SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY MUNICH

Bachelor's Thesis in Informatics

An Analysis of League-Training in Multi-Agent Reinforcement Learning Applications

Sohrab Tawana

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY MUNICH

Bachelor's Thesis in Informatics

An Analysis of League-Training in Multi-Agent Reinforcement Learning Applications

Eine Analyse von League-Training in Multi-Agent Reinforcement Learning Anwendungen

Author:	Sohrab Tawana
Supervisor:	Prof. Dr. Martin Bichler
Advisor:	Nils Kohring
Submission Date:	15.05.2023

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.05.2023 Sohrab Tawana

Abstract

This thesis investigates whether league training can be successfully applied to other multi-agent reinforcement learning (MARL) scenarios beyond its initial implementation for AlphaStar. AlphaStar is a StarCraft II agent trained by Google DeepMind using league training, a novel MARL method developed for this purpose. This thesis compares the performance of agents trained using league training with the performance of agents trained using only self-play. This comparison is made in the context of a digital soccer simulation environment, SimpleSoccer.

The research involves setting up a league training environment in the existing codebase of the Chair of Decision Sciences & Systems, thereby providing a basis for comparing the effectiveness of league training with that of self-play training. A variety of validation agents are used as benchmarks for evaluating agent performance between the two training methods under study. The primary metric for performance evaluation is the win-rate of the trained agents against these respective validation agents.

Analysis of league training in additional MARL applications provides insight into its strengths, limitations, and potential for generalization. Specifically, the performance of MARL policies trained using self-play and league training is evaluated under various initialization scenarios, including untrained, reinforcement learning, and imitation learning policy initialization.

The results show that league training slightly outperforms self-play against weak validation agents under most initialization conditions, and matches or underperforms self-play against strong validation agents under most initialization conditions. However, the additional computational cost of league training makes it less efficient in the setting of this thesis, as the performance differences are not substantial. Future research directions include the incorporation of expert knowledge and the development of more sophisticated reward structures to improve agent performance in league training.

Contents

1. Introduction

Multi-agent reinforcement learning is an area of growing interest in the field of artificial intelligence, with applications ranging from robotics and autonomous systems to computational economics and finance. In contrast to single-agent reinforcement learning, MARL focuses on environments in which multiple agents interact, cooperate, or compete with each other. While the potential for MARL is vast, training effective agents in such complex environments presents several unique challenges [1].

1.1. Motivation

One of these challenges is the non-stationary nature of MARL environments. This complicates the learning process, as agents are constantly adapting to each other's strategies, often leading to suboptimal behavior or non-convergence. This can cause self-play training algorithms to stagnate and agents to develop limited, easily exploitable strategies.

League training was introduced to address these challenges among others and was successfully used to train AlphaStar, an agent designed to play the real-time strategy game StarCraft II [2]. This training method exposes agents to different opponents with different strategies, allowing them to develop a strong and generalized understanding of the game. The effectiveness of league training was demonstrated when AlphaStar reached the Grandmaster level in all three StarCraft races, outranked 99.8% of all human players on the official ladder, and even defeated professional human players.

The motivation for this thesis is to investigate the applicability of league training to other MARL applications.

1.2. Research Objective

The primary objective of this thesis is to analyze whether league training can be applied to other MARL applications to train agents that outperform those trained by self-play alone. Specifically, this research will focus on conducting experiments to evaluate the performance of agents trained using league training versus those trained using self-play. By conducting an analysis of league training applied to an additional MARL application, this thesis aims to provide insight into the strengths and limitations of this approach, as well as its potential for generalization to other applications.

This thesis aims to contribute to the broader reinforcement learning (RL) and artificial intelligence community by investigating the applicability and effectiveness of league training in various domains, including additional MARL applications, thereby providing a better understanding of this method and promoting the development of robust and versatile agents capable of thriving in complex, multi-agent environments.

1.3. Related Work

This thesis will refer to concepts from previous work. This section introduces the most important of them.

Actor-Critic Reinforcement Learning

Actor-critic reinforcement learning (ACRL) is a paradigm in RL that combines the strengths of both policy-based and value-based methods. ACRL uses a two-component architecture: the actor, which generates actions based on the current policy, and the critic, which evaluates the quality of those actions by estimating the associated state-value function. The actor and critic networks are trained in parallel, with the critic's feedback used to update the actor's policy. The iterative process typically involves the use of temporal difference learning and gradient-based optimization methods. This combination allows ACRL to effectively balance exploration and exploitation, resulting in improved convergence and stability [3].

One of the main benefits of the critic component in ACRL is its ability to reduce the variance in the learning updates. By providing a more accurate estimate of the state-value function, the critic helps the actor focus on actions that are more likely to yield long-term rewards. This reduces the noise in learning updates and accelerates convergence towards an optimal policy. Furthermore, the critic's role in estimating state values helps to stabilize the learning process, avoiding the large fluctuations that can occur in policy-based methods alone. This results in a more robust and efficient learning process, especially when dealing with complex, high-dimensional problems [4]. ACRL has been successfully applied to a wide range of complex, high-dimensional problems, making it a popular choice in RL research and real-world applications [5].

Proximal Policy Optimization

Proximal policy optimization (PPO) is a specific ACRL algorithm introduced by OpenAI in 2017 that aims to improve the training stability and sample efficiency of policy gradient methods [6]. PPO addresses the problem of large policy updates, which can cause instability in learning, by introducing a trust region constraint. This constraint ensures that the updated policy does not deviate too much from the old policy, allowing for more stable and reliable learning. PPO achieves this by using a clipped objective function that penalizes the policy updates if they exceed a predefined threshold. The key advantage of PPO is its ability to balance the exploration-exploitation tradeoff and provide stable learning updates, making it a popular and effective choice for various RL problems, including those with continuous action spaces.

1.4. Methodology

In this thesis, we investigate the applicability of league training to different MARL scenarios. To do so, we first select representative environments that capture the challenges and characteristics of multi-agent systems. For this purpose, we choose the SimpleSoccer environment.

Before applying league training, it is important to establish baseline algorithms for comparison. We consider different approaches to MARL, such as self-play and independent learners. Self-play is a training technique where agents learn by competing against themselves or earlier versions of themselves, which has been shown to be particularly effective in games and competitive environments. Independent learners are agents that learn their own policies without considering the strategies of other agents, simplifying the learning process but potentially leading to suboptimal performance in scenarios where coordination and adaptation are critical. We use ACRL and PPO as baseline algorithms because they have been shown to be effective in various RL applications. By comparing the performance of agents trained only with these baseline algorithms to agents trained with league training, we can evaluate the effectiveness of the league training approach.

Next, we implement league training following the methodology outlined in the AlphaStar project. This process includes the implementation of the league, the different types of agents, the matchmaking, and the performance evaluation mechanisms. We adapt the league training implementation to the chosen environment and problem formulation while ensuring that the approach remains generalizable across different MARL applications.

Finally, to evaluate the performance of agents trained using league training, we conduct a series of experiments comparing these agents to those trained using self-play. The main evaluation metric we use is the win-rate against validation agents.

1.5. Overview of Results

This thesis investigates the effectiveness of league-training in multi-agent reinforcement learning (MARL) applications, as compared to self-play training, through a series of experiments. The experimental framework includes three types of policies to initialize the agents with: untrained, reinforcement learning (RL), and imitation learning (IL) policies. The performance of agents trained by both methods is evaluated against three levels of validation policies, as well as a handcrafted policy as a level four validation policy.

In experiments with the untrained policy, the league-trained agents perform better against the lower-level validation policies, but worse against the higher-level validation policies. With the RL policies, the league-trained agents again perform better against the lower-level validation policies, and the same or worse against the higher-level validation policies. Experiments with IL policies show different performance depending on the amount of expert data used for the IL. The trend of league-trained agents performing better against the lower-level validation policies and worse against the higher-level validation policies still persists. In general, league training shows robust performance, often matching or exceeding self-play training, although the effectiveness varies depending on the specifics of the initialization policy.

1.6. Thesis Structure

The structure of this thesis follows a logical progression, beginning with a basic understanding before moving on to practical application and analysis. It is divided into four chapters.

The thesis begins with an "Introduction" that outlines the motivation for this study, the research objective, relevant previous studies, the chosen methodology, and a brief overview of the findings.

The focus then shifts to "League Training", the core topic of this thesis. The chapter provides a detailed introduction to league training, specifically examining its implementation in Google DeepMind's AlphaStar. The specifics of league training, such as the types of agents used, prioritized fictitious self-play, training using human gameplay statistics, and agent policy initialization with supervised learning, are explored. We also describe the training environment, specifically the SimpleSoccer environment and our modifications to it. Finally, this section highlights the implementation process of league training and discusses the differences between the AlphaStar and SimpleSoccer implementations.

The third section, "Experimental Analysis", presents the experimental setup and performance analysis. It starts with details about the league training setup, the self-play training setup, the seeding of the experiments, the preparation of the validation policies, and the performance evaluation metrics. The results of the performance analysis of different policy initializations (untrained, reinforcement learning, and imitation learning) are presented, followed by a comparison of initialization policies.

The final section, "Conclusion", summarizes the results and a provides a discussion of them. This section also outlines potential future research directions, providing a springboard for further investigation into the topic.

2. League Training

2.1. Introduction to League Training

The concept of league training is to expose a RL agent to a variety of different strategies, in order to make its strategy as robust as possible. Initially, each agent is initialized with IL or supervised learning (SL) based on human match replays to help the policies converge to useful strategies. Throughout the league training process, agents play matches against each other, adjusting their strategies and improving their play. The league is periodically populated with checkpoints of the agents' strategies, which are then used as new agents with frozen policies, called historical agents [2]. In this thesis, all agents other than historical agents are called "learning agents".

2.2. AlphaStar League Training Implementation

In the AlphaStar implementation, league training involves three pools of learning agents, each of which plays a specific role in the learning process. The following sections provide an overview of the different types of agents and their characteristics.

2.2.1. Types of Agents

Main Agents

Main agents form the primary focus of league training, and the goal is to improve their performance. Their potential training partners include all the main agents (including themselves) and historical agents in the league. After reaching a certain number of trained steps, a checkpoint of the agent's strategy is added to the league as a historical agent [2].

Main Exploiters

Main exploiters aim to identify and exploit weaknesses in the strategies of the main agents. These exploiters play exclusively against main agents and their historical agents, thus learning their past and present weaknesses. After reaching a certain win-rate against main agents, or after a certain number of trained steps, a checkpoint of their strategy is added to the league as a historical agent. During the checkpointing process, the main exploiters' strategies are reset to the initial SL strategy, allowing them to discover and exploit new weaknesses. This challenge to the main agents helps prevent chasing cycles and ensures the development of robust strategies [2].

League Exploiters

League Exploiters are similar to main exploiters, but they target the entire league instead of just the main agents. They train against all the historical agents in the league, developing strategies that exploit common weaknesses shared by the entire league. When their strategy reaches a certain win-rate against all historical agents, or after a certain number of trained steps, a checkpoint of their strategy is added to the league as a historical agent. During the checkpointing process, league exploiters' strategies have a 1 in 4 chance of being reset to the initial SL strategy. League exploiters ensure that main agents learn to counter a range of strategies that no player in the league can beat, further promoting generalization and robustness [2].

