the GNN methods scale the worst with increasing numbers of agents. The poor performance of *GATv2* is largely due to the model that performed best in terms of reward using 3 attention heads, which increases complexity. Moreover, all GNN-based methods rely on the PyTorch Geometric package, which raises a warning during training about potential performance degradation due to

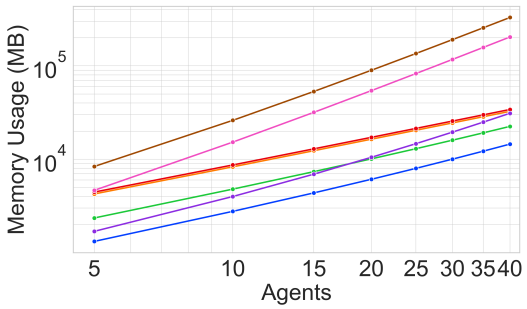


(a) Training time.

(b) Training time on logarithmic scale.



(c) Memory usage.



(d) Memory usage on logarithmic scale.

(e) Legend.

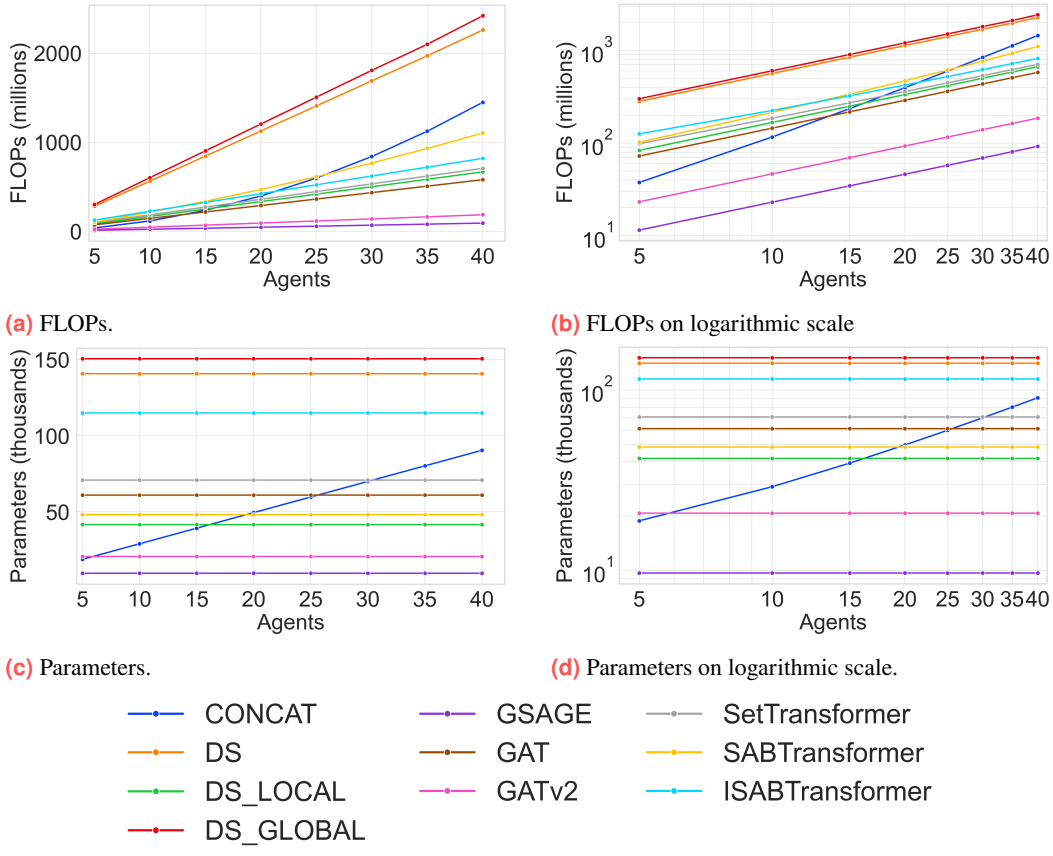
Fig. 5.16.: Impact of increasing the number of agents in the balance environment on the runtime metrics.

missing optimizations[1]. This could partially explain the training data trends of these methods.

CONCAT is the fastest method in absolute terms, growing roughly linearly with the number of agents, as indicated by its slope of 0.951 (see Table 5.6). In contrast, all DS and transformer methods have smaller slopes. *SABTransformer* and *ISABTransformer* have particularly low slopes of 0.524 and 0.679, respectively. For reference, a slope of 0.5 would correspond to a square root relationship. However, both have an $R^2 < 0.98$, which is the threshold for fitting a linear model in this thesis. This suggests that their scaling behavior is more complex than a simple power law relationship. Nevertheless, the transformer methods scale very well with increasing numbers of agents, as shown in Figure 5.16.

[1] UserWarning: There is a performance drop because we have not yet implemented the batching rule for aten::scatter_add_

Memory usage. Memory usage appears to grow very consistently, with the lines in the log-log plot being almost perfectly straight. This is confirmed by the R-squared values, which are at worst 0.997, indicating that all memory growth can be accurately approximated by linear regression on the log-transformed data. The graph attention methods have the largest slopes, meaning their memory usage grows faster than linearly, with *GATv2* and *GAT* approaching quadratic growth. The DS methods grow the slowest, roughly linearly or slightly superlinear, with smaller scaling exponents than *CONCAT*.



(a) FLOPs.

(b) FLOPs on logarithmic scale

(c) Parameters.

(d) Parameters on logarithmic scale.

(e) Legend to plots.

Fig. 5.17.: Impact of increasing the number of agents in the balance environment on the model complexity metrics. For the agent-count invariant methods FLOPs increase more slowly compared to *CONCAT*, and the parameters do not change with varying agent counts.

5.6.2 Model Complexity Metrics

The increase in runtime metrics with more agents in the environment is primarily due to the growing complexity of the models, which have more parameters and require more operations for each forward pass and backpropagation during training. Another aspect impacting the runtime metrics is how well the models are optimized for the hardware and how efficiently the code is implemented. Therefore, analyzing the model complexity

Metric	Strategy	Exponent (a)	R-squared	Complexity Class
Training Time	*CONCAT*	0.951	0.995	Roughly linear
	DS	0.862	0.996	Sublinear
	DS_LOCAL	0.882	0.994	Sublinear
	DS_GLOBAL	0.887	0.998	Sublinear
	GSAGE	1.254	0.984	Superlinear
	GAT	2.207	0.936	Upward Curve
	GATv2	1.709	0.997	Superlinear
	SetTransformer	0.791	0.972	Downward Curve
	SABTransformer	0.524	0.936	Downward Curve
	ISABTransformer	0.679	0.932	Downward Curve
Memory usage*	*CONCAT*	1.154	0.999	Superlinear
	DS	0.980	1.000	Roughly linear
	DS_LOCAL	1.086	0.999	Superlinear
	DS_GLOBAL	0.981	1.000	Roughly linear
	GSAGE	1.405	0.997	Superlinear
	GAT	1.774	0.999	Superlinear
	GATv2	1.824	0.999	Superlinear
FLOPs	*CONCAT*	1.763	0.999	Superlinear
	DS	1.001	1.000	Linear
	DS_LOCAL	1.000	1.000	Linear
	DS_GLOBAL	1.000	1.000	Linear
	GSAGE	1.000	1.000	Linear
	GAT	1.000	1.000	Linear
	GATv2	1.000	1.000	Linear
	SetTransformer	0.951	1.000	Roughly linear
	SABTransformer	1.145	0.999	Superlinear
	ISABTransformer	0.905	0.999	Sublinear
Parameters	*CONCAT*	0.763	0.995	Sublinear
	DS	0.000	0.000	Constant
	DS_LOCAL	0.000	0.000	Constant
	DS_GLOBAL	0.000	0.000	Constant
	GSAGE	0.000	0.000	Constant
	GAT	0.000	0.000	Constant
	GATv2	0.000	0.000	Constant
	SetTransformer	0.000	0.000	Constant
	SABTransformer	0.000	0.000	Constant
	ISABTransformer	0.000	0.000	Constant

Tab. 5.6.: Analysis with linear regression of the metrics. The exponent a reflects the scaling of the metric with regard to the number of agents N. E.g., if a metric M scales quadratically, $M(N) \propto N^a$, the exponent is 2, and the data will appear as a straight line in the log-log plots with a slope of 2. For more information on the empirical scaling analysis, see subsection 4.8.3. *Memory usage is not reported for transformer-based methods.

metrics, namely FLOPs and model parameters, is important to determine if they align with the observed runtime behavior.

FLOPs. Apart from *CONCAT*, all embedding methods are growing linearly in the linear plot. This is supported by the complexity analysis, in which most methods have an almost perfect R-squared value of 0.999 or even exactly 1, with slopes of the log-transformed data between 0.999 and 1.007. The only exceptions are *CONCAT* and the transformer methods. *CONCAT* grows faster than linearly, with a scaling exponent of 1.763, approaching a quadratic relationship between FLOPs and the number of agents. *SABTransformer* also grows faster than linearly, but only slightly, with an exponent of 1.145. In contrast, the induced points in *ISABTransformer* and *SetTransformer* are the only methods with slopes significantly smaller than one, making them the methods where FLOPs scales most efficiently. Due to the extra MAB blocks in the *ISABTransformer*, it initially has more FLOPs than *SABTransformer*, but at 15 agents, the *SABTransformer* surpasses it due to its faster growth.

Parameters. The number of parameters exhibits a very consistent pattern. All invariant embedding methods maintain a constant parameter count, which is a direct consequence of their invariance to the number of agents. In contrast, *CONCAT*'s parameters grow with an exponent of 0.763, highlighting the advantage of invariant methods. *CONCAT* starts as one of the methods with the fewest parameters at 5 agents (see Figure 5.17), but with 40 agents, it already ranks as the fourth largest in terms of parameter count.

5.7 Generalizability

This chapter deals with experiments that test the generalizability of the methods, i.e., how well the models perform on agent counts they were not trained on.

5.7.1 Training Data Composition

The invariance towards the number of agents in the environment of the proposed methods allows training on different numbers of agents. To gain insights into how the training set

should be structured, the models are trained on different agent counts, and their rewards are analyzed. Three different training sets are considered:

- High variance: $[4, 6, 8, 10, 12, 14, 16, 18]$
- Low variance: $[6, 10, 14, 18]$
- No variance: $[10]$

In total, the same number of iterations, 80, is run during training, with the iterations evenly split across the training set. For example, the high-variance set contains eight different agent counts, thus each agent count is trained for 8 iterations: 8 iterations with 4 agents, 8 iterations with 6 agents, and so on. The trained models are then tested on two unseen agent counts, 5 and 20, and on one agent count from the training data, namely 10 agents. The training sets are structured to evenly spread the range of the testing agents.

This setup allows for analyzing the resulting rewards on the testing set to determine the optimal way to structure the training data. These results are, however, specific to the balance scenario considered here, broader insights would require repeating the experiment in different environments. It is also insightful to observe whether different invariant methods perform better or worse than others. In this experiment, the *CONCAT* method is included even though it is not agent-count invariant. Thus, the critic must always be trained from scratch for each agent count in the training set. Since the policy is independent of the number of agents, i.e., the observation vector remains the same regardless of the number of agents, the policy can still be trained and evaluated across different agent counts.

Results. The resulting mean rewards of the testing agents are shown as a heatmap in Figure 5.18. Since the critic for *CONCAT* must be trained from scratch for each agent count, it achieves lower rewards on high-variance training datasets compared to all other invariant methods when testing on 5 and 10 agents. For the low-variance training set, *CONCAT* also performs worse than most other methods. Across all tested methods, the rewards for an agent count of 20 are generally low, with none exceeding a reward of 70, regardless of the training set.

Overall, the transformer methods perform the best, consistently yielding the highest rewards or performing competitively with the best methods in the category. Particularly, when generalizing to 5 agents, all transformer methods achieve rewards around or above 80, whereas other methods mostly achieve rewards between 50 and 75. Notably, when testing the trained policies on 5 agents, the low-variance training set performs worse than the high-variance and no-variance for all invariant methods. This is likely because five agents are outside the low-variance training data, which includes agent counts in the interval $[6, 18]$. In contrast, the high variance set is made up of agent counts from four to eighteen.

Testing the models on 10 agents produces the best results, as this agent count is included in each training set. Training with no variance yields the highest rewards, since all 80

Fig. 5.18.: Heatmap resulting rewards from different training data compositions. The cells shows the rewards of testing the trained policy on the specific training dataset for an agent count. For example, the column *low_20* shows the resulting mean reward of a model trained with a low variance training dataset on an environment with twenty agents. The mean reward is the average of the mean reward for running the experiment 10 times. The rows specify which model was trained.

iterations are focused on 10 agents. Nevertheless, for invariant methods, low-variance and high-variance sets also achieve good performance.

In conclusion, the transformer methods are the only models that consistently generalize well across different agent counts, particularly to 5 agents, regardless of whether the training set has high, low, or no variance. Additionally, the experiment shows that testing on agent counts not included in the training set, such as 20 agents or 5 agents in the low-variance set, generally results in lower rewards. The exception is the no-variance scenario evaluated on 5 agents, where all methods perform reasonably well, except for *GSAGE* and *GAT*.

5.7.2 Train Low – Test High

For the experiments in this section, the models are trained on a low number of agents, in this case 5, for 80 iterations, and then the trained policy is evaluated on an increasingly high number of agents, up to 20. For each agent count, 10 rollouts are performed. This tests how well the methods create policies that are capable of generalizing to higher agent counts in the environment. For this experiment, the balance and navigation scenarios are used, as the multi-give-way scenario only works with 4 agents.

Balance. The maximum reward in the balance scenario is around 100 for the models after 80 iterations, regardless of the number of agents in the environment. How the methods

perform, generalizing from being trained on 5 agents to being evaluated on 5, 6, 10, 15 and 20 agents, can be seen in Figure 5.19a.

Up until 10 agents, the rewards are fairly high, with all methods performing well. Running the policy with 6 agents actually achieves a higher mean reward, averaged over all methods. For larger agent counts of 15 and 20, the rewards drop significantly, to values under 70 compared to over 90 for lower agent counts. Especially at 20 agents, bigger differences between the yielded rewards are evident. While the transformer methods almost reach a reward of 75, *DS_LOCAL* only manages a reward of 48.6, with the other DS methods only being marginally better. *CONCAT* yields a reward of 61.7, which is behind all transformer methods, *GSAGE* and *GATv2*.

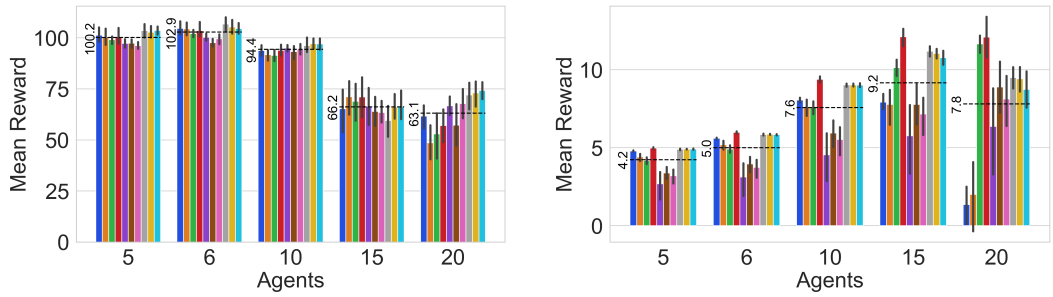
Navigation. The reward structure for the navigation scenario means that the maximum possible reward grows with the number of agents. This is because each agent gets a reward for reaching its goal, and all individual rewards are summed up at the end of an episode. As a rule of thumb, the maximum reward is roughly the number of agents, thus, for 5 agents, the best reward is 5, and for 20 agents, it would be a reward of 20.

In general, the GNN methods underperform in this experiment because of their instabilities in training, even after 80 iterations, which can be seen in Figure 5.14. The transformer methods, on the other hand, yield above-average rewards for all agent counts, with very similar rewards among the transformer variants. Similarly, *CONCAT* achieves above-average results up until 10 agents. For 15 and particularly 20 agents, the rewards collapse. At 20 agents in the environment, the reward drops to 1.4, compared to the average of the methods at this agent count of 7.8. Similar to *CONCAT*, *DS* also performs poorly for 20 agents with a mean reward of only 2.0. This stands in stark contrast to the other DS methods, *DS_LOCAL* and *DS_GLOBAL*, which perform best at 20 agents, with a reward of 11.7 and 12.1 respectively.