Historical Agents

Historical agents are checkpoints of learning agent strategies, that are copied and frozen. They serve to maintain a diverse set of opponents in the league and ensure that the main agents continue to train against a variety of strategies as the league evolves and new agents are introduced [2].

2.2.2. Prioritized Fictitious Self-Play

In league training, each learning agent plays one match at a time against an opponent from the league. In order to obtain a valuable learning signal, league training introduced a special way of assigning an opponent to the agents before each match. MARL in self-play can chase cycles. Including all historical agents in the league as potential opponents can mitigate this problem. This approach is inspired by fictitious self-play (FSP). However, many matches would be played against weak opponents that the agent almost always beats, resulting in a low learning signal during those matches. To address this, Google DeepMind introduced the concept of prioritized fictitious self-play (PFSP). With PFSP, the matchmaking function does not randomly sample opponents from the league with a uniform distribution. Instead, the sampling distribution is based on the learning agent's win-rate against each opponent. PFSP focuses on a certain set of agents, depending on the weighting function used. PFSP tries to find the optimal opponent for each learning agent. The optimal opponent for a learning agent is neither too hard to beat nor too easy to beat, i.e. the optimal opponents for a learning agent play strategies against which the learning agent's strategy has a win-rate between 20% and 70%. The win-rates are continuously updated, providing the matchmaking function with data on the performance of the entire league. This approach with PFSP results in more efficient learning while preventing the development of easily exploitable strategies. More details about PFSP can be found in the Appendix A and in the original AlphaStar paper [2].

2.2.3. Training with Human Gameplay Statistics

The Google DeepMind team extracted statistics, denoted as z, from the human gameplay replays used for initialization with SL. These statistics include among other things the first significant actions taken by human players, such as the first 20 buildings constructed and units built. At the beginning of the league training, the agents' policy parameters are initialized based on the supervised policy, and during league training, the main agents are trained using pseudo-rewards derived from the distance between actions produced by the trained RL policy and the statistics z. Incorporating the statistics z from human gameplay into the AlphaStar training process was critical for enhancing exploration, fostering diverse decision-making, and ultimately achieving optimal RL performance [2].

2.2.4. Agent Policy Initialization with Supervised Learning

In the original AlphaStar paper, the Google DeepMind researchers initialized their agents using supervised learning on actions sampled from replays of human gameplay from the top 22% of human players. As mentioned in the previous paragraph, they also used this human gameplay data to give the agents pseudo-rewards during the RL in the league training.

Since the SimpleSoccer environment lacks human gameplay data and a human interaction interface to create the human gameplay data ourselves, using human gameplay data is not feasible. The alternative is to use a handcrafted policy created by the author of the environment. This policy is analogous to a real soccer strategy, where players on each team have roles and act accordingly. The handcrafted policy first extracts relevant information such as ball and player positions from the observations. It then calculates the distance between the ball and each player. Based on these distances, players are assigned roles (attacker, defender, or keeper) with priority given to the player closest to the ball. Next, target positions are calculated for attackers, defenders, and keepers, and motion actions are calculated to reach these target positions. Kick actions are also determined, with a player only kicking if he is aiming at the goal. In the handcrafted policy, the attacker always dashes to the ball, while the defender and keeper do not.

Agents should be initialized with this handcrafted policy before starting league training. This is where imitation learning comes in. Imitation learning, also known as learning by demonstration or apprenticeship learning, is a type of machine learning in which an agent learns to perform tasks by observing and imitating the actions of an expert. Instead of learning by trial and error or having to design a reward function to induce the agent to learn the desired behavior, the agent learns by imitating the expert's behavior to accomplish the desired task [7].

To facilitate IL, expert data is collected by playing games between two agents that both followed the handcrafted policy. During the games, the agent's observations and the corresponding actions returned by the agent's handcrafted policies are recorded. The amount of data in terms of the number of interactions (steps) during the recording of the expert data determines the degree of behavioral proximity that the trained policy will have to the handcrafted policy. To evaluate league training with different levels of proximity, policies are pre-trained using expert datasets with 40,000, 50,000, 200,000, and 400,000 interactions. The expert datasets are used to pre-train the default CustomActorCriticPolicy using the simplest form of IL called "behavior cloning" [8]. The entire IL process is adapted from the StableBaselines team [9].

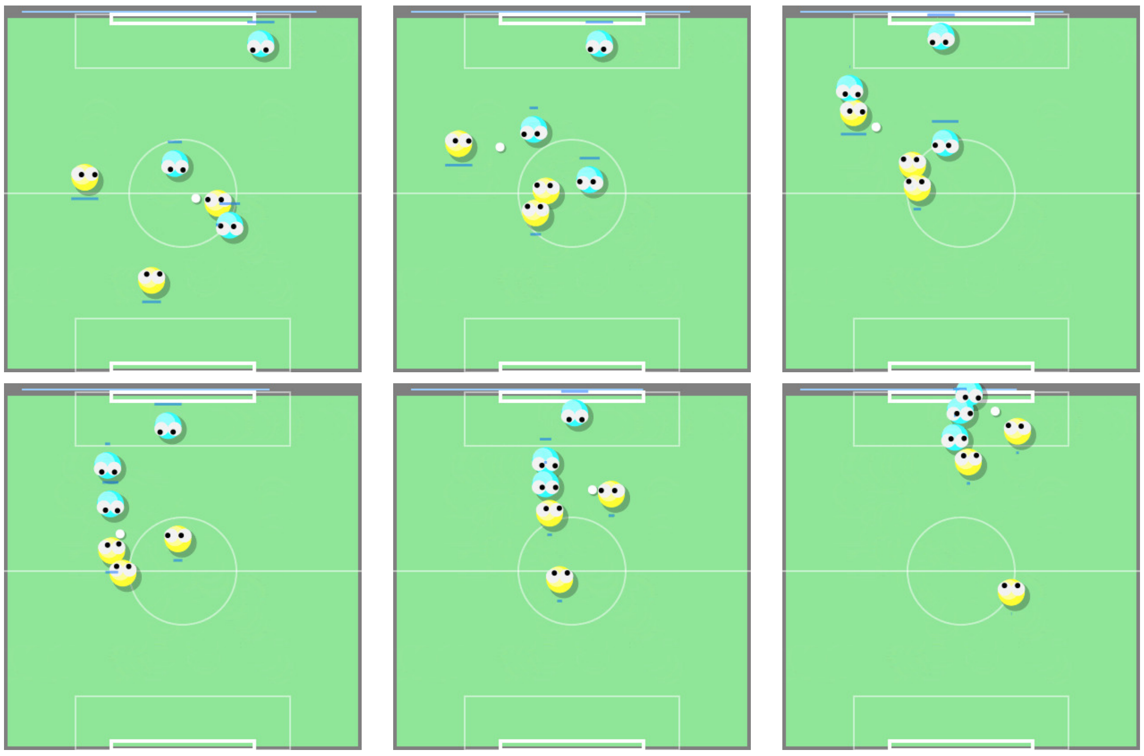

Figure 2.1.: Screenshots of a SimpleSoccer match.

2.3. Training Environment

In this thesis, we analyze league training in a MARL context different from the StarCraft II environment used by Google DeepMind to train AlphaStar. Instead, we apply league training to the SimpleSoccer environment developed by the Chair of Decision Sciences & Systems, which is based on a GitHub project by neitzal [10].

2.3.1. SimpleSoccer

SimpleSoccer is a simulation in which two teams, each consisting of three players, compete in a simplified version of soccer. Each player can perform independent actions, represented as a one-dimensional tensor containing three integers. The first integer encodes the direction of the player's movement, with the numbers zero to seven representing the eight possible discrete angles of movement, and the number eight representing no movement. The second binary integer indicates whether the player is dashing, which allows for faster movement at the cost of energy consumption, or moving at standard speed while recovering energy. The third integer encodes the player's kick action, with options including no kick, soft kick, and hard kick. Agents are only rewarded when the ball is in close proximity to the opponent's goal. A match ends when either a goal is scored or the ball is knocked out of play. The maximum duration of a game is 165 steps, with each step consisting of a cycle where observations are entered into the policy, actions are received from the policy, and the actions are applied to the environment. More details can be found in the GitHub repository [10].

2.3.2. Modifications to the SimpleSoccer Environment

The original SimpleSoccer implementation by the Chair of Decision Sciences & Systems had each player controlled by a single agent. This differed from the AlphaStar paper where a single agent controls all actions of a team. To more closely replicate the setup of the original paper, we modified the SimpleSoccer environment to allow a single agent to control an entire team.

To achieve this, we adapted the observation and action spaces of the SimpleSoccer environment to be flexible and able to accommodate the number of players controlled by an agent. As a result, each agent now controls the entire team, by receiving observations from all team members, and producing actions for each player.

2.4. Implementing League Training

In order to implement league training on the existing code base of the Chair of Decision Sciences & Systems, we implemented several functionalities and modified certain parts of the code. A UML diagram of the setup is provided in Figure B.1, and a detailed explanation can be found in Appendix A.

2.4.1. Policy Used By Agents

Within the SimpleSoccer environment, the agents use a CustomActorCriticPolicy. This policy class extends StableBaselines3's ActorCriticPolicy class [11] with several modifications. One such modification is the introduction of the "action dependent std" parameter, which allows the standard deviation of the action distribution to be a function of the input state, rather than a constant. This is achieved by conditioning the standard deviation on the latent policy representation and overriding the "get action dist from latent" method to handle this functionality. The class also provides a new method, "get stddev", to obtain the average standard deviation of the mixed strategy given an observation. In addition, this custom class allows the user to pass an optional "action activation fn" to apply an activation function to the action output, which can be useful for constraining the action space.

2.4.2. Implementation Process

The implementation process starts with the creation of the league, which allows agents to join the league and train against each other. The next step is to manage agents more effectively by resetting win-rates along with their policies to maintain accurate win-rate evaluations, and handling agent weights and training steps before and after each match.

From the beginning, a sophisticated logging and monitoring system is developed to facilitate debugging of the league training setup. This includes creating separate directories for validation and training logs, improving the verbosity and readability of logs, and exporting win-rates as CSV files for improved post-experiment analysis. More detailed information is added to log entries, parts of the initial weights are logged, and log files are saved with varying levels of verbosity. This provides a wealth of data from which to learn.

A key performance indicator for league training is the evaluation of the trained agents against validation agents that were trained with RL in self-play and kept outside the league as benchmarks. Evaluating agents within the league provides performance data for the matchmaking function to enable it to select optimal opponents for the learning agents using PFSP. Win-rate thresholds are calculated by evaluating the validation agents against each other to provide metrics for comparing the performance of the main league-trained agent with that of the self-play-trained agents.

Throughout the implementation process the code is refactored, refined, and cleaned up at each stage.

2.5. Differences Between the AlphaStar and SimpleSoccer Implementations

The league training setup of this thesis differs from the original AlphaStar setup [2] in several ways. One of the primary differences is the training environment. A professional one-on-one match in StarCraft II typically lasts between 15 and 30 minutes, with AlphaStar performing about 22 actions every 5 seconds, or about 3960 to 7920 actions per match, depending on its duration. In contrast, the maximum number of steps for SimpleSoccer is 165, meaning that SimpleSoccer agent performs only 2.1% to 4.2% of the actions performed by an AlphaStar agent. In addition, StarCraft II is an imperfect information game due to the "fog of war" that limits players' visibility and knowledge of their opponents' movements, units, and strategies on the game map. However, SimpleSoccer is a perfect information game where agents can see the entire field, including the positions of all players and the ball.