Large differences between methods. This experiment showcased how differently the policies from the methods manage to generalize to higher agent counts. Both in the balance and navigation scenario, the method choice greatly influenced the outcome when generalizing to agent counts three or four times higher than what was trained on. Even within a method category, the results can differ greatly, as is the case for DS in the navigation scenario. Overall, the transformer methods stand out as being the only methods that perform above average across both environments in almost all agent count variations, showcasing their robustness.

5.7.3 Training Mostly Low

This experiment extends the one in which the policy was trained on low agent counts and evaluated on higher ones (see subsection 5.7.2). Since training exclusively on low



(a) Balance scenario, trained on 5 agents.

(b) Navigation scenario, trained on 5 agents.

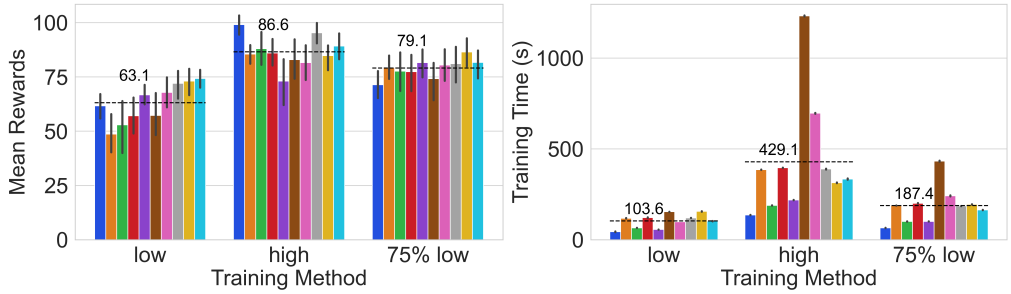
(c) Legend.

Fig. 5.19.: Rewards of training low and generalizing to higher agent counts.

agent counts and rolling the policy out with higher counts, e.g., 20 agents, resulted in only mediocre performance, this experiment investigates whether training the policy for the majority of iterations on a low agent count, 75% with 5 agents, and the remaining iterations with a high agent count, 25% with 20 agents, improves performance. The results are compared against training exclusively in the high-agent domain and exclusively in the low-agent domain, as in the previous experiment (see subsection 5.7.2).

Results. In Figure 5.20, the achieved rewards and training times of the different training strategies are shown. Training for just the last 20 iterations with the high agent count (25%) increases mean rewards by around 25% on average. The DS methods, in particular, benefit compared to training exclusively on 5 agents. *GSAGE* even achieves better performance when trained 25% on high agents than when trained solely on high agents. Similar to the other generalizability experiments (see subsection 5.7.1), *CONCAT* performs poorly when trained on varying agent counts, as the critic must be retrained from scratch for each count. Consequently, the improvement from including some high-agent iterations is small. Although the invariant methods trained mostly on low agent counts do not reach the reward levels of those trained exclusively on high agent counts, the training times reveal the key advantage of this approach. Training with the more computationally expensive high agent count for only 25% of the iterations reduces the average training duration to less than half of the full high-agent training setup.

In general, this experiment demonstrates the flexibility of invariant training methods, as they allow reducing training time by limiting the proportion of high-agent iterations while still maintaining strong generalization performance.



(a) Rewards of training methods.

(b) Training times of training methods.

(c) Legend.

Fig. 5.20.: Rewards of training low and generalizing to higher agent counts in the balance scenario. The average of the methods among one training method is displayed as a dotted line, with the average written above it. Three training methods are plotted, training 80 iterations with 5 agents: low, training all 80 iterations with 20 agents: high, and training 75%, i.e., 60 iterations, with 5 agents and 25% with 20 agents. The mean rewards are the results of rolling the trained models out in an environment with 20 agents.

5.8 MAPPO with Centralized Execution

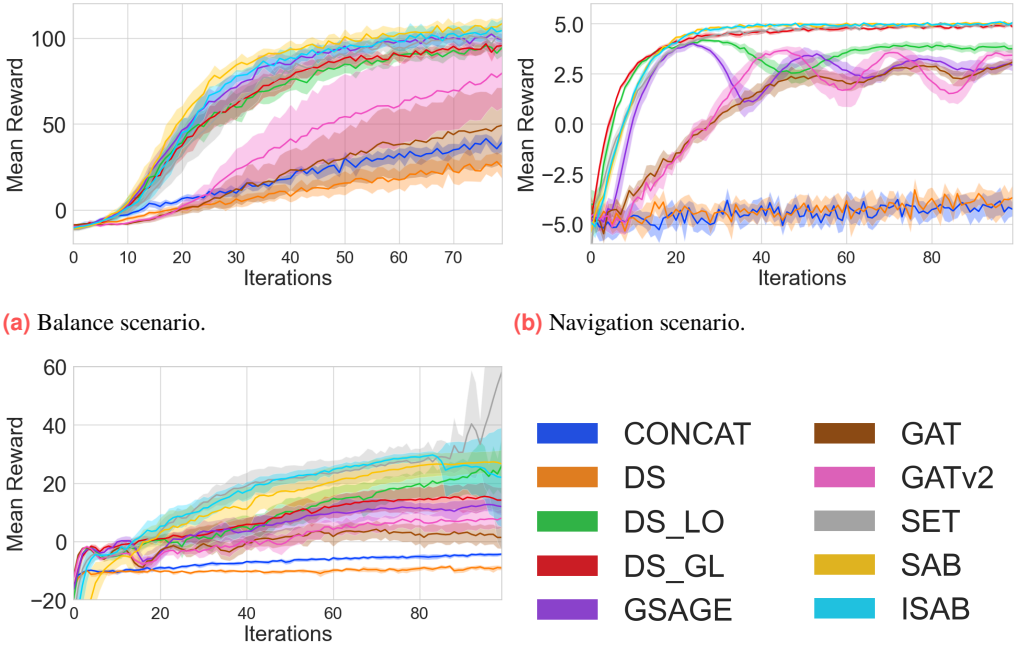
In all experiments in this thesis, the CTDE paradigm has been applied, where policies are trained using centralized information but executed in a decentralized manner, meaning that agents have access only to their own observations during execution. In contrast, CTCE employs both centralized training and centralized execution, allowing both actor and critic networks to access the full global state during training and execution (see paragraph 4.1). This section investigates how the performance of different embedding methods changes when applying CTCE instead of CTDE, as was the case in section 5.5. The hyperparameters for MAPPO, the environments, and the networks architectures are reused from the prior CTDE hyperparameter tuning.

Results. Across all scenarios, the invariant embedding methods demonstrate a clear advantage in the CTCE paradigm. In the balance scenario, *CONCAT* achieves lower rewards and learns more slowly, with only *DS* performing worse. All other methods, except *GAT* and *GATv2*, perform comparably to the tuned CTDE setup.

Similarly, in the navigation scenario, *CONCAT* and *DS* perform poorly, showing little to no learning even after 100 iterations. As in CTDE, the GNN techniques result in oscillating

behavior. Apart from *DS_LOCAL*, which converges to a suboptimal policy, the remaining methods, *DS_GLOBAL* and the transformer-based approaches, perform comparably to their results in the CTDE setup.

In the multi-give-way scenario, the models display a wide range of resulting rewards. *CONCAT* and *DS* again perform poorly, with negative rewards after 100 iterations, while other methods achieve rewards ranging from near 0 to nearly 60. The transformer methods perform best and learn the fastest. However, *ISABTransformer* experiences a sudden collapse in rewards, similar to *DS* in the CTDE setup (see Figure 5.14). Notably, *SetTransformer* exhibits a sudden improvement after around 90 iterations, with rewards jumping from approximately 30 to nearly 60, though with a large confidence interval.



(a) Balance scenario.

(b) Navigation scenario.

(c) Multi-give-way scenario.

Fig. 5.21.: MAPPO with CTCE in the balance, navigation, and multi-give-way scenarios.

Limitations. The results of this experiment should be interpreted with caution. The hyperparameters were directly transferred from the CTDE setup without additional tuning. Therefore, the poor performance of methods such as *CONCAT* or *DS* does not imply that these approaches cannot be improved. In particular, in the navigation scenario, the learning curves of *CONCAT* and *DS* suggest instabilities that may be mitigated by tuning the learning rate or other hyperparameters. Nonetheless, the experiment highlights the importance of selecting an appropriate architecture, which significantly impacts MARL performance. Additionally, the results emphasize once again the robustness of transformer-based methods, which perform consistently well across all scenarios and experiments presented in this thesis.

Discussion

6

This chapter deals with interpreting the findings from the experiments chapter on a higher level and put them in the broader context of this thesis in terms of related work and the research questions formulated in section 1.3.

6.1 Key Findings

Across the experiments, four central findings emerged:

1. *CONCAT* remains a strong baseline, but several PI critics, particularly transformer-based ones, reached comparable performance while offering theoretical advantages and more flexibilty in the training process.

2. Scalability analyses confirmed that PI critics handle growing agent populations better than *CONCAT*, despite higher initial training costs.

3. Generalizability experiments showed that invariant critics can extend to unseen agent counts without retraining, an advantage over *CONCAT*.

4. The proposed transformer based methods, *SABTransformer*, *ISABTransformer* and *SetTransformer* stand out as being competitive across all experiments and scenarios.

6.1.1 Performance of proposed PI methods (RQ1)

The primary objective of this work was to design, implement, and evaluate permutation-invariant (PI) and agent-count invariant methods for the centralized critic in MAPPO. These methods were expected to be competitive with the often used *CONCAT* strategy, which is neither PI nor can handle varying agent counts natively. Despite its simplicity, *CONCAT* proved to be a strong and difficult-to-beat baseline, making it a suitable reference point throughout the experiments.

Hyperparameter tuning. A large part of the experiments focused on tuning hyperparameters for the proposed PI methods in the balance scenario (subsection 4.2.2). This was essential for achieving transparent and reproducible results, which are often underreported

in related work. The hyperparameter tuning was challenging due to the large number of parameter options. In addition to architectural choices for the critics, both MARL algorithm parameters and environment-specific settings can be adjusted. This results in a high number of tunable options that can not be fully tuned due to resource constraints. Parameters for MAPPO and VMAS were therefore largely adopted from related work, while critical architecture hyperparameters were tuned directly. This tuning was necessary to provide a fair assessment of how PI critics perform in MAPPO for each distinct architecture. Otherwise, differences in performance would be tied primarily to parameter selection rather than the underlying architectures. The tuned models, therefore, represent each method's best case in the balance scenario, allowing for meaningful comparisons. The tuning process revealed that many hyperparameters had only a small influence on the achieved rewards, e.g., the encoder width in DS methods, or the number of attention heads in transformer-based critics.

Overall performance of all methods across environments

With all methods tuned in the balance scenario, they can be compared with each other. They were analyzed in the balance scenario, as well as in the multi-give-way and navigation scenarios, to test the robustness of the methods.

Balance scenario. In the balance environment, all methods could be tuned to reach competitive reward levels compared to *CONCAT*. However, training efficiency differed considerably: lightweight models such as *DS_LOCAL* or *GSAGE* achieved comparable training times, whereas larger architectures like *GAT* and *SABTransformer* required up to three times longer training times (see Figure 5.13).

Navigation scenario. The outcomes of the navigation scenario varied more than in the balance scenario. Several methods slightly outperformed *CONCAT* either in absolute rewards reached or in learning speed. In particular, DS-based critics achieved better learning efficiency, reaching a mean reward greater than 2 after only 10 iterations, while all other methods, including *CONCAT*, are at approximately 1. Meanwhile, transformer-based methods, especially *SetTransformer* and *SABTransformer*, are consistently between 5 and 10 percent higher than *CONCAT* after 40 iterations onwards. In contrast, GNN-based methods exhibited severe training instabilities, with oscillating rewards (see Figure 5.14).

Multi-give-way scenario. This environment also produced mixed results. As in navigation, GNN critics underperformed regarding their achieved rewards, whereas DS and transformer methods were more robust. Transformer critics in particular achieved rewards close to *CONCAT*, though none of the PI methods surpassed the baseline (see Figure 5.14).

Instabilities during learning. Across the navigation and multi-give-way scenarios, instabilities were especially pronounced in GNN-based critics. These fluctuations may point to optimization challenges or insufficient adaptation of MAPPO to such architectures. Further methodological adjustments, such as altering clipping thresholds, may be necessary to stabilize training. Since the hyperparameters tuned in the balance scenario were reused in the other scenarios, testing the methods in the untuned scenario provides a robustness check of the methods. These instabilities also demonstrate the overall difficult learning task in MARL with dynamic agents, which influence one another, leading to the moving target problem in MARL (see paragraph 2.1.2).

6.1.2 Scalability (RQ2)

A key motivation for developing agent-invariant critics is their ability to handle environments with varying numbers of agents without an increase in model size. This section evaluates the experiments concerning scalability in terms of network parameters, computational cost, training time, and memory usage.

FLOPs and parameters. The analysis in section 5.6 highlights the invariant methods' strongest advantage. As the number of agents increases, the network size for agent-count invariant critics remains constant, as they can handle an arbitrary number of agents in the environment (see Figure 5.17). Moreover, the required FLOPs per forward pass for the invariant methods increase only linearly, with the exception of the *SABTransformer*, whereas *CONCAT* sees near-quadratic growth.

Training times and memory usage. In practice, the relationship between scalability and training efficiency is not as straightforward as for FLOPs and parameters. Despite its theoretical disadvantages, *CONCAT* achieved the lowest absolute training times and memory usage due to its simplicity and highly optimized implementation, even for high agent counts (see Figure 5.16). However, the growth trends reveal that DS and transformer-based critics scale better, with training time increasing at a slower rate as the agent count rises. The *SABTransformer* is particularly efficient, showing an almost square-root growth trend in its training time. Memory usage exhibited similar patterns, with DS critics in particular having only modest growth. These results suggest that although PI critics appear more expensive in absolute terms for small environments, they offer significantly better scalability for larger agent populations. For example, in hypothetical scenarios with hundreds of agents, *CONCAT* would likely become impractical.

The notable exceptions are GNN-based methods, which scale almost quadratically in some instances of training time and memory usage, while the other agent-count invariant methods and *CONCAT* grow approximately linearly. This may stem from less optimized implementations in the underlying code, but it also points to inherent challenges in applying GNN critics at scale with fully connected graphs.

6.1.3 Generalizability (RQ3 & RQ4)

Generalizability is another central aspect of the proposed PI and agent count invariant critics. By design, they can process varying agent counts without needing to retrain the networks from scratch in the training phase. This makes them suitable to be deployed across environments with different agent populations, raising questions about training data composition and the overall generalizability potential of the methods. The experiments are all additionally run with the *CONCAT* strategy. The trained policies can be applied to environments with arbitrary agent counts, since the observation vectors do not grow or shrink with the agent count. However, the critic is centralized, and for *CONCAT*, training on different agent counts entails training the critic from scratch for each distinct agent count in the training process.

Training data composition. Drawing general conclusions from the experiments concerning the training data composition is difficult, as the results of the different compositions were very similar (see subsection 5.7.1). Policies trained on multiple agent counts performed similarly to those trained on only a few in the balance scenario. This suggests that generalization across unseen agent counts is possible. Performance declined when tested on agent counts outside the training range. For instance, models trained only on 6 to 18 agents underperformed when evaluated on 5 agents, whereas training on a broader range (4 – 18 agents) yielded better results. Transformer critics performed best in these rollouts, especially at low agent counts. By contrast, policies trained with *CONCAT* struggled more with high training data variance, because its critic must be retrained for each distinct agent configuration, hindering the critic's learning.

Limits of generalizability. When testing the generalizability of the trained policies, it is also important to analyze the limits of the models, i.e., at which agent count the performance of the policies deteriorates notably. Policies trained on 5 agents performed reasonably well up to double that size. Beyond this, performance dropped significantly, with clear differences between the methods. In the navigation scenario, testing with 20 agents, DS methods degraded substantially, while transformer and some GNN critics maintained stronger performance. In the multi-give-way scenario, *CONCAT* yielded barely any positive reward at 20 agents, while all agent invariant methods still produced good rewards, apart from *DS*, which performed similarly to *CONCAT*.

Mixed training on low and high agent counts. An additional experiment tested whether training mostly on low agent counts, and only partially (25%) on higher counts, could substitute for training fully with high agent environments. The goal is to generate capable policies for high agent count environments with fewer resources. The results showed noticeable improvements compared to training solely on low counts, though they did not match the performance of full high-count training. Nonetheless, this hybrid strategy highlights the flexibility of agent count invariant critics, which gained the largest performance boost under such mixed setups, while keeping training costs manageable.

Comparison with *CONCAT*. Across all generalizability experiments, *CONCAT*-trained policies achieved competitive performance. *CONCAT* only underperformed significantly in the navigation scenario when generalizing to high agent counts (see Figure 5.19b). However, the lack of invariance to the number of agents requires retraining the critic for every new agent count configuration, limiting flexibility and efficiency. In contrast, invariant critics maintained performance across a wide range of agent counts without retraining, making them more suitable for environments with variable or unpredictable population sizes, particularly during training.

6.2 Relation to Existing Research

This section discusses how the results of this thesis fit into and extend existing research on permutation- and agent-count invariant critics.

6.2.1 Mean and Max Pooling

The aggregator function is a key design decision in DS-based and GraphSage-based critics. Prior work on DS methods [8] and GraphSage [10] found that mean pooling typically performs equally well or slightly better than max pooling. The pooling experiments within the balance scenario (section 3.3) confirm this finding. Mean and max pooling achieved very similar performance across all DS variants and *GSAGE*, with a minor advantage for mean pooling in terms of sample efficiency for *GSAGE* and *DS_LOCAL*. This consistency reinforces the choice of mean pooling as the default in invariant critics.

6.2.2 Architecture Guidelines

Similar to findings in SwarmRL [8], the hyperparameter tuning experiments confirmed that shallow architectures are often sufficient. For DS-based methods, decoders with only two layers worked best. In some cases, notably *DS* and *DS_LOCAL*, removing the encoder entirely yielded the strongest performance in the balance scenario. The only exception was *DS_GLOBAL*, which benefited from a dedicated encoder, though still with a small width of 16.

For GNN-based critics, a preprocessing step, encoding the input with a single-layer neural network before applying the graph module, proved most effective. However, due to the instabilities observed in training, no strong or universal guidelines could be drawn for GNN variants.

In the transformer methods, the *SetTransformer* implementation aligned with its original proposal [39]. The best-performing configuration utilized two SAB and two ISAB blocks,

which aligns with the architecture described in the source paper. Therefore, this configuration seems to be a reasonable starting point.

6.2.3 Sample Efficiency (RQ5)

A central motivation for developing PI critics in MARL is their potential to reduce the curse of dimensionality and therefore improve sample efficiency compared to non-PI approaches like *CONCAT* [7]. However, the experiments in this thesis provide only limited evidence for sample efficiency gains when using PI critics. In the balance scenario, only the *SABTransformer* resulted in a slight improvement of the sample efficiency, while in the multi-give-way scenario, no method surpassed *CONCAT*. In the navigation scenario, the DS-based methods managed small improvements of the concatenation strategy, but only at the beginning of the training phase. Overall, the expected consistent advantage of PI methods in terms of sample efficiency could not be confirmed.

The one clear case of improved efficiency appeared with the CTCE training setup (section 5.8), in which nearly all PI methods, except for *DS_LOCAL*, outperformed *CONCAT* in all scenarios. While this suggests that PI critics may unlock efficiency benefits in alternative training regimes, CTCE is less commonly used than CTDE, limiting the broader impact of this finding.

6.3 Limitations

This thesis has several limitations that affect the generality and interpretation of its results.

Environment choice. A limiting factor is the reliance on the VMAS environments. While these provide a flexible and standardized testbed (see section 4.2), they impose two important constraints. First, cooperation requirements remain relatively rudimentary, for example, in the navigation scenario, cooperation required by the agents is limited, and the communication is relatively simple. Second, scalability is restricted, as the environments support at most around 40 agents before training and testing become too expensive. As a result, the learned policies depend only weakly on the number of agents, and the ability to test strong generalization across environments with many agents is limited. Consequently, the insights to RQ3 and RQ4 about generalizability must be interpreted with caution.

Methodological issues. The experimental methodology introduces further limitations. Training instabilities in the GNN-based critics complicated the comparison, and it remains unclear whether these issues are from fundamental architectural challenges or the lack of hyperparameter tuning for the multi-give-way and navigation scenario. Especially,

the scalability issues for GNN-based methods could also be attributed to an inefficient implementation of the GNN mechanism by PyTorch, or the graph generation, which was implemented by the author. Similarly, the surprisingly poor performance of *CONCAT* in the CTCE setup may be attributable to unfavorable learning parameter configurations rather than a true structural weakness. These factors suggest that some of the observed differences may be partially due to implementation details rather than reflecting only architectural properties.

6.4 Implications

This thesis has several implications for research and practice in MARL. First, while simple strategies like *CONCAT* remain very strong baselines, permutation- and agent-count invariant critics, especially transformer-based ones, offer advantages for scalability and generalization. The scalability results show that invariant critics are essential in large systems such as swarm robotics, where concatenation-based critics become infeasible. Therefore, in settings with variable or large agent populations, these methods should be considered instead of permutation-sensitive methods. In dynamic environments, in which variable agent counts can be encountered, the case for agent-count invariant methods is even stronger, since they do not require retraining for each distinct agent count in training. In sum, invariant critics are a practical tool for scaling Multi-Agent Reinforcement Learning, motivating future research on training regimes, GNN stability, and applications beyond benchmark environments.

7 Conclusion

This thesis investigates permutation-invariant methods for the centralized critic in MAPPO with the ability to handle a variable-sized agent population. With these features, the methods supposedly offer more flexibility in training and better sample efficiency. The aim was to evaluate the competitiveness against the commonly used method of simply concatenating all agent observations as input for the critic network, which is neither PI nor invariant to the number of agents in the environment.

Nine different critic architectures were developed and evaluated based on prior research of PI and agent count invariant techniques from the domains of DeepSet, GNN, and transformers. These methods were tuned and benchmarked in VMAS environments. The experiments analyzed the adaptability, generalizability, and scalability across the scenarios for the different methods.

This work demonstrated that PI critics can match and sometimes exceed the concatenation strategy (RQ1), though the baseline remains a strong point of reference. A scalability analysis (RQ2) revealed that the developed agent count invariant critics scale better overall with the number of agents in terms of parameters and FLOPs, even if their absolute runtime was not lower at the tested magnitudes. In addition, scaling trends for the training times showcased the superior scalability of the transformer methods. With respect to generalization (RQ3, RQ4), the invariant critics proved capable of extending to unseen agent numbers when trained across a sufficiently broad range of agent counts. The performance degraded once the number of agents exceeded roughly double the training domain. Finally, while the theoretical sample efficiency gains (RQ5) were not apparent under the popular CTDE paradigm, PI critics showed clear advantages using CTCE. This suggests that the theoretical advantages might not have been captured by the scenarios in CTDE, but in other instances, the sample efficiency gains are evident.

Overall, while the performance of the concatenation strategy of creating a centralized critic remains strong, PI and agent count invariant methods provide benefits in scalability and generalizability, suggesting they are a promising direction for MARL in large and dynamic environments. Particularly, the methods based on the SetTransformer proved robust and competitive in all experiments conducted throughout the thesis.

7.1 Future Work

There are several directions that could be further explored, focusing on addressing methodological limitations, exploring alternative architectures, developing more challenging environments, and handling variable-sized observations.

Training instabilities. Future research should investigate the training instabilities observed in the GNN-based critics. These fluctuations may stem from unsuitable learning parameters, fully connected graph structures, or optimization challenges. Exploring alternative graph topologies that better reflect the underlying agent topology, rather than connecting all agents fully, could also stabilize training and improve performance. GSAGE already has mechanisms that could be used to further mitigate the scalability issues, such as sampling only a portion of the neighborhood from each agent.

Exploration of alternative architectures. Beyond the architectures evaluated in this thesis, others may offer similar or better advantages in scalability and expressivity. Graph Transformer architectures such as GTAT [41] or hyper-network [7] approaches for generating embeddings represent promising directions, potentially improving representation quality and flexibility further.

More challenging environments. Current experiments were constrained by environments with limited cooperation requirements and relatively small maximum agent counts. Future work could focus on developing environments that demand more complex coordination and support larger agent populations, allowing for a more thorough evaluation of scalability, generalization, and learning dynamics in high agent count scenarios.

Handling variable-sized observations. In scenarios with agent-to-agent communication, observation vectors possibly vary in size depending on the number of agents in range. The invariant methods proposed for critic design could be adapted to allow agents to process variable-sized observations during execution. However, permutation equivariance would be more critical in these scenarios, or a hybrid approach could be applied: standard neural networks could process the fixed-size portion of the observations, while size-invariant modules handle the variable-sized components, enabling flexible and generalizable agent policies.

7.2 Outlook

In closing, this thesis highlights several promising directions for advancing multi-agent reinforcement learning. PI and agent count invariant critics have shown the potential to improve scalability and adaptability, especially in settings with large or dynamic agent populations. As MARL continues towards increasingly realistic applications, the ability to efficiently handle variable numbers of agents will become critical. Integrating the insights gained here with advances in scalable and more flexible architectures may help to construct more robust and generalizable MARL systems in the future. Ultimately, the methods explored in this thesis could serve as building blocks toward integrating controlled benchmark experiments and the complex, dynamic nature of real-world multi-agent systems.

Bibliography

[1] M. I. Jordan and T. M. Mitchell. “Machine learning: Trends, perspectives, and prospects”. In: *Science* 349.6245 (2015), pp. 255–260. eprint: https://www.science.org/doi/pdf/10.1126/science.aaa8415 (cit. on p. 1).

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018 (cit. on pp. 1, 7, 17).

[3] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. *Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms*. 2021. arXiv: 1911.10635 [cs.LG] (cit. on pp. 1, 7, 8).

[4] Dom Huh and Prasant Mohapatra. *Multi-agent Reinforcement Learning: A Comprehensive Survey*. 2024. arXiv: 2312.10256 [cs.MA] (cit. on p. 1).

[5] OpenAI. *ChatGPT*. https://chat.openai.com. Available at https://chat.openai.com. 2022 (cit. on pp. 1, 29).

[6] Grammarly, Inc. *Grammarly: AI Writing Assistance*. https://app.grammarly.com/. Accessed: 2025-07-24. 2024 (cit. on p. 1).

[7] Xiaotian Hao, Hangyu Mao, Weixun Wang, et al. *Breaking the Curse of Dimensionality in Multiagent State Space: A Unified Agent Permutation Framework*. 2022. arXiv: 2203.05285 [cs.LG] (cit. on pp. 2, 4, 10, 98, 102).

[8] Maximilian Hüttenrauch, Adrian Šošić, and Gerhard Neumann. “Deep reinforcement learning for swarm systems”. In: *J. Mach. Learn. Res.* 20.1 (Jan. 2019), pp. 1966–1996 (cit. on pp. 2, 3, 80, 97).

[9] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, et al. *Deep Sets*. 2018. arXiv: 1703.06114 [cs.LG] (cit. on pp. 2, 10–13, 15, 24, 32, 45, 46, 56).

[10] Will Hamilton, Zhitao Ying, and Jure Leskovec. “Inductive Representation Learning on Large Graphs”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, et al. Vol. 30. Curran Associates, Inc., 2017 (cit. on pp. 2, 12, 21, 26, 33, 34, 47, 48, 56, 80, 97).

[11] Petar Veličković, Guillem Cucurull, Arantxa Casanova, et al. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML] (cit. on pp. 2, 13, 26, 29, 34, 35, 37, 47–49).

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL] (cit. on pp. 2, 12, 29, 30, 34, 36, 37).

[13] Chao Yu, Akash Velu, Eugene Vinitsky, et al. *The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games*. 2022. arXiv: 2103.01955 [cs.LG] (cit. on pp. 2, 4, 10).

[14] Christopher Amato. *An Introduction to Centralized Training for Decentralized Execution in Cooperative Multi-Agent Reinforcement Learning*. 2024. arXiv: 2409.03052 [cs.LG] (cit. on pp. 2, 3).

[15] Jae Won Lee and Jangmin O. "A Multi-agent Q-learning Framework for Optimizing Stock Trading Systems". In: *Database and Expert Systems Applications*. Ed. by Abdelkader Hameurlain, Rosine Cicchetti, and Roland Traunmüller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 153–162 (cit. on p. 7).

[16] Jangmin O, Jongwoo Lee, Jae Won Lee, and Byoung-Tak Zhang. "Adaptive stock trading with dynamic asset allocation using reinforcement learning". In: *Inf. Sci.* 176.15 (Aug. 2006), pp. 2121–2147 (cit. on p. 7).

[17] J. Cortes, S. Martinez, T. Karatas, and F. Bullo. "Coverage control for mobile sensing networks". In: *IEEE Transactions on Robotics and Automation* 20.2 (2004), pp. 243–255 (cit. on p. 7).

[18] Jongeun Choi, Songhwai Oh, and Roberto Horowitz. "Distributed learning and cooperative control for multi-agent systems". In: *Automatica* 45.12 (2009), pp. 2802–2814 (cit. on p. 7).

[19] Cristiano Castelfranchi. "The theory of social functions: challenges for computational social science and multi-agent learning". In: *Cognitive Systems Research* 2.1 (2001), pp. 5–38 (cit. on p. 7).

[20] Joel Z. Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. *Multi-agent Reinforcement Learning in Sequential Social Dilemmas*. 2017. arXiv: 1702.03037 [cs.MA] (cit. on p. 7).

[21] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. *Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving*. 2016. arXiv: 1610.03295 [cs.AI] (cit. on p. 7).

[22] Ming Zhou, Jun Luo, Julian Villella, et al. *SMARTS: Scalable Multi-Agent Reinforcement Learning Training School for Autonomous Driving*. 2020. arXiv: 2010.09776 [cs.MA] (cit. on p. 7).

[23] Maximilian Hüttenrauch, Adrian Šošić, and Gerhard Neumann. *Deep Reinforcement Learning for Swarm Systems*. 2019. arXiv: 1807.06613 [cs.MA] (cit. on pp. 7, 10, 12, 15, 25, 26, 31–33, 45, 65, 138).

[24] Noam Brown and Tuomas Sandholm. "Superhuman AI for multiplayer poker". In: *Science* 365.6456 (2019), pp. 885–890. eprint: https://www.science.org/doi/pdf/10.1126/science.aay2400 (cit. on p. 7).

[25] OpenAI, : Christopher Berner, et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. arXiv: 1912.06680 [cs.LG] (cit. on p. 7).

[26] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2024 (cit. on pp. 7, 10, 19).

[27] Jayesh K. Gupta, Maxim Egorov, and Mykel Kochenderfer. "Cooperative Multi-agent Control Using Deep Reinforcement Learning". In: *Autonomous Agents and Multiagent Systems*. Ed. by Gita Sukthankar and Juan A. Rodriguez-Aguilar. Cham: Springer International Publishing, 2017, pp. 66–83 (cit. on p. 8).

[28] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279 (cit. on p. 9).

[29] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, et al. *The StarCraft Multi-Agent Challenge*. 2019. arXiv: 1902.04043 [cs.LG] (cit. on pp. 9, 41).

[30] Matteo Bettini, Ryan Kortvelesy, Jan Blumenkamp, and Amanda Prorok. *VMAS: A Vectorized Multi-Agent Simulator for Collective Robot Learning*. 2022. arXiv: 2207.03530 [cs.RO] (cit. on pp. 9, 40, 116).

[31] Ryan Lowe, Yi Wu, Aviv Tamar, et al. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments". In: *Neural Information Processing Systems (NIPS)* (2017) (cit. on p. 9).

[32] Igor Mordatch and Pieter Abbeel. "Emergence of Grounded Compositional Language in Multi-Agent Populations". In: *arXiv preprint arXiv:1703.04908* (2017) (cit. on p. 9).

[33] Peter Sunehag, Guy Lever, Audrunas Gruslys, et al. *Value-Decomposition Networks For Cooperative Multi-Agent Learning*. 2017. arXiv: 1706.05296 [cs.AI] (cit. on p. 9).

[34] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder de Witt, et al. *QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning*. 2018. arXiv: 1803.11485 [cs.LG] (cit. on pp. 9, 13).

[35] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. *Counterfactual Multi-Agent Policy Gradients*. 2024. arXiv: 1705.08926 [cs.AI] (cit. on p. 9).

[36] Meng Zhou, Ziyu Liu, Pengwei Sui, Yixuan Li, and Yuk Ying Chung. *Learning Implicit Credit Assignment for Cooperative Multi-Agent Reinforcement Learning*. 2020. arXiv: 2007.02529 [cs.LG] (cit. on p. 9).

[37] Ryan Lowe, Yi Wu, Aviv Tamar, et al. *Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments*. 2020. arXiv: 1706.02275 [cs.LG] (cit. on p. 10).

[38] Zhenhui Ye, Yining Chen, Guanghua Song, Bowei Yang, and Shen Fan. *Experience Augmentation: Boosting and Accelerating Off-Policy Multi-Agent Reinforcement Learning*. 2020. arXiv: 2005.09453 [cs.LG] (cit. on p. 10).

[39] Juho Lee, Yoonho Lee, Jungtaek Kim, et al. *Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks*. 2019. arXiv: 1810.00825 [cs.LG] (cit. on pp. 10, 12, 13, 15, 24, 36, 37, 52, 72, 97).