Another difference is the policy used by the agents. While the AlphaStar agents use a complex ensemble of networks with 139 million weights distributed across multiple networks [2], the SimpleSoccer agents use a CustomActorCriticPolicy with 15,851 weights. In addition, to train AlphaStar, the Google DeepMind team derived statistics z from each human gameplay replay, which encodes the actions taken during the match. The trained policies were conditioned on this statistic z during both SL for the initial policy and RL during league training. In the case of SimpleSoccer league training, agent policies are not conditioned on expert data; instead, the expert data is only used to pre-train agent policies before league training through a special IL method called "behavior cloning".

The pre-training process also differs between the two implementations. AlphaStar's expert data consists of 971,000 replays from the top 22% of human players, while the expert data used in IL for SimpleSoccer comes from a single handcrafted policy provided by the environment creator. After SL, AlphaStar agents were fine-tuned using only winning replays of high-performing players, while SimpleSoccer agents are not fine-tuned after IL.

Finally, the league training setup itself differs between the two implementations. AlphaStar's league training started with one main agent, one main exploiter, and two exploiters per race, for a total of 12 learning agents. Since SimpleSoccer does not have different races, the SimpleSoccer league starts with one main agent, one main exploiter, and two league exploiters, resulting in four learning agents. The rewards during the league training are also different. AlphaStar agents received terminal rewards at the end of each match and pseudo-rewards for actions based on their distance from the statistics z extracted from human replays used in the initial SL. In contrast, SimpleSoccer agents are only rewarded when the ball is very close to the opponent's goal.

3. Experimental Analysis

3.1. Experiment Setup

In this chapter, the effectiveness of league training is compared with a classic self-play setup in several experiments. The experiments are differentiated by the weights used to initialize each agent's policy at the beginning of the experiments. The agents in both training setups have a CustomActorCriticPolicy that is trained using PPO.

3.1.1. League Training Setup

League training matches are played simultaneously on 20,000 separate SimpleSoccer environments until all of them are finished at least once. Environments are considered finished when a goal is scored or when the ball leaves the field and is out of play. Environments that finish early are restarted and continue until the last environment of the initial 20,000 environments finishes. The maximum length of an environment is 165 steps. The league training experiments run for a total of 2000 matches, where each of the 20,000 environments is considered as a separate game between the agents, i.e., every single match is played simultaneously on 20,000 separate SimpleSoccer environments. During training, the main agent's policy is periodically evaluated against the benchmark validation policies.

3.1.2. Self-Play Training Setup

Each self-play training also runs simultaneously on 20,000 separate SimpleSoccer environments. This high number of training environments for both league training and self-play is necessary due to the sample inefficiency of current deep learning techniques. For a fair comparison, the length of the self-play training (number of steps) is adjusted to the number of steps for which the main agent was trained in the league training. League training is a generalization of self-play, where the opponents are changed for more variety. In the self-play setup, two agents play against each other using the same policy during training. The "experience" of both agents, i.e., observations, corresponding actions, and rewards, is used to train the policy. During training, the main agent's policy is periodically evaluated against the benchmark validation policies.

3.1.3. Seeding of the Experiments

Each self-play and league training experiment is run three times with different seeds (0, 1, and 2). This reduces the effects of stochasticity on the results of the experiments. The curves in the graphs represent the mean of the three different seeds from each experiment, and the same colored shaded area around the mean represents the standard deviation of the three different seeds.

3.1.4. Preparation of Validation Policies

In preparation for the experiments, validation policies are trained. Their purpose is to serve as benchmarks against which the performance of the trained agents can be evaluated during the experiments.

Three validation policies are trained using RL in self-play. The training runs for 960 million steps, during which the learning agent's policy is periodically evaluated against the handcrafted policy. In addition, the weights of the agent's policies are periodically saved. After the training, the performance of the trained policy against the handcrafted policy is analyzed, and three different levels of performance are selected. This results in three validation policies that are trained for 160 million, 256 million, and 480 million steps, respectively. For simplicity, they are referred to as "level-one", "level-two", and "level-three" policies, based on their win-rate against the handcrafted policy and the number of steps trained.

The fourth and final validation policy is the handcrafted policy mentioned above, which is provided in the SimpleSoccer environment.

3.1.5. Performance Evaluation Metrics

The main performance metric used in the experiments is the win-rate of the agent against the respective validation agents. This data is presented in the form of graphs. The horizontal axis of the graphs represents the steps trained by the validated agent. The vertical axis of the graphs represents the win-rate of the validated agent against the respective validation agents. During a validation run, the learning agent plays one match against each of the validation policies, where each match is played simultaneously on 20,000 SimpleSoccer environments. The learning agent's performance against the validation opponent is evaluated based on the rewards it receives per environment. Environments in which the learning agent's reward is higher than its opponent's are considered wins, unless the difference in total rewards is less than the draw margin 0.01, in which case these environments are considered draws. The win-rate is calculated using the following formula:

$$\text{win-rate} = \frac{\text{wins} + 0.5 \cdot \text{draws}}{\text{games}}$$

This performance evaluation metric allows for a comprehensive understanding of the agents' progress during the experiments. By comparing the win-rates of agents trained using league training and self-play, the effectiveness of the league training method can be evaluated. In addition, the performance of the agents against validation policies of varying difficulty levels provides insight into the generalizability and adaptability of the agents' learned strategies.

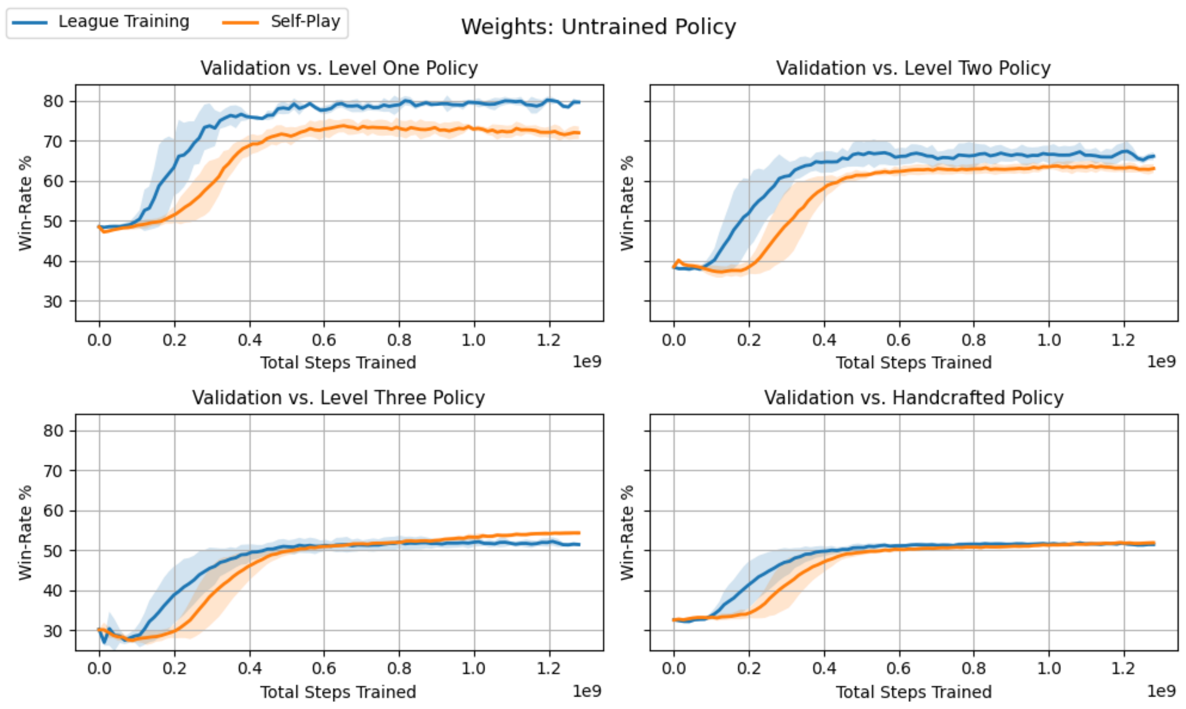

Figure 3.1.: Win-Rates vs. validation policies during training started with untrained weights.

3.2. Experimental Performance Analysis

3.2.1. Untrained Policy Initialization

In the first pair of experiments, agent policies in both the league training and self-play training methods are randomly initialized with default, untrained instances of the CustomActorCriticPolicy, following common practice.

The graph showing the performance of trained agents against the level-one policy in Figure 3.1 shows that agents trained with league training consistently outperform agents trained with self-play against the level-one policy. Both training methods show significant performance gains up to about 300 to 400 million training steps. Beyond this point, the performance curves for both groups of agents begin to plateau. The league training curve maintains a slight upward trend until the end of the training, while the self-play training curve shows a slight downward trend after 1 billion training steps.

Similarly, the graph of the performance of trained agents against the level-two policy in Figure 3.1 shows comparable performance trajectories during training. However, the difference between agents trained using league training and self-play is not as pronounced as the difference observed when validating against the level-one policy.

The graph of the performance of trained agents against the level-three policy in Figure 3.1 begins with a similar pattern. The main difference is that the self-play curve does not plateau as much as the league training curve. Instead, the self-play curve maintains a linear slope and surpasses the performance of the league-trained agents after about 600 million training steps.

The final graph of the experiments with untrained weights initialization in Figure 3.1 illustrates the performance of trained agents against the handcrafted policy. This graph is very similar to the one against the level-three policy, with the only difference being that the slope of the self-play curve is less steep, so that it surpasses the league training curve later, at about 1.1 billion training steps.

Performance Overview for Initialization with Untrained Policy

To provide a comprehensive overview of the performance progression in this section across validation opponents and training methods in this section, a synthesis of the previous observations is presented. The training graphs can be divided into two distinct sections. In the first section, both training methods experience a sharp increase in performance up to about 300 to 400 million training steps. From that point on, in the second section, the slope of both curves decreases until they plateau or maintain a slight upward or downward trend, depending on which validation policy they are evaluated against. In the first section, the league-trained agents outperform the self-play-trained agents against all validation policies. In the second section, the league-trained agents continue to outperform the self-play-trained agents against the level-one and level-two policies. However, the self-play-trained agents eventually catch up to the league-trained agents against the more difficult level-three and handcrafted validation policies, and eventually even slightly outperform the league-trained agents against the level-three policy.

3.2.2. Reinforcement Learning Policy Initialization

In the following set of experiments, both training methods are examined with agents initialized with weights corresponding to the set of the three validation policies - level-one, level-two, and level-three policy. These are pre-trained with RL in self-play.

Level-One Policy

Initially, the agents of both training methods are initialized with the weights of the least trained level-one validation policy. All four graphs show similar progressions to those observed in the previous set of experiments with untrained weights. However, the steep performance increase at the beginning of training lasts for about 200 million steps, but during this shorter period of steep performance increase, the performance jump is similar to that of the trainings that started with the untrained policy. Another difference from the previous experiment is that the league-trained agents begin to outperform their self-play counterparts only after this steep performance gain phase. After the steep performance gain, the performance graphs of all training methods plateau and generally remain constant until the end.