[40] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, et al. *Relational inductive biases, deep learning, and graph networks*. 2018. arXiv: 1806.01261 [cs.LG] (cit. on p. 10).

[41] Jiawei Shen, Qurat-ul-Tajnab Ain, Yang Liu, et al. "GTAT: Empowering Graph Neural Networks with Cross Attention". In: *Scientific Reports* 15 (2025), p. 4760 (cit. on pp. 10, 13, 14, 102).

[42] Krikamol Muandet, Kenji Fukumizu, Bharath Sriperumbudur, and Bernhard Schölkopf. "Kernel Mean Embedding of Distributions: A Review and Beyond". In: *Foundations and Trends® in Machine Learning* 10.1–2 (2017), pp. 1–141 (cit. on pp. 10, 12).

[43] Bor Jiun Lin and Chun-Yi Lee. "HGAP: boosting permutation invariant and permutation equivariant in multi-agent reinforcement learning via graph attention network". In: *Proceedings of the 41st International Conference on Machine Learning*. ICML'24. Vienna, Austria: JMLR.org, 2024 (cit. on pp. 10, 23).

[44] Fengzhuo Zhang, Boyi Liu, Kaixin Wang, et al. *Relational Reasoning via Set Transformers: Provable Efficiency and Applications to MARL*. 2022. arXiv: 2209.09845 [cs.LG] (cit. on pp. 10, 12, 14).

[45] Zican Hu, Zongzhang Zhang, Huaxiong Li, et al. *Attention-Guided Contrastive Role Representations for Multi-Agent Reinforcement Learning*. 2024. arXiv: 2312.04819 [cs.MA] (cit. on p. 10).

[46] Edward Wagstaff, Fabian Fuchs, Martin Engelcke, Ingmar Posner, and Michael A. Osborne. "On the Limitations of Representing Functions on Sets". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 6487–6494 (cit. on p. 11).

[47] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. "A survey and critique of multiagent deep reinforcement learning". In: *Autonomous Agents and Multi-Agent Systems* 33.6 (Oct. 2019), pp. 750–797 (cit. on pp. 11, 15).

[48] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*. 2017. arXiv: 1612.00593 [cs.CV] (cit. on pp. 11, 25, 26).

[49] Tongyue Li, Dianxi Shi, Songchang Jin, et al. "Multi-Agent Hierarchical Graph Attention Actor–Critic Reinforcement Learning". In: *Entropy* 27.1 (2025) (cit. on p. 11).

[50] Bocheng ZHAO, Mingying HUO, Zheng LI, et al. "Graph-based multi-agent reinforcement learning for collaborative search and tracking of multiple UAVs". In: *Chinese Journal of Aeronautics* 38.3 (2025), p. 103214 (cit. on pp. 12, 13).

[51] Fernando Gama, Antonio G. Marques, Geert Leus, and Alejandro Ribeiro. "Convolutional Neural Network Architectures for Signals Supported on Graphs". In: *IEEE Transactions on Signal Processing* 67.4 (Feb. 2019), pp. 1034–1049 (cit. on p. 12).

[52] Iou-Jen Liu, Raymond A. Yeh, and Alexander G. Schwing. *PIC: Permutation Invariant Critic for Multi-Agent Deep Reinforcement Learning*. 2019. arXiv: 1911.00025 [cs.LG] (cit. on p. 12).

[53] Yifan Hu, Junjie Fu, and Guanghui Wen. "Graph Soft Actor–Critic Reinforcement Learning for Large-Scale Distributed Multirobot Coordination". In: *IEEE Transactions on Neural Networks and Learning Systems* 36.1 (2025), pp. 665–676 (cit. on p. 12).

[54] Lavanya Ratnabala, Robinroy Peter, Aleksey Fedoseev, and Dzmitry Tsetserukou. *HIPPO-MAT: Decentralized Task Allocation Using GraphSAGE and Multi-Agent Deep Reinforcement Learning*. 2025. arXiv: 2503.07662 [cs.MA] (cit. on p. 12).

[55] Shaked Brody, Uri Alon, and Eran Yahav. *How Attentive are Graph Attention Networks?* 2022. arXiv: 2105.14491 [cs.LG] (cit. on pp. 13, 35, 36, 47, 49).

[56] Ziheng Liu, Jiayi Zhang, Enyu Shi, et al. *Graph Neural Network Meets Multi-Agent Reinforcement Learning: Fundamentals, Applications, and Future Directions*. 2024. arXiv: 2404.04898 [cs.IT] (cit. on p. 13).

[57] Chao Lu, Qiang Bao, Shiqi Xia, et al. "Centralized reinforcement learning for multi-agent cooperative environments". In: *Evolutionary Intelligence* 17.1 (Feb. 2024), pp. 267–273 (cit. on pp. 13, 15, 22).

[58] Shariq Iqbal and Fei Sha. *Actor-Attention-Critic for Multi-Agent Reinforcement Learning*. 2019. arXiv: 1810.02912 [cs.LG] (cit. on pp. 13, 15).

[60] Frans A Oliehoek, Christopher Amato, et al. *A concise introduction to decentralized POMDPs*. Vol. 1. Springer, 2016 (cit. on pp. 17–19).

[61] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023 (cit. on p. 17).

[62] David Silver. *Lectures on Reinforcement Learning*. URL: https://www.davidsilver.uk/teaching/. 2015 (cit. on pp. 17, 18).

[63] Daniel S. Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. "The Complexity of Decentralized Control of Markov Decision Processes". In: *Mathematics of Operations Research* 27.4 (2002), pp. 819–840 (cit. on p. 19).

[64] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG] (cit. on pp. 19, 20, 22, 61).

[65] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG] (cit. on p. 19).

[66] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. arXiv: 1506.02438 [cs.LG] (cit. on p. 20).

[67] Zafarali Ahmed, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. *Understanding the impact of entropy on policy optimization*. 2019. arXiv: 1811.11214 [cs.LG] (cit. on p. 21).

[68] Shivam Kalra, Mohammed Adnan, Graham Taylor, and H. R. Tizhoosh. "Learning Permutation Invariant Representations Using Memory Networks". In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm. Cham: Springer International Publishing, 2020, pp. 677–693 (cit. on pp. 23, 24).

[69] Hossein Gholamalinezhad and Hossein Khosravi. *Pooling Methods in Deep Neural Networks, a Review*. 2020. arXiv: 2009.07485 [cs.CV] (cit. on p. 25).

[70] Xiaomeng Wu, Go Irie, Kaoru Hiramatsu, and Kunio Kashino. "Weighted Generalized Mean Pooling for Deep Image Retrieval". In: *2018 25th IEEE International Conference on Image Processing (ICIP)*. 2018, pp. 495–499 (cit. on p. 25).

[71] Charu C Aggarwal and Charu C Aggarwal. "An introduction to neural networks". In: *Neural networks and deep learning: A textbook* (2018) (cit. on pp. 26, 27).

[72] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536 (cit. on p. 28).

[73] Léon Bottou. “Large-Scale Machine Learning with Stochastic Gradient Descent”. In: *Proceedings of COMPSTAT’2010*. Ed. by Yves Lechevallier and Gilbert Saporta. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186 (cit. on p. 28).

[74] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. 2021. arXiv: 2104.13478 [cs.LG] (cit. on p. 28).

[75] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. “Geometric Deep Learning: Going beyond Euclidean data”. In: *IEEE Signal Processing Magazine* 34.4 (July 2017), pp. 18–42 (cit. on p. 28).

[76] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG] (cit. on p. 29).

[77] Google DeepMind. *Gemini*. https://deepmind.google/technologies/gemini. Available at https://deepmind.google/technologies/gemini. 2023 (cit. on p. 29).

[78] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. https://D2L.ai. Cambridge University Press, 2023 (cit. on p. 30).

[79] Grant Sanderson. *Visualizing Attention, a Transformer’s Heart | Chapter 6, Deep Learning*. 3Blue1Brown. 2024 (cit. on p. 30).

[80] Leonid Berlyand and Pierre-Emmanuel Jabin. *An Introduction*. Berlin, Boston: De Gruyter, 2023 (cit. on p. 31).

[81] Lorenzo Canese, Gian Carlo Cardarilli, Luca Di Nunzio, et al. “Multi-Agent Reinforcement Learning: A Review of Challenges and Applications”. In: *Applied Sciences* 11.11 (2021) (cit. on pp. 31, 45).

[82] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. “DeepWalk: online learning of social representations”. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD ’14. ACM, Aug. 2014, pp. 701–710 (cit. on p. 33).

[83] Matteo Bettini. *Multi-Agent Reinforcement Learning (PPO) with TorchRL Tutorial*. https://docs.pytorch.org/rl/stable/tutorials/multiagent_ppo.html. Accessed: 2025-05-24. 2025 (cit. on pp. 39, 40, 61, 62).

[84] Matthias Fey and Jan E. Lenssen. “Fast Graph Representation Learning with PyTorch Geometric”. In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019 (cit. on pp. 47–49).

[85] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. “Power-Law Distributions in Empirical Data”. In: *SIAM Review* 51.4 (2009), pp. 661–703. eprint: https://doi.org/10.1137/070710111 (cit. on p. 57).

[86] R. Horan and M. Lavelle. *Log-Log Plots*. Basic Mathematics module, PPLATO project, University of Plymouth. PDF self-assessment resource for understanding log-log plots. Last revision date: March 11, 2004. Version 1.0. 2004 (cit. on p. 57).

[87] Jared S. Murray. *Exponential growth, power laws, and the log transformation*. `https://jaredsmurray.github.io/sta371h/files/15_exponential_power_laws.pdf`. Lecture slides, University of Texas at Austin (cit. on p. 57).

Appendices

Hyperparameter Tuning Experiments

A

A.1 Scenario parameters

PPO Settings	
Learning Rate	3×10^{-4}
Max Gradient Norm	1.0
Clipping Epsilon	0.2
Discount Factor (γ)	0.99
GAE Lambda (λ)	0.9
Entropy Reg. Coefficient	1×10^{-4}

Training Settings	
Decentralized Execution	True
Frames per Batch	6000
Number of Iterations	80
Number of Epochs	8
Minibatch Size	400

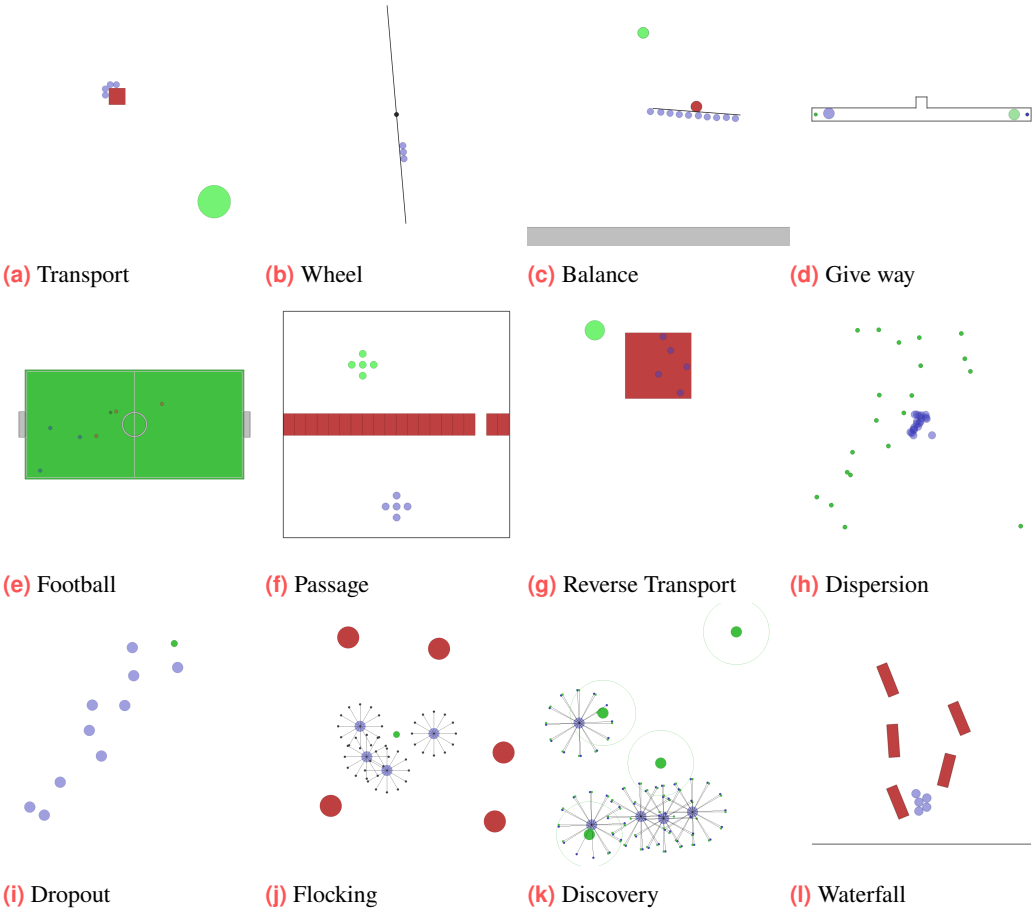
Tab. A.1.: Default PPO and training settings. Regarding the training settings, *Frames per Batch* specifies the number of environment-agent steps collected before each policy update, while *Minibatch Size* defines how many of these steps are used for each gradient update. The collected batch is split into minibatches and then iterated over for *Number of Epochs*, so the optimizer performs multiple updates per batch.

Balance scenario	
Positional shaping reward	100
Fall penalty	-10
Agent radius	0.03
Line length	0.8
Number of Agents	5-40
Maximum Steps	100

Navigation scenario	
Positional shaping reward	1
Final reward	0.01
Agent collision penalty	-1
Agent radius	0.1
Lidar range	0.35
Number of Agents	4-20
Maximum Steps	200

Multi-Give-Way scenario	
Positional shaping reward	4
Final reward	10
Agent collision penalty	-0.1
Passage collision penalty	-0.1
Agent radius	0.1
Number of Agents	4
Maximum Steps	200

Tab. A.2.: Default environment and reward settings.



(a) Transport (b) Wheel (c) Balance (d) Give way

(e) Football (f) Passage (g) Reverse Transport (h) Dispersion

(i) Dropout (j) Flocking (k) Discovery (l) Waterfall

Fig. A.1.: Pre-implemented scenarios by VMAS[30]. The agents are the blue shapes and they interact with landmarks to solve the given task.

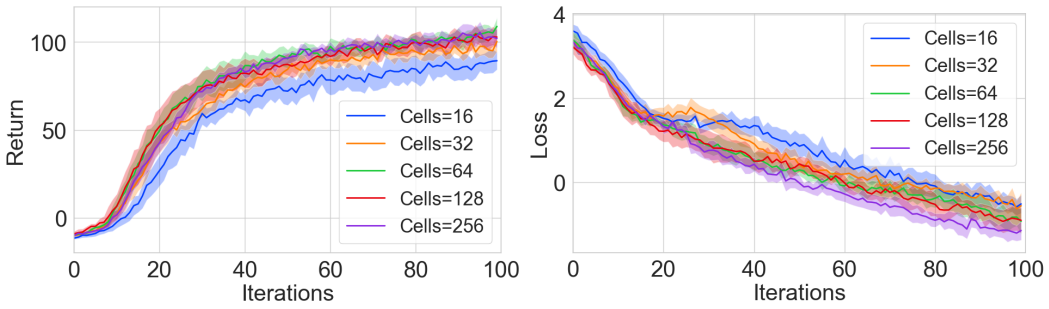
A.2 *CONCAT* baseline

Cells	Training Time	Memory Usage	Inference Time	Mean Rewards	Loss	FLOPs	Parameters
16	42.32 ± 1.29	600.67	$4.26 \cdot 10^{-4} \pm 1.75 \cdot 10^{-5}$	89.36 ± 4.93	−0.52 ± 0.21	6.34	3.16
32	50.34 ± 0.41	838.57	$4.37 \cdot 10^{-4} \pm 4.30 \cdot 10^{-6}$	100.27 ± 5.63	−0.56 ± 0.31	14.72	7.36
64	59.04 ± 0.40	1321.23	$5.04 \cdot 10^{-4} \pm 3.49 \cdot 10^{-5}$	108.91 ± 4.82	−0.93 ± 0.31	37.64	18.82
128	81.43 ± 0.68	2301.85	$5.18 \cdot 10^{-4} \pm 1.14 \cdot 10^{-5}$	102.84 ± 4.76	−0.91 ± 0.30	108.04	54.02
256	143.17 ± 0.50	4330.49	$6.32 \cdot 10^{-4} \pm 1.01 \cdot 10^{-5}$	102.01 ± 5.33	−1.15 ± 0.22	347.14	173.56

Tab. A.3.: Resulting metrics of changing the number of cells for *CONCAT*. Depth set to 2.