In the first graph in Figure 3.2, we can see that the league-trained agents begin to outperform the self-play-trained agents against the level-one policy by about 5 percent after the initial

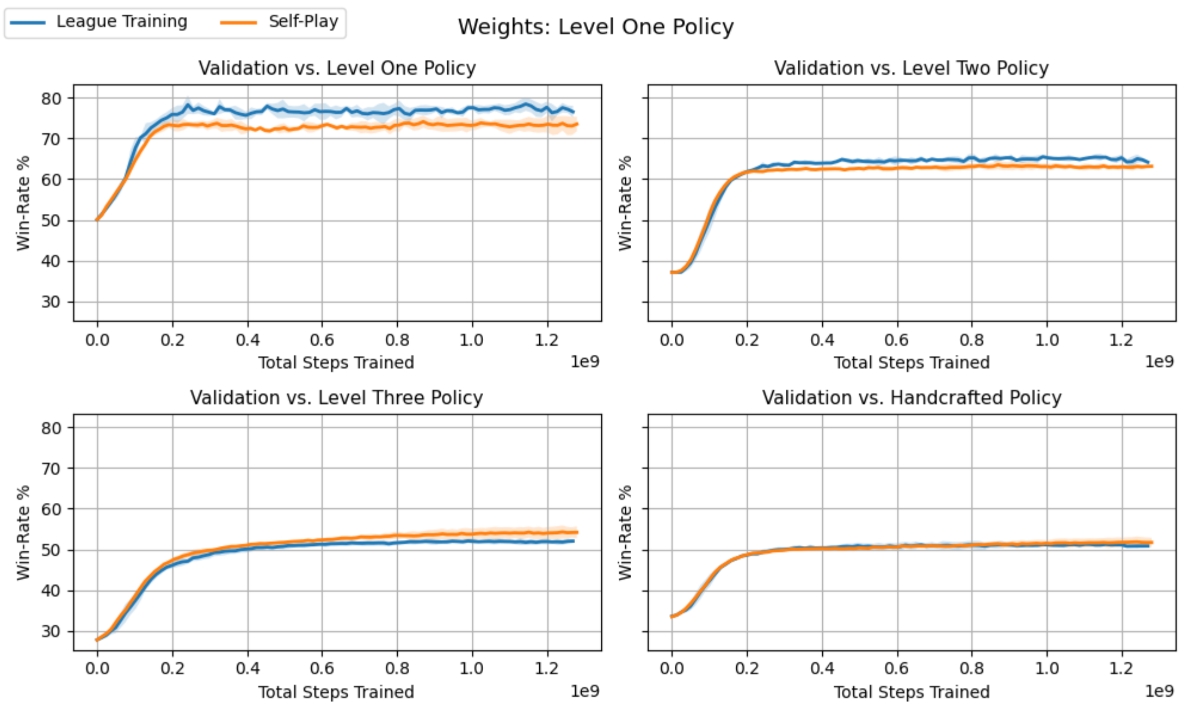

Figure 3.2.: Win-Rates vs. validation policies during training started with level-one policy.

performance gain phase. The performance of the league-trained and self-play-trained agents against the remaining validation policies is quite similar. League-trained agents perform slightly better against the level-two policy, while self-play-trained agents perform slightly better against the level-three policy. Both training methods show comparable performance against the handcrafted policy.

Level-Two Policy

Next, the agents of both training methods are initialized with the level-two validation policy. The performance graphs against the validation policies in Figure 3.3 show that the progression of the curves is similar to the level-one policy graphs in Figure 3.2. The steep performance gain phase at the beginning is shorter than in the previous experiment - it lasts for about 100 million steps. The performance gain across all validation policies is not as pronounced as before when initializing with less pre-trained policies.

Initialization with the level-two policy does not result in significant performance differences between the two training methods examined. When competing against the level-one policy, the league-trained agents again slightly outperform the self-play-trained agents, although the difference is not as pronounced as for the untrained and the level-one policy initialization, and the difference disappears toward the end of training. The league-trained agents also slightly outperform the self-play-trained agents against the level-two policy. Against the level-three and handcrafted policies, the performance curves of both the league-trained and

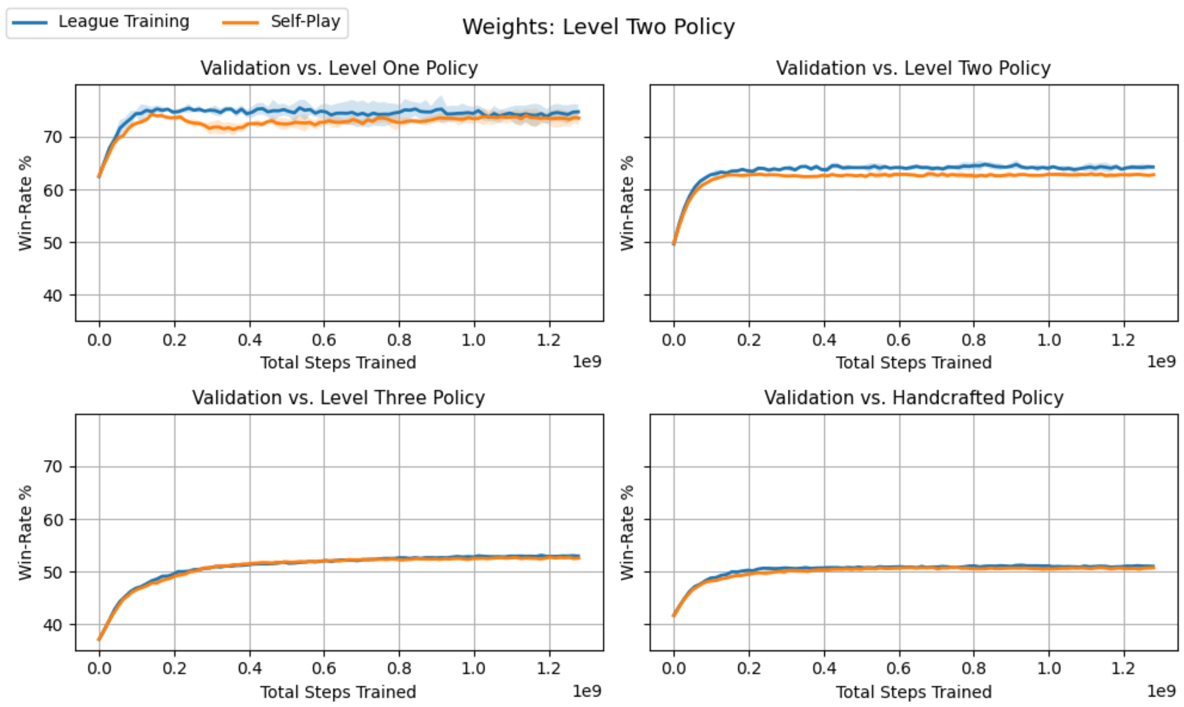

Figure 3.3.: Win-Rates vs. validation policies during training started with level-two policy.

self-play-trained agents are almost indistinguishable.

Level-Three Policy

Finally, the agents of both training methods are initialized with the most extensively trained level-three validation policies. The performance graphs for these trainings in Figure 3.4 do not show a steep initial performance gain for either training method. Both training methods remain nearly constant from the start. Initialization with the level-three policy is not well suited for performance gains over any of the validation policies. Both self-play-trained and league-trained agents show comparable performance against all validation opponents.

Performance Overview for Initialization with Reinforcement Learning Policies

Throughout the experiments initialized with the RL policies used for validation (level-one, level-two, and level-three policies), we observe a variety of performance trends across validation opponents and training methods.

When agents are initialized with the level-one policy, league-trained agents outperform self-play-trained agents against the level-one policy. Performance differences between the two training methods are small against the other validation policies, and both methods show comparable performance against the handcrafted policy.

For the level-two policy initialization, the performance curves follow a similar trajectory

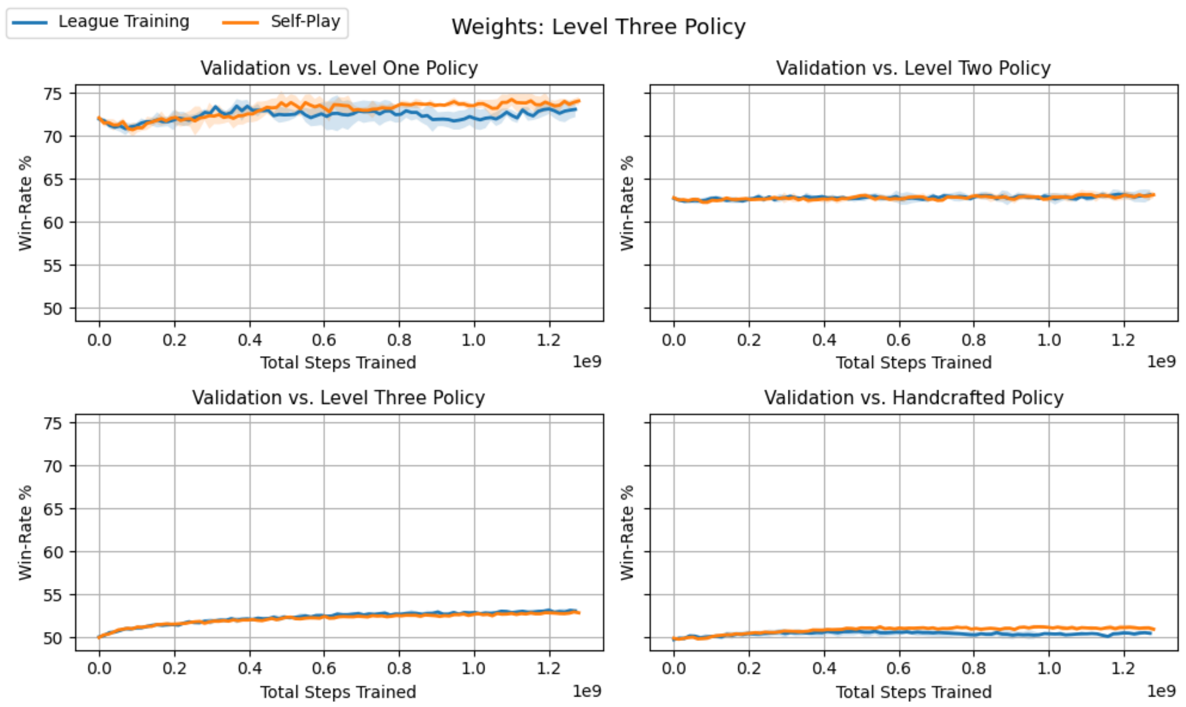

Figure 3.4.: Win-Rates vs. validation policies during training started with level-three policy.

as for the level-one policy initialization. However, the performance differences between the league-trained and self-play-trained agents are less pronounced compared to initializations with less pre-trained policies. Against the level-three and handcrafted policies, both methods show almost indistinguishable performance.

Initializing agents with level-three policies does not significantly affect performance gains against any validation policy. Both self-play-trained and league-trained agents show similar performance against all validation opponents, with no noticeable performance gains over the course of the training.

In summary, in all RL policy initialization experiments, except for the level-three policy, the league-trained agents perform better against the level-one and level-two policies. However, the performance differences between the league-trained and self-play-trained agents diminish as the initialization policies increase in steps trained. Against the level-three and handcrafted validation policies, both training methods show similar performance, regardless of the initialization policy.

3.2.3. Imitation Learning Policy Initialization

This section presents the results of initializing agents with policies that have been pre-trained using IL. Four policies were pre-trained using different amounts of expert data, and named after the number of interactions in the expert data used for their IL (e.g., a policy trained using expert data with 40,000 interactions is called a "40k policy"). Experiments were conducted

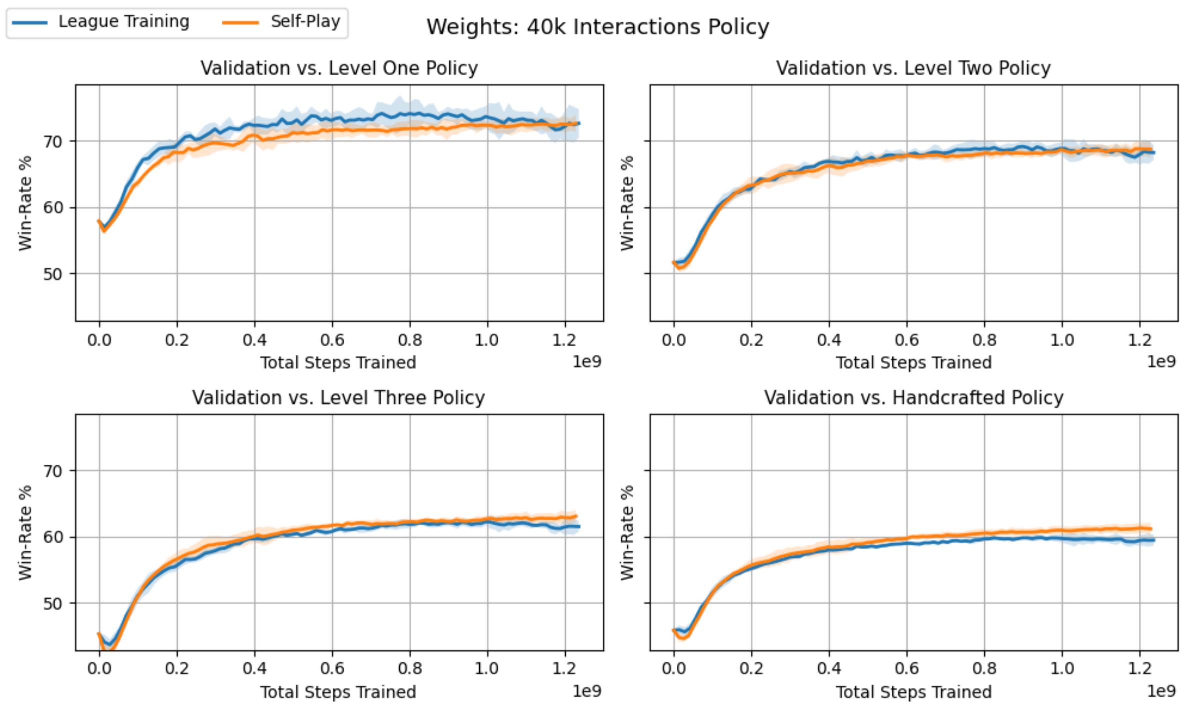

Figure 3.5.: Win-Rates vs. validation policies during training started with 40k policy.

using both self-play and league training methods.

40k Policy

The first IL weights were pre-trained with expert data containing 40,000 interactions. The performance graphs in Figure 3.5 show that both training methods experience a sharp increase in performance for about 150 million steps and then begin to plateau, although both performance curves maintain a slightly positive slope.

Upon closer examination of the performance of league-trained and self-playing agents against the level-one policy, league-trained agents outperform self-playing agents by a small margin up to about 1 billion steps, where their performance against the level-one policy converges. Against the level-two policy, both training methods produce agents with similar performance throughout the training process.

In contrast, against the level-three and handcrafted policies, league-trained and self-play-trained agents initially perform similarly, but self-play-trained agents gradually gain a slight advantage as training progresses. By the end of training, self-play-trained agents outperform league-trained agents by 2 to 3 percent against the most difficult level-three and handcrafted policies.

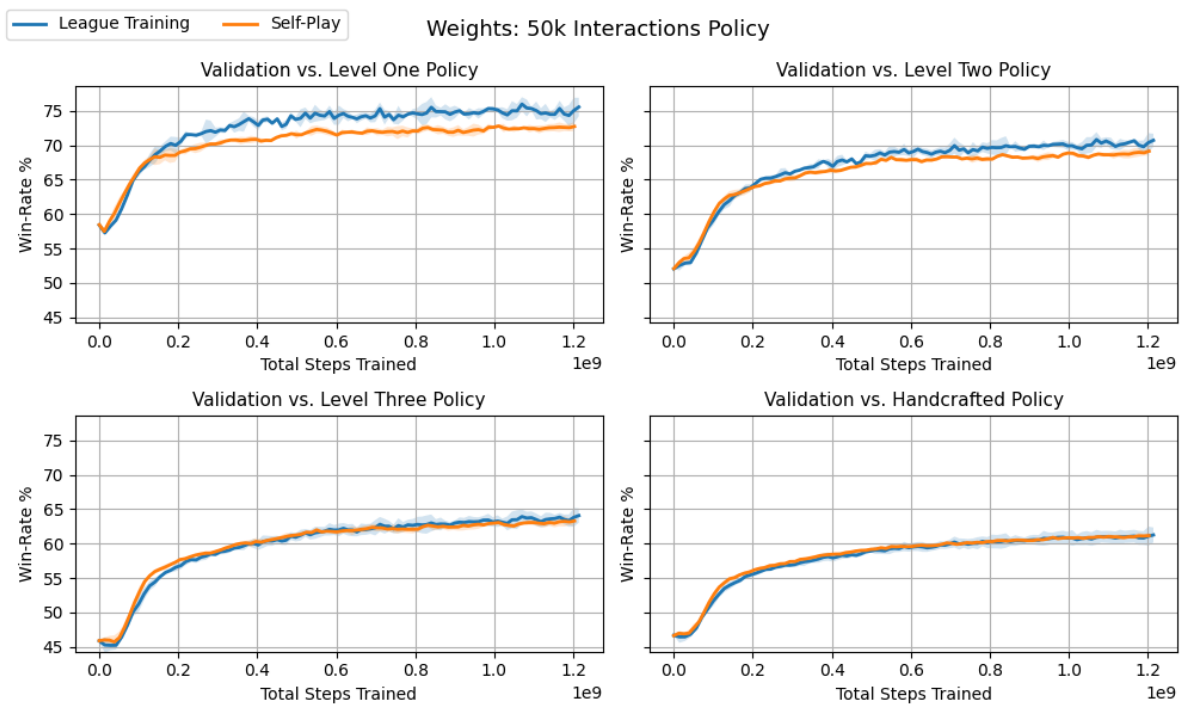

Figure 3.6.: Win-Rates vs. validation policies during training started with 50k policy.

50k Policy

The second IL weights were pre-trained with expert data containing 50,000 interactions. As observed in Figure 3.6, again both the self-play and league training methods yield a sharp increase in performance for about 150 million steps and then begin to plateau, although both maintain a slightly positive slope until the end of the training runs.

Upon closer inspection, the league-trained agents begin to outperform the self-play-trained agents by about 3 percent against the level-one policy after the initial steep performance gain phase. Against the level-two policy, the difference is smaller, but the league-trained agents still outperform the self-play-trained agents by about 2 percent. Similar to previous experiments, both training methods produce agents with comparable performance against the level-three and handcrafted policies.

200k Policy

The third IL weights were pre-trained with expert data containing 200,000 interactions. In Figure 3.7, again both training methods show a steep performance gain for about 150 million training steps. After that, the performance gain of the agents plateaus to a slightly positive slope until the end of the training.

Against the level-one policy, league-trained agents slightly outperform self-play-trained agents between about 300 million and 900 million trained steps. Before and after that, both

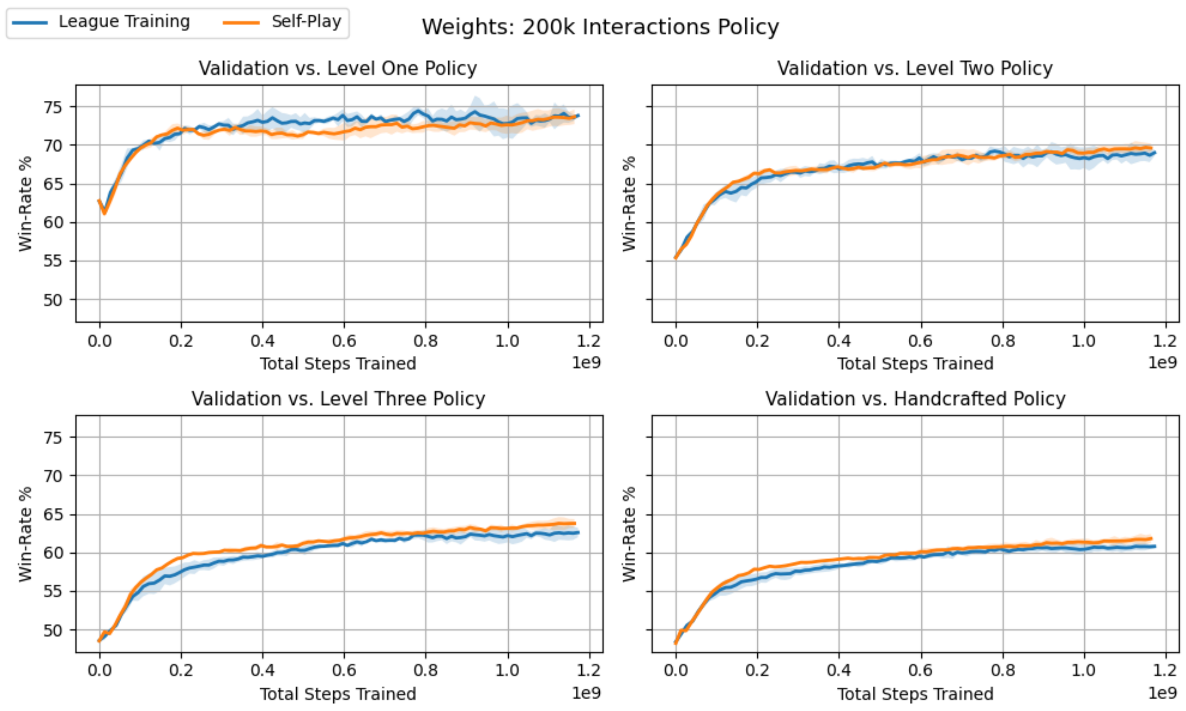

Figure 3.7.: Win-Rates vs. validation policies during training started with 200k policy.

training methods produce agents with similar performance. Against the level-two policy, agents trained by both methods perform comparably. Against the most demanding level-three and handcrafted policies, self-play-trained agents slightly outperform league-trained agents.

400k Policy

The fourth and final IL weights were pre-trained with expert data containing 400,000 interactions. In Figure 3.8, again both training methods show a steep performance gain for about 150 million training steps. After that, the performance gain of the agents plateaus to a slightly positive slope until the end of the training.

Looking at the individual performance graphs against specific validation policies in Figure 3.8, league-trained agents slightly outperform self-play-trained agents by across all policies. The difference in performance is more pronounced against the level-one and level-two policies and decreases against the level-three and handcrafted policies.