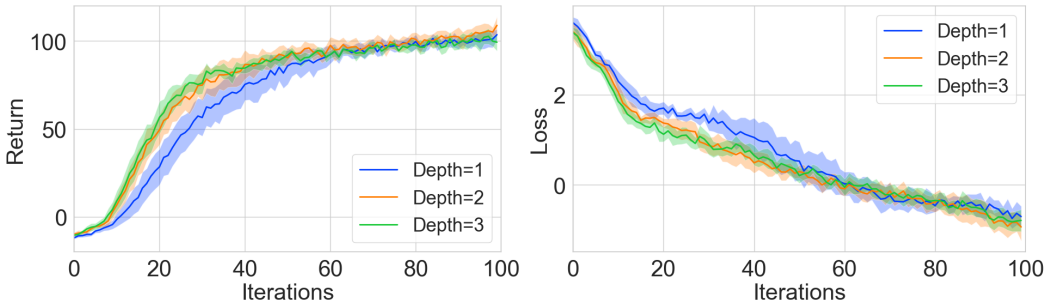
Depth	Training Time	Memory Usage	Inference Time	Mean Rewards	Loss	FLOPs	Parameters
1	46.43 ± 4.02	841.15	$4.04 \cdot 10^{-4} \pm 2.04 \cdot 10^{-5}$	103.63 ± 2.96	−0.70 ± 0.23	21.00	10.50
2	59.57 ± 0.95	1322.31	$4.96 \cdot 10^{-4} \pm 1.80 \cdot 10^{-5}$	108.91 ± 4.82	−0.93 ± 0.31	37.64	18.82
3	79.43 ± 2.11	1801.92	$6.07 \cdot 10^{-4} \pm 2.68 \cdot 10^{-5}$	99.35 ± 4.94	−0.79 ± 0.24	54.28	27.14

Tab. A.4.: Resulting metrics of changing the depth for *CONCAT*. Layer size set to 64.



(a) Impact on *CONCAT* average reward of tuning the number of cells, with depth 2.

(b) Impact on *CONCAT* loss of tuning the number of cells, with depth 2.



(c) Impact on *CONCAT* average reward of tuning the depth of the core mlp with 64 cells.

(d) Impact on *CONCAT* loss of tuning the depth of the core mlp with 64 cells.

Fig. A.2.: Tuning of number of cells and depth of the MLP for *CONCAT*. Based on these results, *mlp_core_num_cells*= 64 and *depth*= 2 were chosen to balance complexity and performance.

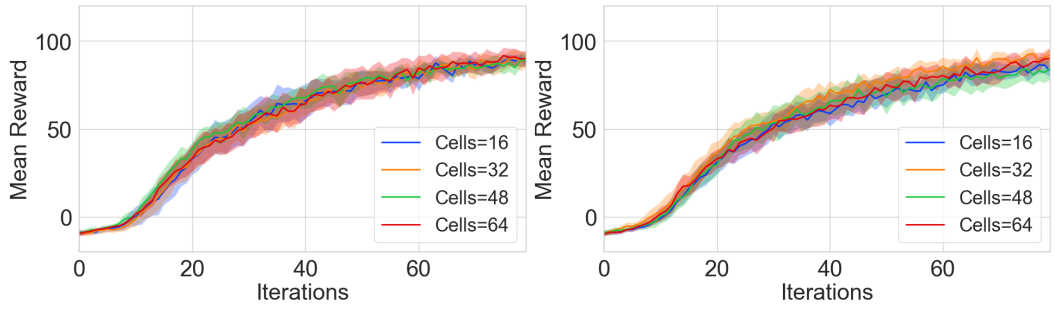
A.3 DeepSet Methods

Strategy	Depth	Cells	Training Time	Memory	Inference Time	Reward	Loss	Flops	Parameter
DS	None	None	102.48 ± 3.82	4252.44	$6.05 \cdot 10^{-4} \pm 2.06 \cdot 10^{-5}$	98.95 ± 3.49	−0.63 ± 0.21	281.60	140.80
	1	16	113.81 ± 1.16	4358.32	$6.28 \cdot 10^{-4} \pm 2.11 \cdot 10^{-5}$	88.25 ± 3.81	0.09 ± 0.24	283.78	141.88
		32	116.49 ± 1.09	4503.18	$5.90 \cdot 10^{-4} \pm 1.44 \cdot 10^{-5}$	91.68 ± 4.54	−0.05 ± 0.25	304.38	152.20
		48	118.05 ± 0.21	4648.48	$6.07 \cdot 10^{-4} \pm 1.65 \cdot 10^{-5}$	95.04 ± 4.36	−0.20 ± 0.35	327.04	163.52
		64	120.47 ± 0.60	4794.82	$6.24 \cdot 10^{-4} \pm 1.72 \cdot 10^{-5}$	90.72 ± 3.47	−0.11 ± 0.23	351.74	175.88
	2	16	118.50 ± 0.98	4417.38	$6.64 \cdot 10^{-4} \pm 3.28 \cdot 10^{-5}$	94.00 ± 3.32	−0.14 ± 0.25	284.86	142.44
		32	119.29 ± 0.91	4621.72	$6.14 \cdot 10^{-4} \pm 2.37 \cdot 10^{-5}$	92.09 ± 3.12	−0.02 ± 0.19	308.62	154.30
		48	122.19 ± 0.59	4827.44	$6.03 \cdot 10^{-4} \pm 1.71 \cdot 10^{-5}$	90.29 ± 4.73	0.0061 ± 0.22	336.46	168.22
		64	122.60 ± 0.54	5034.88	$6.04 \cdot 10^{-4} \pm 7.46 \cdot 10^{-6}$	89.55 ± 10.24	−0.07 ± 0.35	368.38	184.20
DS_LOCAL	None	None	105.61 ± 1.82	4372.17	$6.05 \cdot 10^{-4} \pm 1.24 \cdot 10^{-5}$	95.97 ± 3.74	−0.42 ± 0.19	297.98	149.00
	1	16	120.13 ± 1.51	4886.61	$6.19 \cdot 10^{-4} \pm 2.33 \cdot 10^{-5}$	95.47 ± 5.04	−0.11 ± 0.33	300.16	150.08
		32	127.34 ± 1.39	5526.08	$6.27 \cdot 10^{-4} \pm 2.63 \cdot 10^{-5}$	92.74 ± 3.92	−0.20 ± 0.23	320.78	160.38
		48	129.37 ± 0.98	6166.56	$6.01 \cdot 10^{-4} \pm 1.48 \cdot 10^{-5}$	90.89 ± 3.61	−0.21 ± 0.29	343.42	171.72
		64	131.29 ± 1.14	6810.60	$6.35 \cdot 10^{-4} \pm 3.01 \cdot 10^{-5}$	90.62 ± 4.56	−0.34 ± 0.22	368.14	184.06
	2	16	126.49 ± 2.35	4946.09	$6.29 \cdot 10^{-4} \pm 2.89 \cdot 10^{-5}$	95.73 ± 4.20	−0.35 ± 0.36	301.26	150.62
		32	131.06 ± 2.22	5646.94	$6.49 \cdot 10^{-4} \pm 2.23 \cdot 10^{-5}$	93.55 ± 4.91	−0.44 ± 0.32	325.00	162.50
		48	134.37 ± 0.64	6351.44	$6.67 \cdot 10^{-4} \pm 1.36 \cdot 10^{-5}$	89.77 ± 5.42	0.01 ± 0.22	352.84	176.42
		64	140.23 ± 1.94	7059.99	$6.64 \cdot 10^{-4} \pm 3.98 \cdot 10^{-5}$	93.16 ± 3.63	−0.21 ± 0.33	384.78	192.38
DS_GLOBAL	None	None	100.17 ± 3.01	4272.57	$6.21 \cdot 10^{-4} \pm 1.21 \cdot 10^{-5}$	94.23 ± 3.54	−0.32 ± 0.23	297.98	149.00
	1	16	111.92 ± 1.78	4393.49	$6.07 \cdot 10^{-4} \pm 1.64 \cdot 10^{-5}$	94.67 ± 2.78	−0.45 ± 0.26	300.16	150.08
		32	112.86 ± 0.71	4538.40	$6.37 \cdot 10^{-4} \pm 2.66 \cdot 10^{-5}$	93.49 ± 3.01	−0.17 ± 0.29	320.78	160.38
		48	111.55 ± 1.09	4683.60	$6.43 \cdot 10^{-4} \pm 3.66 \cdot 10^{-5}$	91.15 ± 3.17	−0.16 ± 0.22	343.42	171.72
		64	115.36 ± 0.41	4830.03	$6.06 \cdot 10^{-4} \pm 1.33 \cdot 10^{-5}$	89.29 ± 5.72	−0.17 ± 0.28	368.14	184.06
	2	16	112.05 ± 2.16	4452.33	$6.18 \cdot 10^{-4} \pm 5.90 \cdot 10^{-5}$	97.78 ± 5.33	−0.55 ± 0.31	301.26	150.62
		32	115.41 ± 0.51	4656.78	$6.12 \cdot 10^{-4} \pm 2.15 \cdot 10^{-5}$	95.72 ± 2.77	−0.30 ± 0.22	325.00	162.50
		48	117.66 ± 0.65	4862.74	$6.13 \cdot 10^{-4} \pm 1.26 \cdot 10^{-5}$	90.45 ± 5.34	−0.31 ± 0.12	352.84	176.42
		64	118.49 ± 0.61	5070.04	$6.16 \cdot 10^{-4} \pm 1.33 \cdot 10^{-5}$	91.72 ± 3.85	−0.34 ± 0.29	384.78	192.38

Tab. A.5.: Encoder width and depth experiment for DeepSet Approaches. All with 256 MLP core with depth 2.

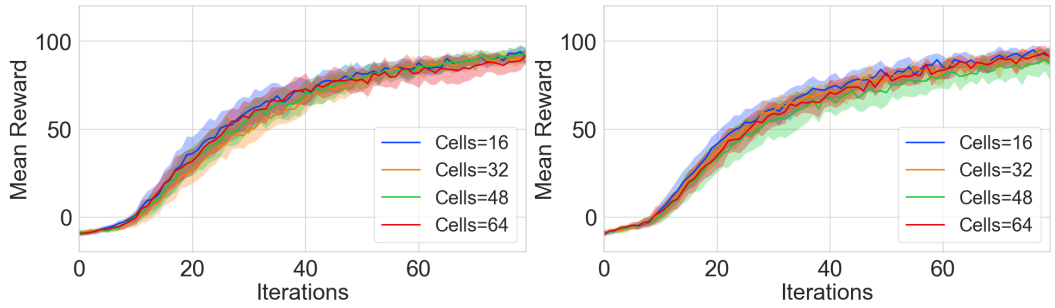
Strategy	Depth	Cells	Training Time	Memory	Inference Time	Reward	Loss	Flops	Parameter
DS	None	None	61.94 ± 0.26	2234.58	$5.52 \cdot 10^{-4} \pm 7.79 \cdot 10^{-6}$	90.92 ± 2.52	-0.11 ± 0.15	75.26	37.64
	1	16	64.45 ± 0.48	2341.30	$5.13 \cdot 10^{-4} \pm 1.53 \cdot 10^{-5}$	88.13 ± 2.32	0.04 ± 0.26	77.44	38.72
		32	71.74 ± 1.08	2483.28	$5.73 \cdot 10^{-4} \pm 3.42 \cdot 10^{-5}$	89.02 ± 4.41	-0.15 ± 0.25	89.86	44.92
		48	72.03 ± 0.37	2625.97	$6.14 \cdot 10^{-4} \pm 3.77 \cdot 10^{-5}$	89.38 ± 3.55	-0.22 ± 0.26	104.32	52.16
		64	74.20 ± 0.33	2769.17	$5.91 \cdot 10^{-4} \pm 1.09 \cdot 10^{-5}$	90.00 ± 4.47	-0.06 ± 0.26	120.84	60.42
	2	16	67.62 ± 0.54	2400.54	$5.24 \cdot 10^{-4} \pm 2.63 \cdot 10^{-5}$	82.97 ± 3.43	0.28 ± 0.28	78.54	39.26
		32	72.48 ± 0.34	2600.49	$5.23 \cdot 10^{-4} \pm 1.71 \cdot 10^{-5}$	91.91 ± 4.15	-0.19 ± 0.25	94.08	47.04
		48	74.26 ± 0.82	2805.04	$5.48 \cdot 10^{-4} \pm 3.58 \cdot 10^{-5}$	84.45 ± 6.31	0.09 ± 0.35	113.74	56.86
		64	75.27 ± 0.26	3008.82	$5.08 \cdot 10^{-4} \pm 7.65 \cdot 10^{-6}$	90.07 ± 6.08	0.0065 ± 0.31	137.48	68.74
DS_LOCAL	None	None	59.21 ± 0.70	2351.98	$5.01 \cdot 10^{-4} \pm 8.16 \cdot 10^{-6}$	96.51 ± 3.57	-0.40 ± 0.17	83.46	41.72
	1	16	71.64 ± 0.48	2866.50	$5.10 \cdot 10^{-4} \pm 1.61 \cdot 10^{-5}$	90.86 ± 4.96	-0.15 ± 0.24	85.64	42.82
		32	74.95 ± 0.71	3502.86	$5.13 \cdot 10^{-4} \pm 1.14 \cdot 10^{-5}$	90.44 ± 4.23	-0.12 ± 0.16	98.06	49.02
		48	77.99 ± 0.49	4141.49	$5.20 \cdot 10^{-4} \pm 1.73 \cdot 10^{-5}$	92.23 ± 4.00	-0.13 ± 0.30	112.52	56.26
		64	82.67 ± 0.78	4781.88	$5.33 \cdot 10^{-4} \pm 1.93 \cdot 10^{-5}$	91.86 ± 3.86	-0.12 ± 0.28	129.02	64.52
	2	16	74.17 ± 0.36	2925.68	$4.98 \cdot 10^{-4} \pm 7.55 \cdot 10^{-6}$	91.30 ± 5.98	-0.32 ± 0.16	86.72	43.36
		32	78.73 ± 0.74	3623.32	$5.20 \cdot 10^{-4} \pm 4.06 \cdot 10^{-5}$	90.44 ± 3.94	0.09 ± 0.27	102.28	51.14
		48	84.02 ± 0.61	4325.60	$5.05 \cdot 10^{-4} \pm 9.07 \cdot 10^{-6}$	87.10 ± 9.58	-0.12 ± 0.30	121.92	60.96
		64	86.78 ± 0.58	5031.43	$5.15 \cdot 10^{-4} \pm 1.71 \cdot 10^{-5}$	90.85 ± 4.46	-0.11 ± 0.22	145.66	72.84
DS_GLOBAL	None	None	55.81 ± 0.31	2252.23	$5.12 \cdot 10^{-4} \pm 1.28 \cdot 10^{-5}$	95.30 ± 2.96	-0.27 ± 0.15	83.46	41.72
	1	16	61.31 ± 0.33	2373.03	$4.95 \cdot 10^{-4} \pm 6.23 \cdot 10^{-6}$	91.74 ± 3.72	-0.13 ± 0.18	85.64	42.82
		32	62.66 ± 0.45	2514.98	$5.03 \cdot 10^{-4} \pm 8.63 \cdot 10^{-6}$	92.32 ± 4.87	-0.02 ± 0.28	98.06	49.02
		48	63.60 ± 0.46	2657.88	$5.32 \cdot 10^{-4} \pm 6.21 \cdot 10^{-5}$	93.17 ± 4.89	-0.12 ± 0.24	112.52	56.26
		64	64.56 ± 0.25	2800.85	$5.20 \cdot 10^{-4} \pm 2.76 \cdot 10^{-5}$	92.90 ± 6.41	-0.22 ± 0.27	129.02	64.52
	2	16	63.00 ± 0.91	2431.69	$5.03 \cdot 10^{-4} \pm 9.13 \cdot 10^{-6}$	90.80 ± 3.53	0.0021 ± 0.32	86.72	43.36
		32	65.99 ± 1.35	2633.50	$5.49 \cdot 10^{-4} \pm 1.88 \cdot 10^{-5}$	93.89 ± 4.24	-0.11 ± 0.22	102.28	51.14
		48	66.22 ± 1.25	2835.38	$5.34 \cdot 10^{-4} \pm 7.28 \cdot 10^{-6}$	91.91 ± 3.77	-0.11 ± 0.24	121.92	60.96
		64	69.42 ± 0.60	3040.98	$5.37 \cdot 10^{-4} \pm 5.39 \cdot 10^{-6}$	90.67 ± 3.50	-0.19 ± 0.30	145.66	72.84

Tab. A.6.: Encoder width and depth experiment for DeepSet Approaches. All with 128 MLP core with depth 2.



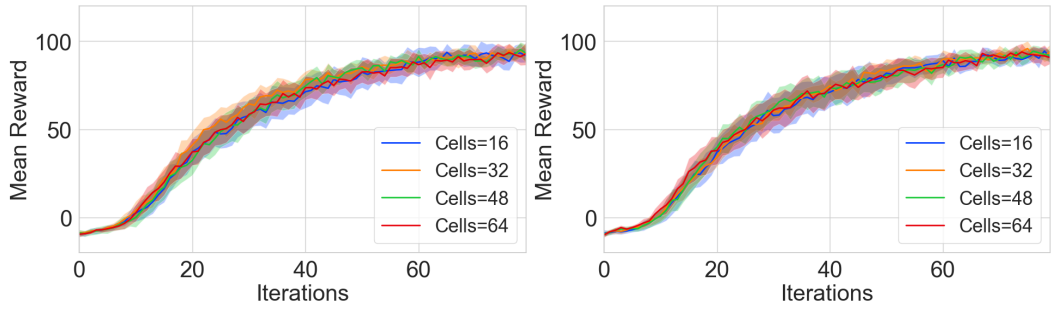
(a) DS depth 1

(b) DS depth 2



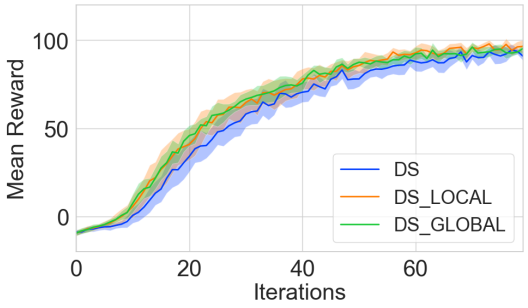
(c) DS LOCAL depth 1

(d) DS LOCAL depth 2



(e) DS GLOBAL depth 1

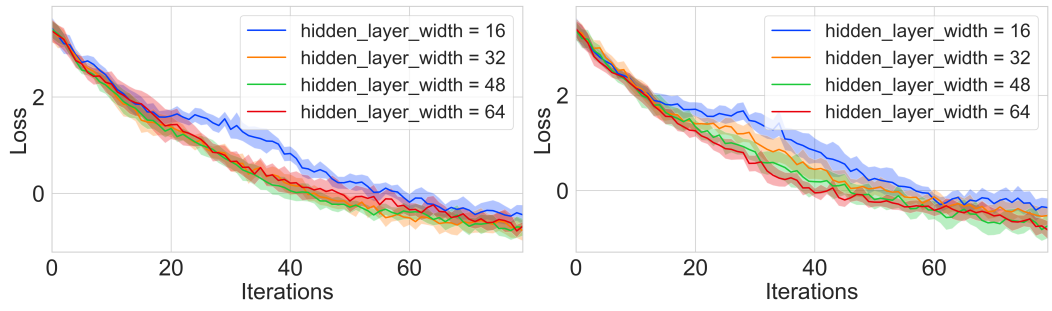
(f) DS GLOBAL depth 2



(g) No depth variant

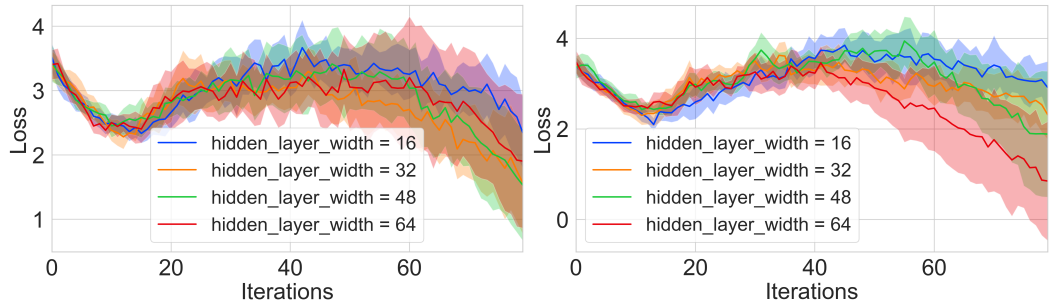
Fig. A.3.: Plots of the mean rewards achieved by varying depth of the encoding network, with number of core cells 128.

A.4 GNN Methods



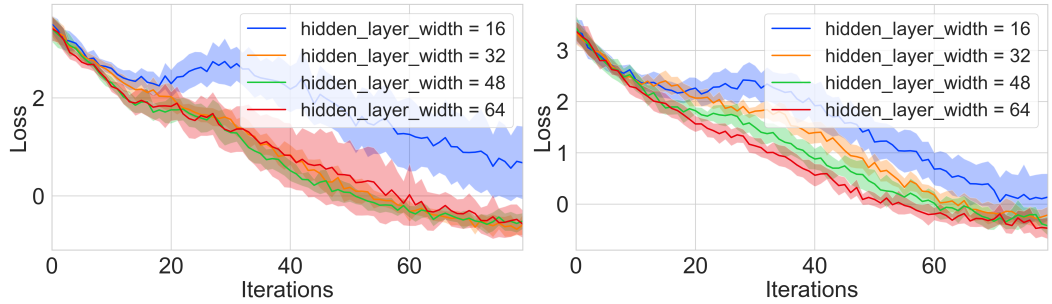
(a) GSAGE with encode=True

(b) GSAGE with encode=False



(c) GAT with encode=True

(d) GAT with encode=False



(e) GATv2 with encode=True

(f) GATv2 with encode=True

Fig. A.4.: Loss curves of GNN embedding techniques with varying embedding width and encoder network either set to True, i.e. a linear layer followed by an activation followed by an output layer, or set to false, i.e. just one linear layer, without any activation layers. Note, that the limits of the y-axis is quite different across the figures because of the unstable loss of the *GAT* models.

Method	MLP	Width	Training Time	Memory Usage	Inference Time	Mean Rewards	Loss
GraphSAGE	True	16	42.16 ± 1.10	1231.60	$4.42 \cdot 10^{-4} \pm 2.51 \cdot 10^{-5}$	93.52 ± 2.69	−0.45 ± 0.20
		32	49.49 ± 0.41	1690.41	$4.64 \cdot 10^{-4} \pm 1.08 \cdot 10^{-5}$	97.98 ± 2.74	−0.77 ± 0.21
		48	54.65 ± 0.32	2152.33	$5.46 \cdot 10^{-4} \pm 1.66 \cdot 10^{-4}$	94.74 ± 3.73	−0.70 ± 0.18
		64	57.23 ± 0.35	2616.74	$4.53 \cdot 10^{-4} \pm 5.89 \cdot 10^{-6}$	93.96 ± 2.64	−0.68 ± 0.09
GraphSAGE	False	16	39.98 ± 0.31	1172.33	$4.31 \cdot 10^{-4} \pm 6.06 \cdot 10^{-6}$	91.88 ± 4.45	−0.36 ± 0.19
		32	45.18 ± 0.66	1571.86	$4.43 \cdot 10^{-4} \pm 1.14 \cdot 10^{-5}$	93.87 ± 3.37	−0.52 ± 0.19
		48	51.11 ± 0.22	1973.33	$4.51 \cdot 10^{-4} \pm 1.42 \cdot 10^{-5}$	92.16 ± 4.81	−0.74 ± 0.26
		64	53.02 ± 0.32	2376.16	$4.50 \cdot 10^{-4} \pm 1.08 \cdot 10^{-5}$	96.78 ± 2.88	−0.83 ± 0.19
GAT	True	16	59.78 ± 0.63	2235.11	$4.60 \cdot 10^{-4} \pm 2.14 \cdot 10^{-5}$	72.59 ± 8.09	2.35 ± 0.55
		32	73.27 ± 0.67	3505.82	$4.36 \cdot 10^{-4} \pm 3.56 \cdot 10^{-6}$	70.28 ± 8.26	1.57 ± 0.79
		48	88.89 ± 0.73	4761.42	$4.69 \cdot 10^{-4} \pm 1.92 \cdot 10^{-5}$	75.87 ± 6.60	1.53 ± 0.86
		64	102.34 ± 1.12	6039.32	$4.65 \cdot 10^{-4} \pm 1.76 \cdot 10^{-5}$	77.78 ± 8.99	1.90 ± 1.04
GAT	False	16	57.76 ± 0.54	2176.51	$4.34 \cdot 10^{-4} \pm 5.82 \cdot 10^{-6}$	60.97 ± 7.00	2.91 ± 0.58
		32	69.84 ± 0.49	3379.75	$4.52 \cdot 10^{-4} \pm 1.23 \cdot 10^{-5}$	63.75 ± 8.98	2.32 ± 0.54
		48	85.26 ± 0.52	4578.94	$4.55 \cdot 10^{-4} \pm 1.51 \cdot 10^{-5}$	68.79 ± 8.81	1.89 ± 1.39
		64	96.42 ± 0.64	5798.77	$4.45 \cdot 10^{-4} \pm 7.47 \cdot 10^{-6}$	65.94 ± 10.07	0.85 ± 1.32
GATv2	True	16	71.27 ± 1.47	3288.34	$4.49 \cdot 10^{-4} \pm 1.72 \cdot 10^{-5}$	81.08 ± 9.40	0.68 ± 0.74
		32	93.34 ± 0.60	5639.57	$4.52 \cdot 10^{-4} \pm 1.14 \cdot 10^{-5}$	94.78 ± 7.37	−0.59 ± 0.20
		48	123.13 ± 0.72	8026.99	$4.76 \cdot 10^{-4} \pm 1.87 \cdot 10^{-5}$	96.08 ± 3.22	−0.51 ± 0.16
		64	149.04 ± 0.71	10388.47	$4.84 \cdot 10^{-4} \pm 1.68 \cdot 10^{-5}$	95.50 ± 4.38	−0.57 ± 0.28
GATv2	False	16	67.64 ± 0.68	3234.95	$4.44 \cdot 10^{-4} \pm 2.15 \cdot 10^{-5}$	85.90 ± 9.05	0.14 ± 0.46
		32	87.88 ± 0.85	5531.84	$4.44 \cdot 10^{-4} \pm 6.70 \cdot 10^{-6}$	91.20 ± 2.49	−0.21 ± 0.16
		48	116.31 ± 0.50	7840.69	$4.68 \cdot 10^{-4} \pm 1.56 \cdot 10^{-5}$	92.73 ± 4.41	−0.42 ± 0.19
		64	141.41 ± 0.88	10148.89	$5.05 \cdot 10^{-4} \pm 6.05 \cdot 10^{-5}$	92.25 ± 3.71	−0.47 ± 0.21

Tab. A.7.: Comprehensive comparison of GNN models with and without MLP encoders across varying hidden layer widths. Includes training time, memory usage, inference time, mean rewards, and loss.

Strategy	Heads	Training Time		Memory Usage	
		Mean	Std	Mean	Std
GAT	1	66.82	3.11	3280.85	456.28
	2	91.13	7.25	5330.22	745.25
	3	120.18	12.89	7375.23	1031.65
GATv2	1	79.83	1.62	4577.62	90.67
	2	116.54	1.53	7948.50	93.91
	3	158.78	1.34	11327.37	91.54

Tab. A.8.: Training times and memory usage of *GAT* and *GATv2* for varying numbers of attention heads in the encoder network, with layer width chosen from best performing model in Table A.7.

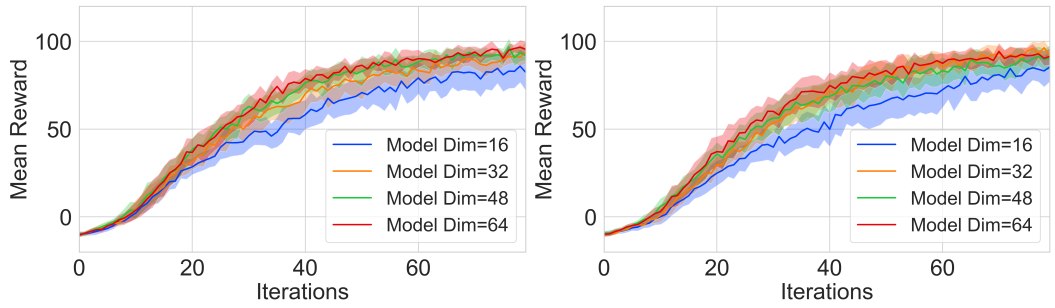
A.5 Transformer Methods

Strategy	Model Dim.	Inference Time	Loss	FLOPs	Parameters
ISAB Transformer	16	$4.56 \cdot 10^{-4} \pm 1.73 \cdot 10^{-5}$	-0.20 ± 0.26	18.02	9.00
	32	$4.74 \cdot 10^{-4} \pm 6.57 \cdot 10^{-6}$	-0.36 ± 0.19	59.38	31.30
	48	$4.69 \cdot 10^{-4} \pm 1.07 \cdot 10^{-6}$	-0.50 ± 0.22	124.08	66.92
	64	$4.72 \cdot 10^{-4} \pm 3.29 \cdot 10^{-6}$	-0.99 ± 0.14	212.12	115.84
SAB Transformer	16	$4.60 \cdot 10^{-4} \pm 1.16 \cdot 10^{-6}$	0.04 ± 0.29	10.18	4.38
	32	$4.57 \cdot 10^{-4} \pm 1.83 \cdot 10^{-6}$	-0.50 ± 0.18	30.60	13.90
	48	$4.58 \cdot 10^{-4} \pm 1.31 \cdot 10^{-6}$	-0.54 ± 0.11	61.26	28.52
	64	$4.64 \cdot 10^{-4} \pm 2.95 \cdot 10^{-6}$	-0.54 ± 0.07	102.14	48.26
SetTransformer	16	$4.65 \cdot 10^{-4} \pm 2.09 \cdot 10^{-6}$	-0.30 ± 0.25	32.92	17.08
	32	$4.66 \cdot 10^{-4} \pm 2.44 \cdot 10^{-6}$	-0.63 ± 0.14	110.50	59.78
	48	$4.64 \cdot 10^{-4} \pm 2.33 \cdot 10^{-6}$	-0.68 ± 0.13	232.70	128.06
	64	$4.66 \cdot 10^{-4} \pm 2.17 \cdot 10^{-6}$	-0.97 ± 0.15	399.56	221.96
SetTransformerOG	16	$5.01 \cdot 10^{-4} \pm 3.61 \cdot 10^{-5}$	-0.33 ± 0.32	33.18	15.68
	32	$5.49 \cdot 10^{-4} \pm 4.59 \cdot 10^{-5}$	-0.36 ± 0.30	114.28	56.96
	48	$5.45 \cdot 10^{-4} \pm 3.83 \cdot 10^{-5}$	-1.01 ± 0.15	243.30	123.84
	64	$5.27 \cdot 10^{-4} \pm 4.74 \cdot 10^{-5}$	-0.97 ± 0.19	420.26	216.32

Tab. A.9.: Inference time, loss, FLOPs, and parameter counts for varying model dimensions across transformer-based strategies.