Performance Overview for Initialization with Imitation Learning Policies

Below is a summary of the performance results for both self-play and league training methods across all experiments where the agents were initialized with IL policies.

Against the level-one policy, the league-trained agents consistently outperform the self-play-trained agents slightly, especially in the middle stages of training. However, the performance

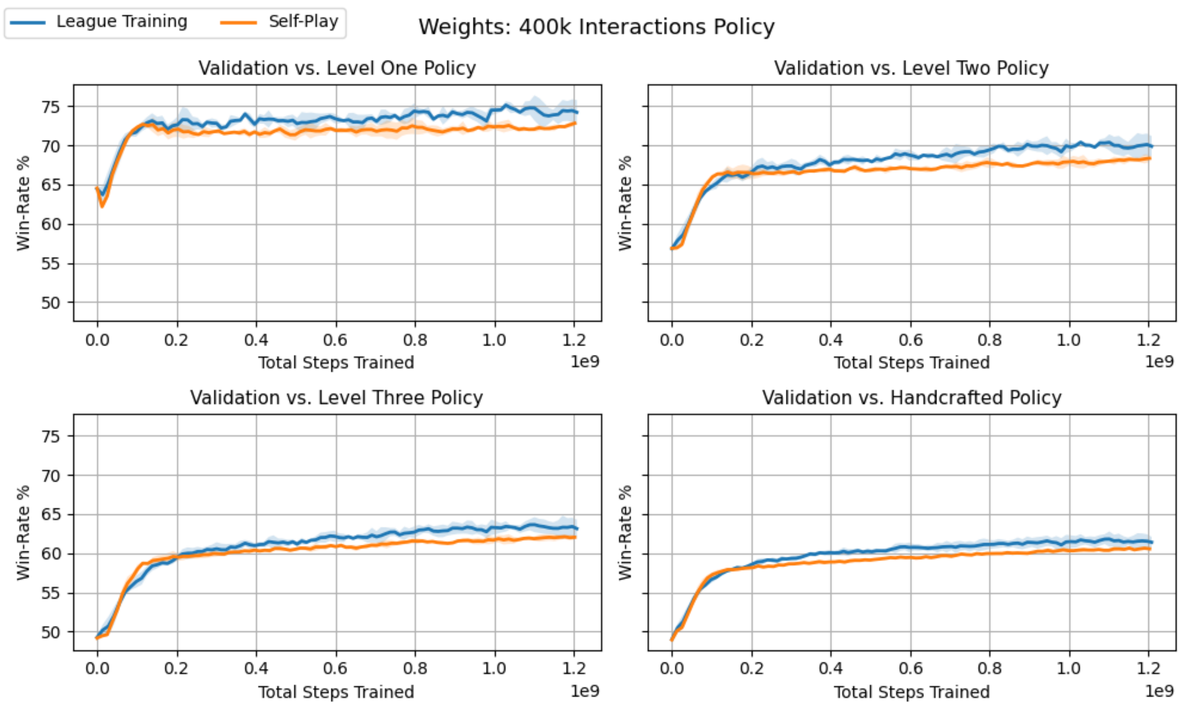

Figure 3.8.: Win-Rates vs. validation policies during training started with 400k policy.

of both sets of agents converges towards the end of the training process for the 40k and 200k policies. League-trained agents initialized with the 50k and 400k policies continue to slightly outperform their self-play counterparts after the early stages of training.

Against the level-two policy, the league-trained agents initialized with the 50k and 400k policies slightly outperformed the self-play-trained agents after the initial training period. However, the performance difference against the level-one policy was generally smaller, and in the 40k and 200k policy experiments, agents from both training methods performed comparably.

For the most difficult level-three and handcrafted policies, both training methods initially showed similar performance. However, the self-play-trained agents initialized with the 40k and 200k policies gradually gained a slight advantage as training progressed. The league-trained agents initialized with the 50k policy performed similarly to their self-play trained counterparts. Initializing league training with the 400k policy produced agents that slightly outperformed their self-play-trained counterparts against all validation policies.

In summary, the performance differences between self-play and league-trained agents varied depending on the amount of expert data used to initialize the IL policy. The agents initialized with the 40k and 200k policies performed very similarly, even though the 40k policies were trained with expert data that contained five times fewer interactions than the 200k policies. This means, that the amount of expert data used for IL, i.e. the behavioral proximity to the handcrafted policy, is not proportional to the performance of the agents during league training or self-play. Another interesting observation comes from comparing

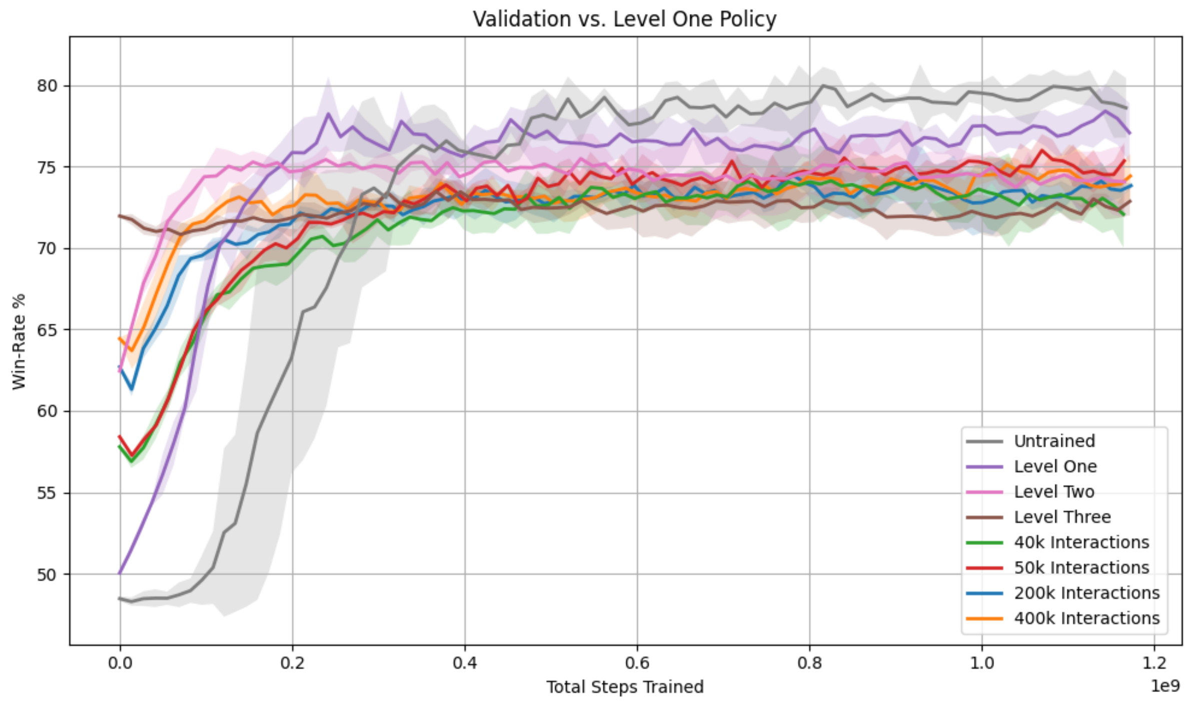

Figure 3.9.: Win-Rates vs. level-one policy during league training.

the graphs of the 40k and 50k policies. The 25% percent increase in expert data made a measurable difference in the performance of the league-trained agents, but not in the performance of the self-play-trained agents. In general, the agents initialized with the 50k and 400k policies responded better to league training than the agents initialized with the 40k and 200k policies.

3.2.4. Comparing Initialization Policies

In this section we will compare the initialization policies to each other. This will help point out the best suited and worst suited policies for initializing self-play or league training agents with. Performance is again measured in terms of win-rate against the validation policies.

League Training

In the league training scenario, the performance of agents initialized with different policies varies significantly. When playing against the level-one validation policy, the agents initialized with the untrained and level-one policies perform best, while the level-three and 40k policies yield the worst results.

When confronted with the level-two validation policy, the situation changes. The best performing agents are those initialized with the 50k and 400k policies, while the level-three and level-two policies are the least successful.

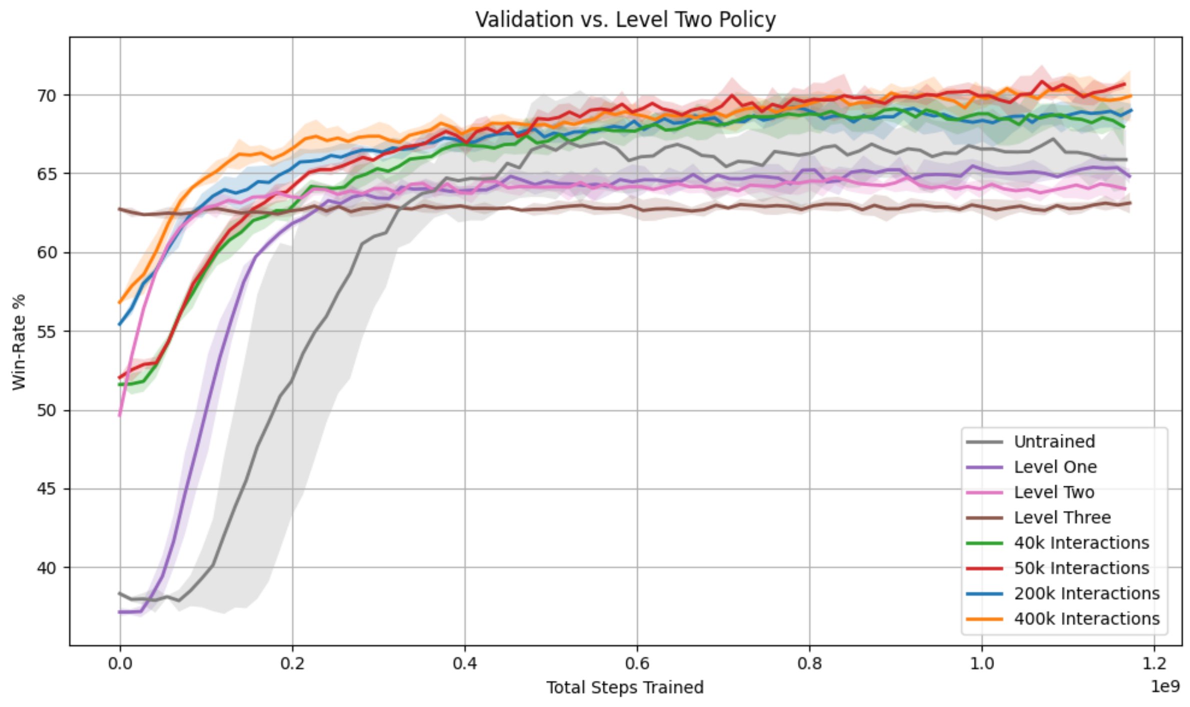

Figure 3.10.: Win-Rates vs. level-two policy during league training.