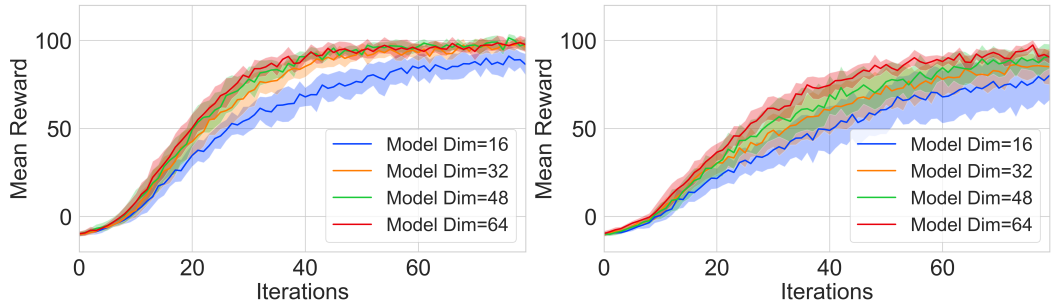
Strategy	Heads	Inference Time	Loss	FLOPs	Parameters	Mean Reward	Training Time
ISAB Transformer	1	$4.44 \cdot 10^{-4} \pm 1.29 \cdot 10^{-6}$	-0.84 ± 0.18	212.12	115.84	102.75 ± 3.15	248.68 ± 0.35
	2	$4.49 \cdot 10^{-4} \pm 1.62 \cdot 10^{-5}$	-0.99 ± 0.14	212.12	115.84	102.84 ± 4.79	382.71 ± 0.52
	4	$4.44 \cdot 10^{-4} \pm 1.16 \cdot 10^{-6}$	-0.86 ± 0.19	212.12	115.84	101.54 ± 4.80	134.11 ± 0.15
SAB Transformer	1	$4.31 \cdot 10^{-4} \pm 1.34 \cdot 10^{-5}$	-0.44 ± 0.14	102.14	48.26	104.71 ± 2.74	139.74 ± 3.50
	2	$4.43 \cdot 10^{-4} \pm 1.64 \cdot 10^{-6}$	-0.54 ± 0.07	102.14	48.26	100.67 ± 4.88	214.70 ± 0.58
	4	$4.43 \cdot 10^{-4} \pm 1.40 \cdot 10^{-6}$	-0.67 ± 0.14	102.14	48.26	103.51 ± 4.01	351.28 ± 0.96
SetTransformer	1	$4.44 \cdot 10^{-4} \pm 1.37 \cdot 10^{-6}$	-0.87 ± 0.18	232.70	128.06	100.72 ± 2.14	321.62 ± 0.33
	2	$4.40 \cdot 10^{-4} \pm 1.21 \cdot 10^{-6}$	-0.68 ± 0.13	232.70	128.06	100.19 ± 3.56	457.49 ± 0.38
	4	$4.44 \cdot 10^{-4} \pm 1.27 \cdot 10^{-6}$	-0.91 ± 0.17	232.70	128.06	99.44 ± 3.49	230.48 ± 0.10
SetTransformerOG	1	$4.42 \cdot 10^{-4} \pm 7.62 \cdot 10^{-7}$	-0.76 ± 0.15	243.30	123.84	99.78 ± 2.44	398.79 ± 0.61
	2	$4.41 \cdot 10^{-4} \pm 1.35 \cdot 10^{-6}$	-1.01 ± 0.15	243.30	123.84	100.41 ± 3.53	633.60 ± 1.09
	4	$4.51 \cdot 10^{-4} \pm 5.87 \cdot 10^{-6}$	-0.74 ± 0.20	243.30	123.84	100.70 ± 4.46	231.59 ± 17.34

Tab. A.10.: Inference time, loss, FLOPs, parameters, mean reward, and training time for varying numbers of attention heads across transformer-based strategies.



(a) Rewards of *SetTransformer* with varying model dimensions.

(b) Rewards of *SetTransformerOG* with varying model dimensions.



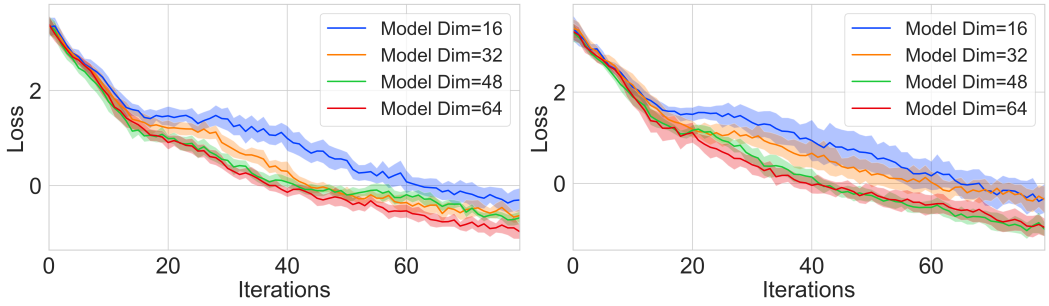
(c) Rewards of *SABTransformer* with varying model dimensions.

(d) Rewards of *ISABTransformer* with varying model dimensions.

Fig. A.5.: Rewards of transformer-based methods with varying model dimensions and normalization. The normalization negatively impacts the performance of the transformer-based models, which is why it is omitted. Figure 5.9 is the same experiment but with normalization.

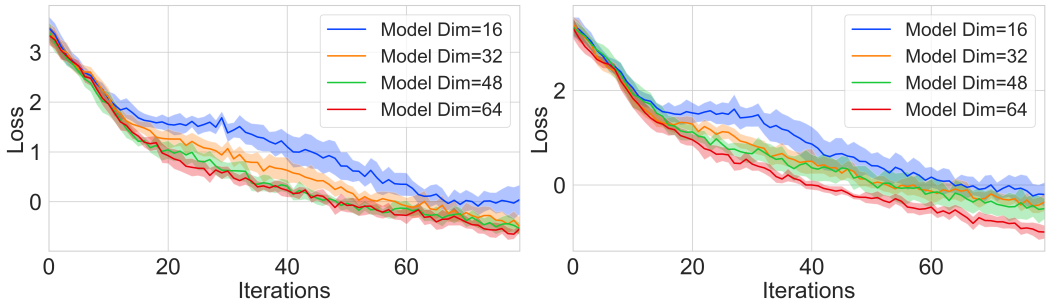
Strategy	Inducing Points	Inference Time	Loss	FLOPs	Parameters	Mean Reward	Training Time
ISAB Transformer	1	$4.40 \cdot 10^{-4} \pm 7.51 \cdot 10^{-6}$	-0.75 ± 0.16	125.80	115.08	102.88 ± 1.82	83.29 ± 0.54
	2	$4.54 \cdot 10^{-4} \pm 1.90 \cdot 10^{-5}$	-0.74 ± 0.14	154.58	115.32	99.36 ± 2.10	226.31 ± 0.80
	3	$4.60 \cdot 10^{-4} \pm 1.98 \cdot 10^{-5}$	-0.74 ± 0.22	183.36	115.58	100.58 ± 4.09	248.36 ± 0.33
	4	$4.79 \cdot 10^{-4} \pm 1.83 \cdot 10^{-5}$	-0.84 ± 0.18	212.12	115.84	102.75 ± 3.15	232.89 ± 0.39
SetTransformer	1	$4.61 \cdot 10^{-4} \pm 3.26 \cdot 10^{-5}$	-0.71 ± 0.13	182.72	127.48	99.89 ± 2.64	169.63 ± 4.24
	2	$4.50 \cdot 10^{-4} \pm 1.30 \cdot 10^{-5}$	-0.87 ± 0.22	199.38	127.68	102.69 ± 2.79	298.59 ± 0.43
	3	$4.64 \cdot 10^{-4} \pm 2.09 \cdot 10^{-5}$	-0.85 ± 0.17	216.04	127.88	95.68 ± 3.47	313.06 ± 1.52
	4	$4.53 \cdot 10^{-4} \pm 1.47 \cdot 10^{-5}$	-0.87 ± 0.18	232.70	128.06	100.72 ± 2.14	304.46 ± 0.77
SetTransformerOG	1	$4.61 \cdot 10^{-4} \pm 2.14 \cdot 10^{-5}$	-0.78 ± 0.25	193.30	123.26	97.07 ± 2.84	235.73 ± 0.94
	2	$4.42 \cdot 10^{-4} \pm 1.43 \cdot 10^{-5}$	-0.79 ± 0.14	209.98	123.46	101.78 ± 3.91	365.74 ± 0.75
	3	$4.49 \cdot 10^{-4} \pm 1.82 \cdot 10^{-5}$	-0.76 ± 0.25	226.64	123.66	97.99 ± 4.34	382.07 ± 0.87
	4	$4.57 \cdot 10^{-4} \pm 1.82 \cdot 10^{-5}$	-0.76 ± 0.15	243.30	123.84	99.78 ± 2.44	371.31 ± 0.55

Tab. A.11.: Inference time, loss, FLOPs, parameters, mean reward, and training time for varying numbers of inducing points across transformer-based strategies.



(a) Loss of *SetTransformer* with varying model dimensions.

(b) Loss of *SetTransformerOG* with varying model dimensions.



(c) Loss of *SABTransformer* with varying model dimensions.

(d) Loss of *ISABTransformer* with varying model dimensions.

Fig. A.6.: Loss of transformer-based methods with varying model dimensions.

Strategy	Count	Inference Time	Loss	FLOPs	Parameters	Mean Reward	Training Time
ISAB	1	$4.43 \cdot 10^{-4} \pm 1.02 \cdot 10^{-5}$	-0.45 ± 0.16	44.60	48.38	99.63 ± 2.17	56.72 ± 0.64
	2	$4.81 \cdot 10^{-4} \pm 1.95 \cdot 10^{-5}$	-0.75 ± 0.16	125.80	115.08	102.88 ± 1.82	86.35 ± 0.52
	3	$4.64 \cdot 10^{-4} \pm 1.12 \cdot 10^{-5}$	-0.86 ± 0.11	207.00	181.76	102.76 ± 3.88	113.84 ± 0.33
SAB	1	$4.21 \cdot 10^{-4} \pm 1.14 \cdot 10^{-5}$	-0.03 ± 0.18	32.78	14.98	89.37 ± 5.97	78.10 ± 1.67
	2	$4.64 \cdot 10^{-4} \pm 3.30 \cdot 10^{-5}$	-0.44 ± 0.14	102.14	48.26	104.71 ± 2.74	134.70 ± 0.35
	3	$4.51 \cdot 10^{-4} \pm 9.20 \cdot 10^{-6}$	-0.80 ± 0.13	171.52	81.54	100.69 ± 3.55	189.35 ± 0.45
SET	1	$4.49 \cdot 10^{-4} \pm 1.21 \cdot 10^{-5}$	-0.78 ± 0.22	106.68	71.04	101.42 ± 3.20	170.90 ± 0.43
	2	$4.64 \cdot 10^{-4} \pm 1.22 \cdot 10^{-5}$	-0.87 ± 0.22	199.38	127.68	102.69 ± 2.79	301.06 ± 0.43
	3	$4.73 \cdot 10^{-4} \pm 1.70 \cdot 10^{-5}$	-1.18 ± 0.09	292.08	184.32	101.87 ± 4.73	431.68 ± 1.13
SETOG	1	$4.64 \cdot 10^{-4} \pm 1.52 \cdot 10^{-5}$	-0.76 ± 0.21	115.74	66.82	100.87 ± 2.79	227.73 ± 0.49
	2	$4.82 \cdot 10^{-4} \pm 2.25 \cdot 10^{-5}$	-0.79 ± 0.14	209.98	123.46	101.78 ± 3.91	374.84 ± 0.65
	3	$4.69 \cdot 10^{-4} \pm 1.83 \cdot 10^{-5}$	-1.08 ± 0.27	304.20	180.10	99.18 ± 3.30	517.86 ± 0.51

Tab. A.12.: Inference time, loss, FLOPs, parameters, mean reward, and training time for varying counts across transformer-based strategies.

Cross Method Experiments

B

Strategy	Training Time	Memory Usage	Inference Time	Mean Reward	Loss	FLOPs	Params
CONCAT	40.44 ± 0.69	1319.93	$4.67 \cdot 10^{-4} \pm 1.15 \cdot 10^{-4}$	101.64 ± 2.89	−0.41 ± 0.16	37.64	18.82
DS	96.93 ± 0.33	4251.57	$5.58 \cdot 10^{-4} \pm 1.81 \cdot 10^{-5}$	98.95 ± 3.49	−0.63 ± 0.21	281.60	140.80
DS_LOCAL	55.24 ± 0.41	2351.64	$4.67 \cdot 10^{-4} \pm 1.01 \cdot 10^{-5}$	96.51 ± 3.57	−0.40 ± 0.17	83.46	41.72
DS_GLOBAL	101.49 ± 0.31	4452.07	$5.99 \cdot 10^{-4} \pm 3.09 \cdot 10^{-5}$	97.78 ± 5.33	−0.55 ± 0.31	301.26	150.62
GSAGE	48.88 ± 1.01	1690.69	$4.43 \cdot 10^{-4} \pm 1.80 \cdot 10^{-5}$	97.98 ± 2.74	−0.77 ± 0.21	11.52	9.66
GAT	138.81 ± 0.67	8379.64	$4.66 \cdot 10^{-4} \pm 2.01 \cdot 10^{-5}$	97.27 ± 4.95	−0.67 ± 0.11	72.46	61.06
GATv2	80.80 ± 0.84	4662.49	$4.49 \cdot 10^{-4} \pm 1.52 \cdot 10^{-5}$	97.65 ± 3.20	−0.59 ± 0.13	23.24	20.74
SET	106.09 ± 0.97	–	$5.33 \cdot 10^{-4} \pm 1.15 \cdot 10^{-4}$	103.35 ± 4.87	−0.77 ± 0.21	98.34	70.94
SAB	141.07 ± 0.59	–	$4.87 \cdot 10^{-4} \pm 1.75 \cdot 10^{-5}$	104.71 ± 2.74	−0.44 ± 0.14	102.14	48.26
ISAB	90.73 ± 0.38	–	$5.91 \cdot 10^{-4} \pm 1.05 \cdot 10^{-4}$	102.88 ± 1.82	−0.75 ± 0.16	125.80	115.08

Tab. B.1.: Cross method comparison in balance scenario.

Strategy	Training Time	Memory Usage	Inference Time	Mean Reward	Loss	FLOPs	Params
CONCAT	49.28 ± 0.36	1338.37	$6.15 \cdot 10^{-4} \pm 6.35 \cdot 10^{-5}$	4.62 ± 0.07	1.58 ± 0.07	40.20	20.10
DS	108.93 ± 0.62	4262.52	$8.94 \cdot 10^{-4} \pm 1.16 \cdot 10^{-4}$	4.21 ± 0.29	1.96 ± 0.08	283.66	141.82
DS_LOCAL	60.60 ± 0.45	2376.21	$6.89 \cdot 10^{-4} \pm 5.81 \cdot 10^{-5}$	3.84 ± 0.21	2.11 ± 0.04	85.50	42.76
DS_GLOBAL	111.20 ± 0.62	4488.20	$8.30 \cdot 10^{-4} \pm 4.22 \cdot 10^{-5}$	4.88 ± 0.11	1.65 ± 0.08	306.18	153.10
GSAGE	52.17 ± 0.88	1696.78	$5.27 \cdot 10^{-4} \pm 3.95 \cdot 10^{-5}$	2.47 ± 0.64	2.30 ± 0.07	11.78	9.80
GAT	143.73 ± 0.45	8369.57	$5.59 \cdot 10^{-4} \pm 2.60 \cdot 10^{-5}$	3.07 ± 0.43	2.17 ± 0.08	72.96	61.32
GATv2	83.94 ± 0.73	4669.63	$5.86 \cdot 10^{-4} \pm 7.48 \cdot 10^{-5}$	2.79 ± 0.38	2.24 ± 0.07	23.62	20.92
SET	108.06 ± 1.57	–	$5.55 \cdot 10^{-4} \pm 2.40 \cdot 10^{-5}$	4.88 ± 0.06	1.43 ± 0.06	100.66	72.10
SAB	144.43 ± 0.24	–	$6.04 \cdot 10^{-4} \pm 5.18 \cdot 10^{-5}$	4.89 ± 0.09	1.51 ± 0.08	103.68	49.02
ISAB	93.57 ± 0.51	–	$5.60 \cdot 10^{-4} \pm 2.84 \cdot 10^{-5}$	4.77 ± 0.08	1.46 ± 0.09	127.34	115.84

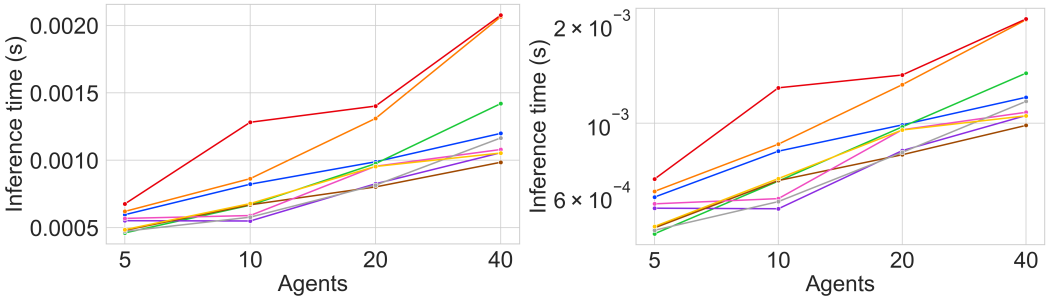
Tab. B.2.: Cross method comparison in navigation scenario.

Strategy	Training Time	Memory Usage	Inference Time	Mean Reward	Loss	FLOPs	Params
CONCAT	50.76 ± 0.87	1055.48	$5.04 \cdot 10^{-4} \pm 9.34 \cdot 10^{-6}$	25.17 ± 4.05	−1.54 ± 0.33	24.38	15.24
DS	107.66 ± 1.69	3454.27	$6.51 \cdot 10^{-4} \pm 1.37 \cdot 10^{-5}$	14.79 ± 6.74	−2.03 ± 0.85	222.82	139.26
DS_LOCAL	67.34 ± 0.75	1879.86	$5.84 \cdot 10^{-4} \pm 1.60 \cdot 10^{-5}$	23.26 ± 4.87	−1.05 ± 0.20	64.32	40.20
DS_GLOBAL	112.63 ± 1.20	3585.97	$6.48 \cdot 10^{-4} \pm 1.15 \cdot 10^{-5}$	21.28 ± 2.10	−1.26 ± 0.31	235.22	147.02
GSAGE	57.03 ± 0.82	1325.38	$5.07 \cdot 10^{-4} \pm 2.57 \cdot 10^{-6}$	8.82 ± 2.54	−0.14 ± 1.29	8.92	9.48
GAT	130.61 ± 0.58	5977.74	$5.45 \cdot 10^{-4} \pm 4.48 \cdot 10^{-5}$	9.62 ± 2.60	0.45 ± 0.17	57.34	60.68
GATv2	82.94 ± 0.22	3278.02	$5.14 \cdot 10^{-4} \pm 9.93 \cdot 10^{-6}$	11.16 ± 1.57	0.32 ± 0.18	18.12	20.44
SET	119.07 ± 0.77	–	$5.06 \cdot 10^{-4} \pm 3.54 \cdot 10^{-6}$	22.94 ± 5.11	−1.16 ± 0.23	78.18	69.22
SAB	146.81 ± 1.05	–	$5.05 \cdot 10^{-4} \pm 3.16 \cdot 10^{-6}$	23.99 ± 4.71	−1.25 ± 0.31	79.06	47.10
ISAB	107.96 ± 0.46	–	$5.03 \cdot 10^{-4} \pm 7.75 \cdot 10^{-7}$	23.34 ± 4.66	−1.29 ± 0.33	104.14	113.92

Tab. B.3.: Cross method comparison in multi-give-way scenario.

Strategy	Pooling	Training Time	Memory Usage	Inference Time	Mean Reward	Loss	FLOPs	Params
DS	mean	110.79 ± 5.72	4252.48	$6.88 \cdot 10^{-4} \pm 6.06 \cdot 10^{-5}$	98.95 ± 3.49	−0.63 ± 0.21	281.60	140.80
	max	111.61 ± 1.05	4258.13	$6.61 \cdot 10^{-4} \pm 1.63 \cdot 10^{-5}$	93.56 ± 3.01	−0.07 ± 0.19	281.60	140.80
DS_LOCAL	mean	62.39 ± 0.79	2352.12	$5.25 \cdot 10^{-4} \pm 9.64 \cdot 10^{-6}$	96.51 ± 3.57	−0.40 ± 0.17	83.46	41.72
	max	60.02 ± 0.62	2381.08	$5.11 \cdot 10^{-4} \pm 1.53 \cdot 10^{-5}$	96.82 ± 3.47	−0.36 ± 0.26	83.46	41.72
DS_GLOBAL	mean	114.34 ± 1.69	4452.36	$6.63 \cdot 10^{-4} \pm 3.09 \cdot 10^{-5}$	97.78 ± 5.33	−0.55 ± 0.31	301.26	150.62
	max	113.53 ± 1.56	4458.36	$6.35 \cdot 10^{-4} \pm 2.01 \cdot 10^{-5}$	95.37 ± 3.14	−0.05 ± 0.30	301.26	150.62
GSAGE	mean	55.17 ± 1.45	1690.87	$5.41 \cdot 10^{-4} \pm 4.02 \cdot 10^{-5}$	97.98 ± 2.74	−0.77 ± 0.21	11.52	9.66
	max	72.82 ± 0.34	2231.64	$5.37 \cdot 10^{-4} \pm 2.42 \cdot 10^{-5}$	92.13 ± 3.33	−0.66 ± 0.17	11.52	9.66

Tab. B.4.: All metrics of mean and max embedding comparison.



(a) Inference time.

(b) Inference time on logarithmic scale.

(c) Legend.

Fig. B.1.: Impact of increasing the number of agents in the balance environment on the inference time.

List of Figures

List of Tables

Abbreviations

NN Neural Network

GNN Graph Neural Network

GCN Graph Convolutional Network

LSTM Long Short Term Memory Network

MARL Multi-Agent Reinforcement Learning

MLP Multi Layer Perceptron

GAT Graph Attention Network

GATv2 Graph Attention Network Version Two

GSAGE GraphSAGE

NLP Natural Language Processing

SGD Stochastic Gradient Descent

PI Permutation Invariant

PI Permutation Invariance

PE Permutation Equivariant

VMAS Vectorized Multi-Agent Simulator

RL Reinforcement Learning

RG Research Goal

TRPO Trust Region Policy Optimization

CPI Conservative Policy Iteration

CLIP Clipped Surrogate Objective

TD temporal-difference

ReLu Rectified Linear Unit

MPE Multi-Agent Particle Environment

COMA Counterfactual Multi-Agent

DS DeepSet approaches

SG Stochastic Game

POSG Partially Observable Stochastic Game

RQ1 How do different permutation and agent count invariant embedding methods perform in MAPPO?

RQ2 Which critic architectures scale most efficiently to larger agent counts, balancing performance and computational cost?

RQ3 How well do the embeddings generalize to different numbers of agents?

RQ4 Are there best practices on how to train generalizable methods?

RQ5 Is the theoretical sample efficiency gain of PI methods apparent in the experiments?

GAE Generalized Advantage Estimation

CTDE Centralized Training Decentralized Execution

CTCE Centralized Training Centralized Execution

GSE Global State Embedding

FLOPs Floating Point Operations

MDP Markov Decision Process

POMDP Partially Observable Markov Decision Process

Dec-POMDP Decentralized Partially Observable Markov Decision Process

SwarmRL Architecture described by [23]

PPO Proximal Policy Optimization

IPPO Independent Proximal Policy Optimization

MAPPO Multi-Agent Proximal Policy Optimization

CPPO Centralized Proximal Policy Optimization

MADDPG Multi-Agent Deep Deterministic Policy Gradient

SAB Set Attention Block

ISAB Induced Set Attention Block

PMA Pooling by Multiheaded Attention

MAB Multiheaded Attention Block

PyG PyTorch Geometric

Symbols

$\mathcal{A}$ Set of possible actions. 17, 18

a Individual action taken by the agent from the action space. 17–19

a Scaling exponent. E.g. with $a = 2$ a metric M scales quadratically with N as such: $M(N) \propto N^a$. 57, 58, 84, 134

$\hat{A}_t$ Estimated advantage at time step t, measuring the relative value of an action compared to the average action in a state. 20–22, 39

$\mathbb{A}$ Set of joint actions across agents. 18

$\vec{a}$ Joint action vector across all agents: $\vec{a} = (a_1, \ldots, a_n)$. 18

α_{ij} Normalized attention coefficient, the result of applying the softmax function to e_{ij}. It indicates the normalized attention weight node i assigns to neighbor j. 35, 49, 133

$\vec{a}$ A learnable parameter vector in the attention mechanism used to compute e_{ij} from the concatenated transformed features of nodes i and j. 35, 36, 48, 49

b Bias term added to the weighted input sum. 27

c_1 Coefficient for the value function loss in the PPO objective. 21

c_2 Coefficient for the entropy bonus in the PPO objective. 21

clip() Method that restricts a value to a fixed interval. In PPO, it constrains the policy ratio $r_t(\theta)$ to the range $[1 - \epsilon, 1 + \epsilon]$ to stabilize learning. 20, 22

d' Hidden layer width of GNN critic methods. 67, 68, 130

δ TD error. 20

d_k Dimensionality of the key vectors in the attention mechanism. Also used to scale the dot product in scaled dot-product attention. 29, 30

d_o Dimensionality of the observation vector of an agent. 39

E Expected value of a random variable under a probability distribution. 17

$\hat{\mathbb{E}}_t$ Empirical expectation over a batch of samples at time step t. 20–22

$\mathcal{E}$ Set of edges in a graph. 28, 34

e_{ij} The neighbor importance score, a scalar representing how important node j is to node i in the attention mechanism. It is computed using a shared self-attention mechanism applied to the transformed node features. 35, 36, 49

ϵ Clipping parameter in PPO that defines how much the probability ratio $r_t(\theta)$ can deviate from 1. 20, 22

f Dimension of the node features. 34

f' Dimension of the transformed node features. 34, 35

f_{pe} A PE function whose output permutes in the same way as its input. 23

f_{pi} A PI function, i.e., the output does not change under input permutations. 23

γ Discount factor, trade-off between short- and long-term rewards. 17, 18, 20

$\mathcal{G}$ Graph composed of a set of nodes $\mathcal{V}$ and edges $\mathcal{E}$. 28

H Input node feature matrix, a set of input node embeddings $H = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}$, where each $\vec{h}_i \in \vec{R}^F$. 34, 47

$\vec{h}'_v$ Final embedded feature vector associated with node $v \in V$. 34

$\vec{h}^l_v$ Embedded feature vector associated with node $v \in V$, in layer l. 33, 34

$\vec{h}_i$ Feature vector associated with node $i \in V$, typically in $\mathbb{R}^d$. 28, 29, 34–36, 47–50, 133

$\vec{h}'_i$ Updated hidden representation (embedding) of node i after aggregating messages from neighbors. 28, 35, 48

$\vec{h}_j$ Feature vector associated of neighbor node $j \in \mathcal{N}_i$. 28, 29, 35, 36, 48, 49

H' Output node feature matrix, a set of updated node embeddings after attention is applied: $H' = \{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\}$, where each $\vec{h}'_i$ incorporates neighborhood information. 35

$\mathbf{I}$ Matrix of inducing points, $\mathbf{I} \in \mathbb{R}^{m \times d}$, used in induced attention mechanisms to reduce computational complexity. 37

ISAB Induced Set Attention Block, efficient variant of SAB using inducing points. 37, 38, 51, 52

K Key matrix in self attention, derived from the input matrix X. 29, 30, 36

k Layer index in a GNN architecture, $k \in \{1, \ldots, l\}$. 33, 34

k Number of seed vectors in SetTransformer, usually $k = 1$ in this thesis. 37

$L_t^{\text{CLIP}}(\theta)$ Clipped surrogate objective, stabilizes policy updates by limiting probability ratio. 21

$L^{\text{CLIP}}(\theta)$ Clipped surrogate objective in PPO that ensures stable policy updates by penalizing large changes in the policy. 20, 22

$L^{\text{CPI}}(\theta)$ Surrogate objective, based on Conservative Policy Iteration (CPI), encourages improving the likelihood of actions with positive advantage. 20

$L(\theta, \phi)$ Total PPO loss function combining surrogate policy loss, value function loss, and entropy bonus. 21

$L_t^{VF}(\phi)$ Squared error between predicted value and target value: $(V_\phi(s_t) - V_t^{\text{targ}})^2$. 21

$\| \cdot \|_2$ L2 norm (Euclidean norm). Used to normalize vectors to unit length. 34

λ GAE decay parameter, controls bias-variance trade-off in advantage estimation. 20

l Total number of message passing layers in the GNN. 33, 34, 143

m Number of attention heads, the number of parallel attention mechanisms (heads) used in a multi-head attention setup. Each head captures different aspects of the neighborhood structure. 30, 35, 49, 67, 130, 133

m Number of inducing points used to reduce computational cost in SetTransformer's attention mechanisms. 37

MAB Multihead Attention Block, computes attention between two input sets. 36, 37

$\mathcal{N}$ Set of agents in multi-agent environments. 18

$\mathcal{N}()$ Function returning the neighbors of a node v, i.e., $\mathcal{N}(v) \subseteq \mathcal{V}$. 33, 34, 143

n Number of rows in a matrix or number of elements in a set. 37, 47

$\mathcal{N}_i$ Set of neighbors of node i. 28, 34, 35, 48–50, 133

Norm Layer normalization, stabilizes and accelerates training of neural networks. 36

$\mathcal{O}$ Set of observations in partially observable environments. 18, 22, 32, 39, 45–47, 50–52, 133

$\vec{o}$ Observation received by the agent, providing partial information about the true environment state. If $\vec{o}$ is associated with an agent i, $\vec{o}_i \in \mathcal{O}$ is the feature vector, also called observation, of agent i. 22, 25, 26, 29, 31, 32, 34, 39, 45–47, 49, 51, 52, 133

$\mathbb{O}$ Set of joint observations across agents. 18

O Big-O notation describes asymptotic upper bounds of algorithms. 37

o A generic instance from the observation space $\mathcal{O}$, often used in abstract definitions of MDPs/POMDPs. In practical applications, especially with high-dimensional inputs, observations are typically represented as vectors, $\vec{o}$. 18

$\mathcal{P}$ State transition probability function. 17, 18

P Probability of an event occurring, used to define distributions over random variables. 17, 18

$\phi()$ Decoder function that is applied to a pooled embedding, typically a NN. 11, 24–26, 28, 29, 32, 38, 45–49, 51, 52, 62, 63, 67, 129, 130, 133

π A policy, mapping states to a probability distribution over actions, defines the agent's behavior. 17, 19

π_θ Policy network (actor) parameterized by θ, maps a given state to a probability distribution over actions. 19, 21, 39

$\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ Policy before an update, used in PPO. 20

π^* Optimal policy that maximizes expected total return from each state. 17

$\pi_\theta(a_t \mid s_t)$ Stochastic policy with parameters θ, representing the probability of taking action a in state s. 20, 22

$\pi(n)$ Permutation function that rearranges the order of input indices $1, \ldots, n$. 23

PMA Pooling by Multihead Attention, aggregates set elements using learnable seed vectors. 37, 38, 51, 52

$\oplus$ Pooling operation that aggregates a variable-sized set into a fixed-size representation. 24–26, 28, 29, 33, 34, 38, 48, 49, 51, 52, 129

$\psi()$ Encoder function applied independently to an input element, typically a NN. x, 11, 24–26, 28, 32, 38, 46–49, 51, 52, 63, 67, 69, 129, 130, 133

Q Query matrix in the attention mechanism, derived from the input matrix X. $Q \in \mathbb{R}^{n \times d_q}$, where n is the number of tokens and d_q is the dimensionality of the query vectors. 29, 30, 36

$\mathcal{R}$ Reward function. 17, 18

r_t Reward received at time step t from the environment after taking action a_t in state s_t. 20

R^2 Goodness-of-fit measure for linear regression. For example, if $R^2 = 0.9$, then at least 90% of the variance in the data is explained by the linear model. 57, 58, 82

$r_t(\theta)$ Probability ratio of current to old policy: $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$. 20, 22

rFF Row-wise feed-forward network, applies an MLP independently to each element in a set or sequence. 36–38, 51, 52

$\mathcal{S}$ Set of environment states. 17, 18

s Current state in the environment. 17–19

$S[\pi_\theta](s_t)$ Entropy of the policy distribution at state s_t, encourages exploration by discouraging early convergence. 21

S Set of k seed vectors with dimension d, modeled as a matrix. 37

s' Follow up state in the environment after s, e.g., current state s, next state s'. 17, 18

SAB Self-Attention Block, applies self-attention within a set. 36, 38, 51, 52

σ Activation function, e.g., Sigmoid, Tanh, ReLU. 27, 33–35, 48–50

ϕ Parameters of the critic network. 19, 21

θ Parameters of the policy network. 19, 21, 22

V Value matrix in the attention mechanism, derived from the input matrix X. 29, 30, 36, 37

$V_\phi(s)$ Estimated state-value function, parameterized by ϕ. Computed by the *critic* network, it predicts the expected return from a given state s under the current policy. 19–22, 39

V_t^{targ} Target value estimate at time step t, used to train the value function. It can be computed using temporal difference (TD) learning, Monte Carlo returns, or from generalized advantage estimation (GAE). 21

v Node index, where $v \in \mathcal{V}$. 33, 34

$\mathcal{V}$ Set of nodes/vertices in a graph. 28, 33, 34, 143, 146

$\mathbf{W}$ Learnable weight matrix used in linear projections such as computing queries, keys, and values in transformers. 29, 30, 33–36, 48, 49

w_i Weight assigned to input element x_i, used in weighted pooling. 25, 27

X X refers to a set of n elements $\{x_1, \ldots, x_n\}$. 11, 24, 32

$\mathbf{X}$ Input matrix consisting of n rows or tokens of dimensionality d_x. 29, 36–38

x_i Generic input element of an input set X, which may be a scalar or vector, depending on the context. 22–25, 27, 32

$\mathbf{Y}$ Generic matrix consisting of n rows of dimensionality d. 36, 37

y_i Generic output element of an output set Y, y_i may be scalar or a vector depending on the context in which it is used. 23

$\mathcal{Z}$ Observation probability function. 18

$\mathbf{Z}$ Output matrix of the encoder in SetTransformer, same dimensions as $\mathbf{X} \in \mathbb{R}^{n \times d}$. 37

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Declaration

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any source other than those cited in the text and acknowledgments.

Munich, September 12, 2025

Simon Hampp