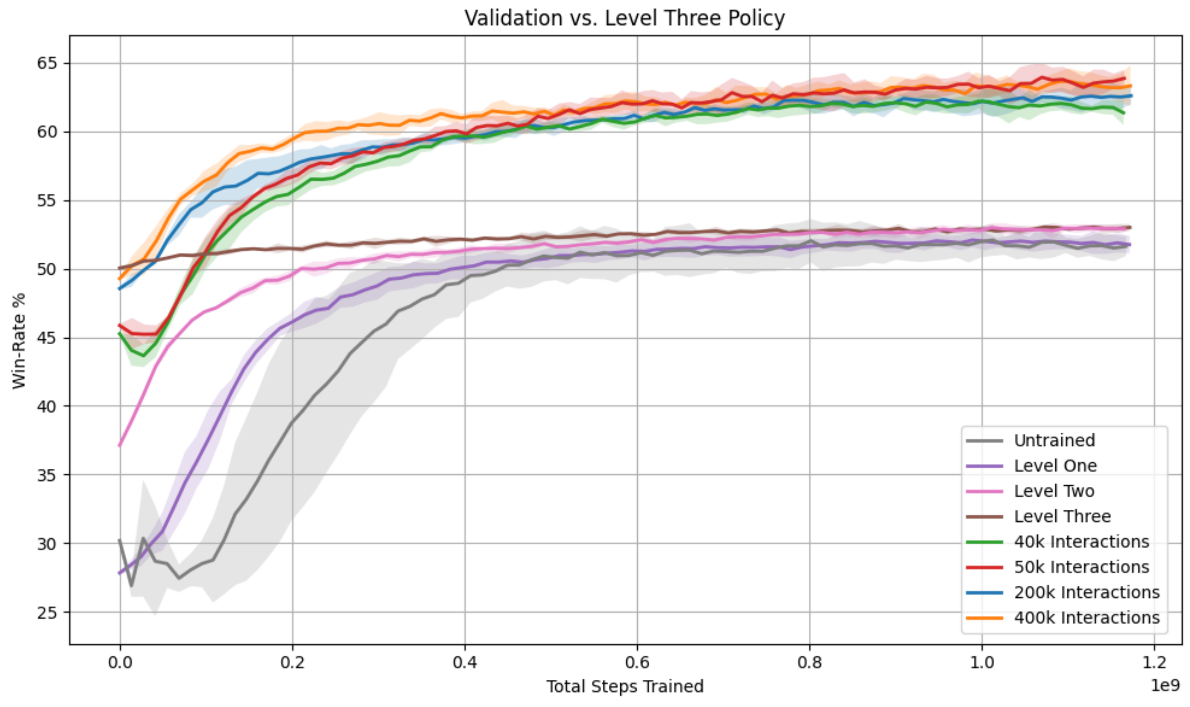

Figure 3.11.: Win-Rates vs. level-three policy during league training.

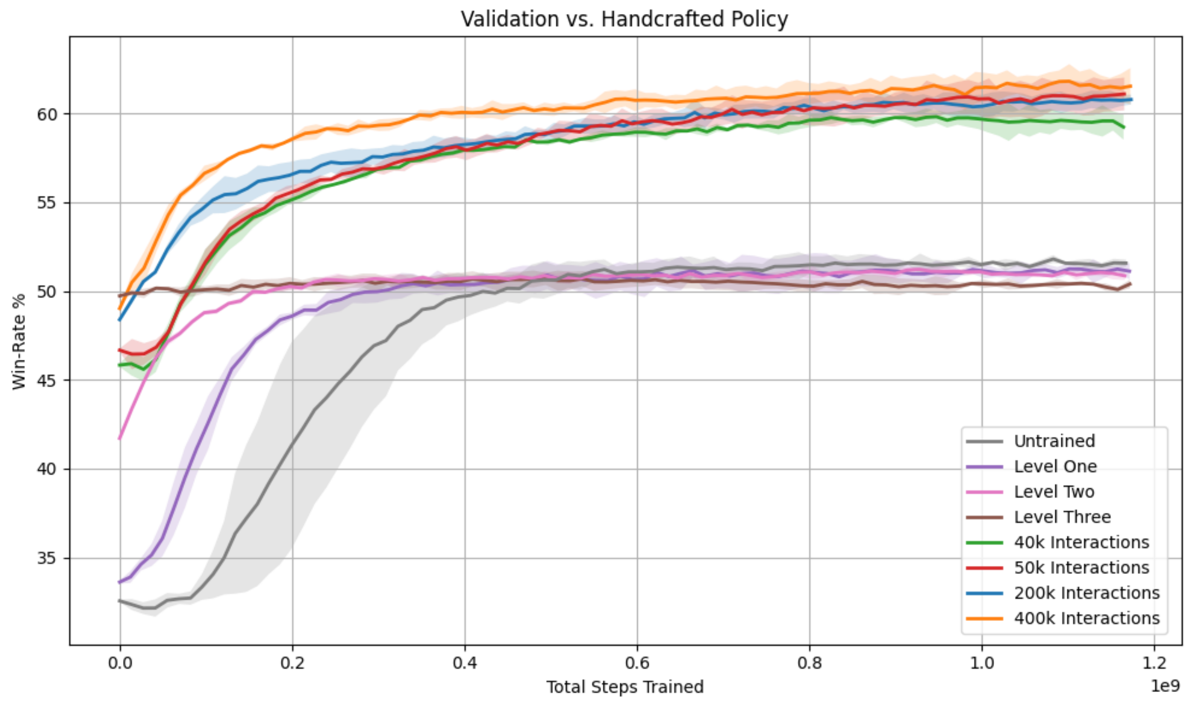

Figure 3.12.: Win-Rates vs. handcrafted policy during league training.

Against the level-three validation policy, there is a clear distinction in performance between two groups of agents. The better performing group consists of agents initialized with the IL (40k, 50k, 200k, 400k) policies. The lower performing group consists of agents initialized with the untrained and RL (level-one, level-two, level-three) policies.

Against the handcrafted validation policy, agent performance follows a pattern similar to that observed for the level-three validation policy. The better performing group includes agents initialized with the IL (40k, 50k, 200k, 400k) policies, while the worse performing group includes those initialized with the untrained and RL (level-one, level-two, level-three) policies.

Self-Play

In the self-play context, the agents behave differently than in league training. When playing against the level-one validation policy, the initial win-rates of agents initialized with different weights range from about 48% to 72%. However, by the end of the training, the difference in win-rates becomes negligible, falling within a range of 72% to 74%. Therefore, the choice of initial weights does not significantly affect the performance against the level-one validation policy.

When faced with the level-two validation policy, agent win-rates can be divided into two performance groups. The better performing group consists of agents initialized with the IL (40k, 50k, 200k, 400k) policies, while the worse performing group consists of agents initialized

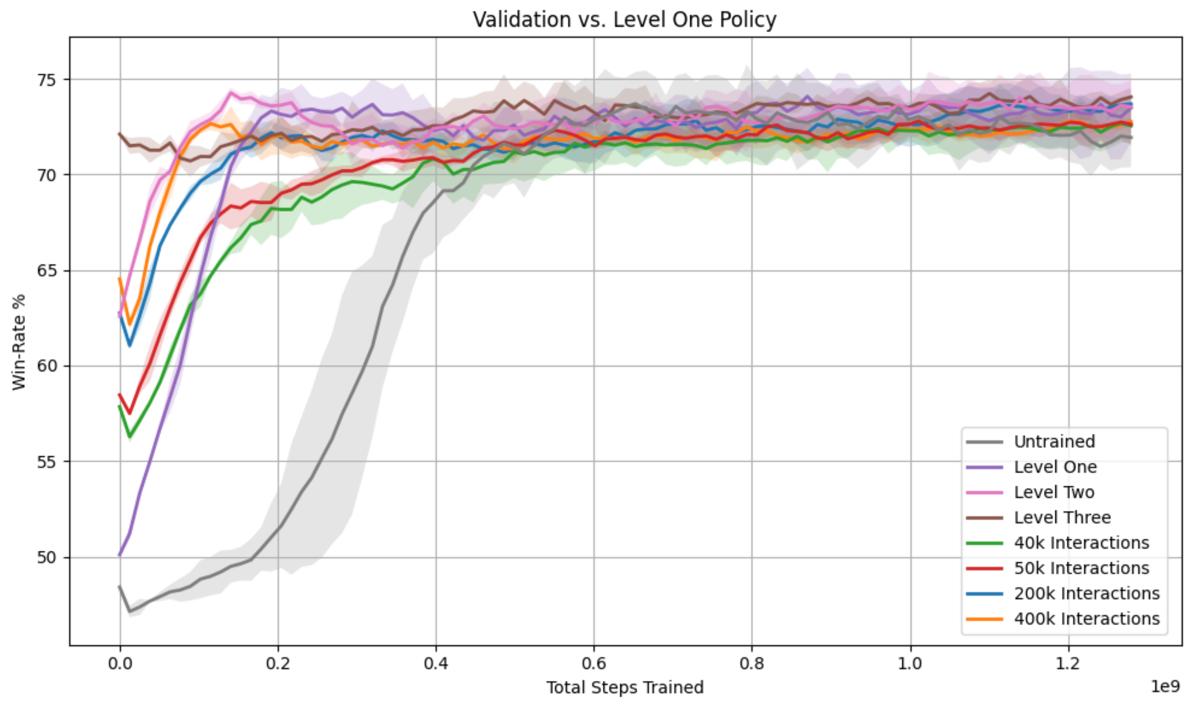

Figure 3.13.: Win-Rates vs. level-one policy during self-play.

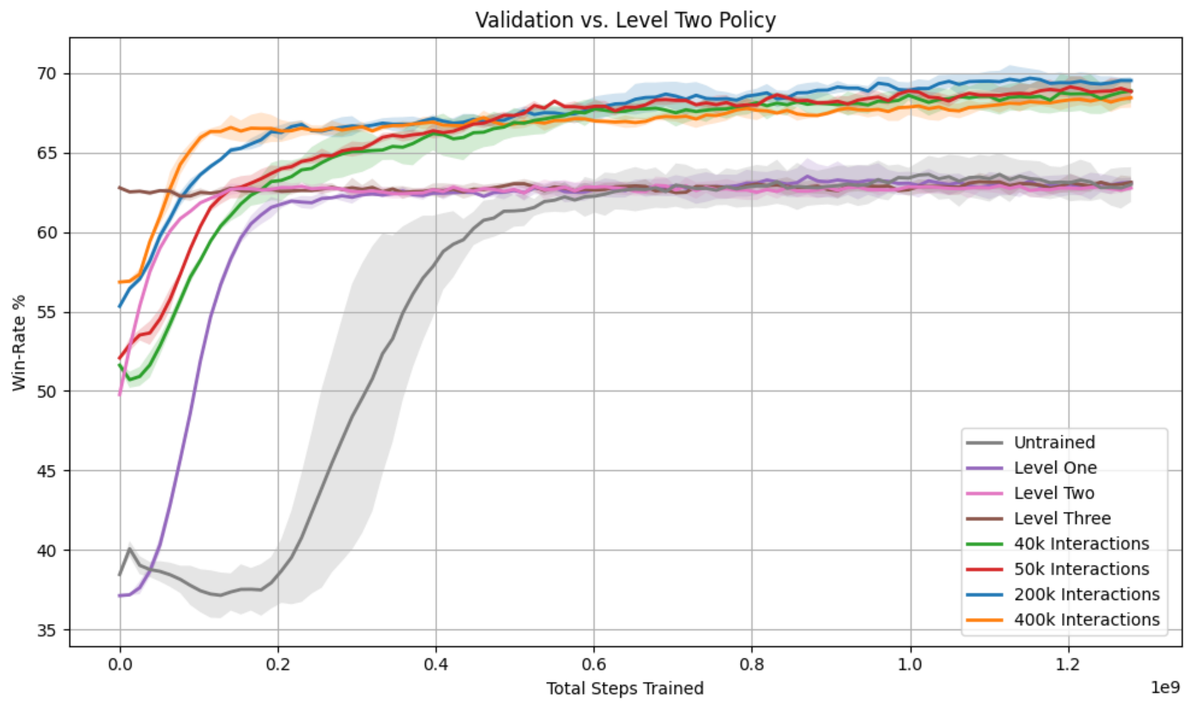

Figure 3.14.: Win-Rates vs. level-two policy during self-play.

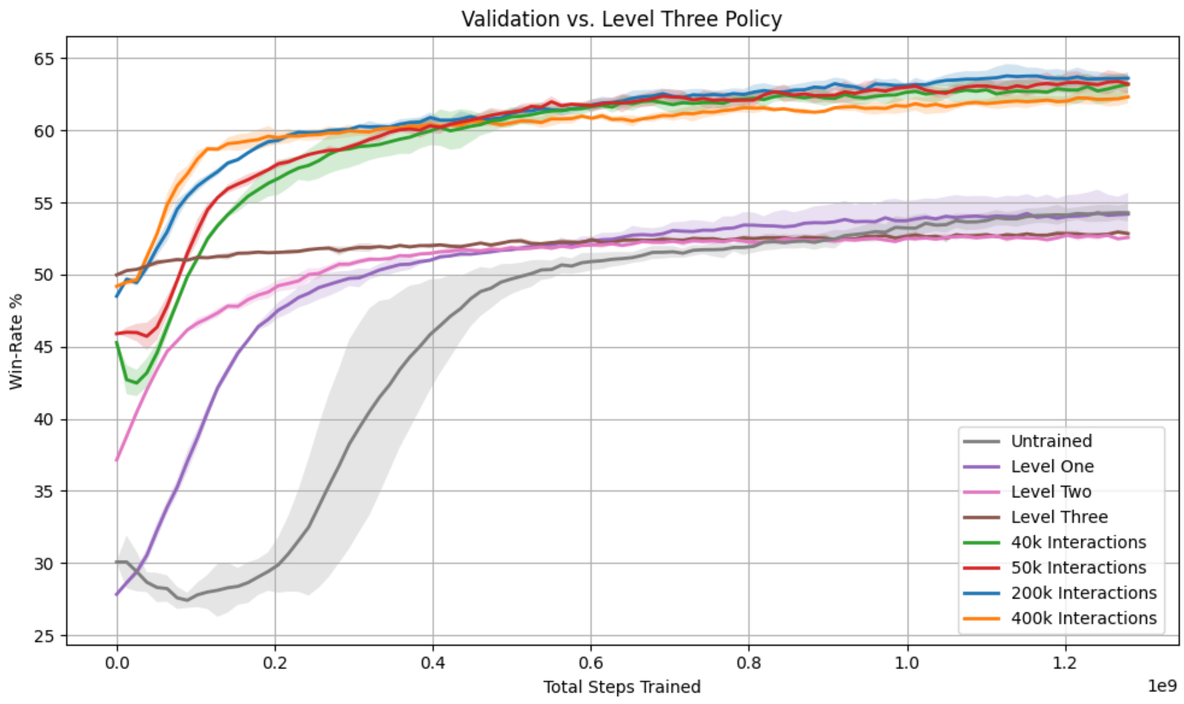

Figure 3.15.: Win-Rates vs. level-three policy during self-play.

with the untrained and RL (level-one, level-two, level-three) policies.

Against the level-three validation policy, a similar pattern emerges. The better performing group consists of agents initialized with the IL (40k, 50k, 200k, 400k) policies, while the worse performing group consists of agents initialized with the untrained and RL (level-one, level-two, level-three) policies.

Finally, when competing against the handcrafted validation policy, agent win-rates can also be divided into two performance groups. The better performing group consists of agents initialized with the IL (40k, 50k, 200k, 400k) policies, while the worse performing group consists of agents initialized with the untrained and RL (level-one, level-two, level-three) policies.

Synthesis of Comparing Initialization Policies

The results of comparing different initialization policies provide interesting insights into the dynamics of agent performance in league training and self-play scenarios.

In league training, the initialization policy has a significant impact on agent performance. When faced with the level-one validation policy, the untrained and level-one initialized agents perform best, while agents initialized with the level-three and 40k policies underperform. As the complexity of the validation policy increases to level-two, the best performers shift to agents initialized with the 50k and 400k policies, while the level-three and level-two initialized agents produce the worst results. Against the level-three and handcrafted validation policies,

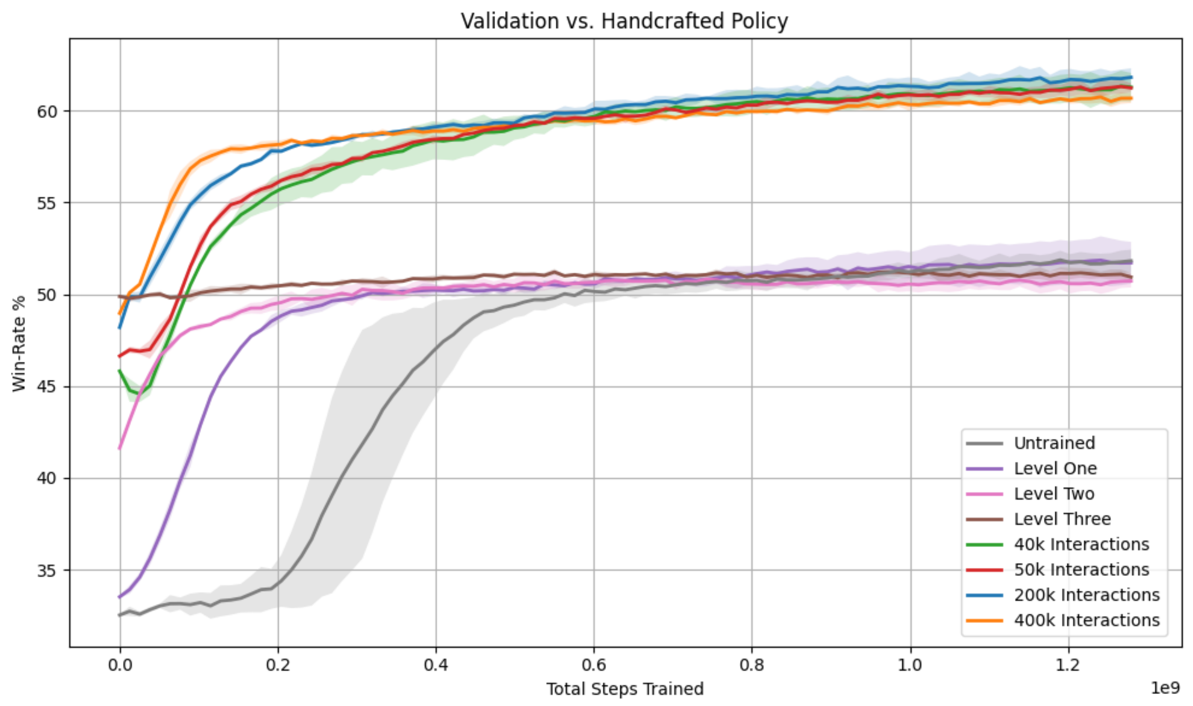

Figure 3.16.: Win-Rates vs. handcrafted policy during self-play.

agents initialized with the IL policies (40k, 50k, 200k, 400k) consistently outperform those initialized with the untrained and RL policies (level-one, level-two, level-three).

Conversely, in self-play, the initial weight choice does not significantly affect the performance against the level-one validation policy. After training, the win-rates of all agents converge to a similar range. However, against higher-level (level-two, level-three) and handcrafted validation policies, the superior performers are those initialized with IL policies. In these cases, the underperforming group consistently consists of agents initialized with the untrained and RL policies.

These results suggest that the policy choice during initialization can significantly affect agent performance, especially in league training scenarios and when faced with more complex validation policies. The IL policies appear to be the most effective initializations for the highest level of performance.

4. Conclusion

This thesis investigated the application of league training to new MARL environments, such as SimpleSoccer environment. We compared the effectiveness of league training with self-play training by extending the existing code base and setting up a league training environment. Agents were initialized with different RL and IL policies, and different validation policies were used as benchmarks for performance evaluation. The primary performance metric considered was the win-rate of the agent against the respective validation agents.

4.1. Summary of Results

Comparing League Training to Self-Play

This thesis presents a series of experiments designed to analyze league-training in MARL applications. The experiments were conducted over three different policy initialization techniques: untrained, RL, and IL policies. The performance of league-trained agents was compared to self-play-trained agents on three different levels of validation policies and a handcrafted policy.

In the experiments initialized with untrained policies, league-trained agents outperformed self-play-trained agents against the less trained validation policies. Interestingly, self-play-trained agents eventually caught up to, and even slightly outperformed, league-trained agents against the most difficult level-three and handcrafted validation policies.

For the experiments initialized with RL policies, league-trained agents generally performed better against level-one and level-two policies, but the differences became less pronounced with higher-level initialization policies. Both league-trained and self-play-trained agents performed similarly against level-three and handcrafted validation policies, regardless of the initialization policy.

The experiments initialized with IL policies showed varying performance differences between league-trained and self-play-trained agents, depending on the amount of expert data used for initialization. In general, agents initialized with the 50k and 400k policies responded more favorably to league training than those initialized with the 40k and 200k policies. The performance gains were not proportional to the amount of expert data used for IL, suggesting that other factors may be at play. In addition, some increase in expert data made a measurable difference in the performance of the league-trained agents, but not in the self-play-trained agents.

Overall, the experiments demonstrate the complexity and nuance of MARL applications. League training showed robust performance across different initialization techniques, often exceeding or matching the performance of self-play training. However, the results also underscored that the effectiveness of training methods can vary depending on the specifics of the initialization policy, suggesting a need for further exploration and fine-tuning of these techniques.

Comparing Initialization Policies

Building on these findings, the subsequent analysis of different initialization policies provided further insight into the nuanced dynamics of agent performance in league training and self-play scenarios.

In league training, the initialization policy had a significant impact on agent performance. Against the level-one validation policy, the untrained and level-one initialized agents excelled, while agents initialized with the level-three and 40k policies lagged behind. With the increased complexity of the level-two validation policy, the top performers shifted to agents initialized with the 50k and 400k policies. In contrast, the level-three and level-two initialized agents produced the worst results. When faced with the highly demanding level-three and handcrafted validation policies, agents initialized with the IL policies (40k, 50k, 200k, 400k) consistently outperformed those initialized with the untrained and RL policies (level-one, level-two, level-three).

In contrast, within the self-play scenarios, the initial weight choice did not significantly affect the performance against the level-one validation policy. All agents' win-rates converged to a similar range after the training. However, when competing against the more advanced level-two, level-three, and handcrafted validation policies, the top performers were consistently those initialized with IL policies. The underperforming group was invariably composed of agents initialized with the untrained and RL policies.

Overall, these results highlight the significant impact of policy choice during initialization on agent performance, especially in the context of league training and when faced with more stringent validation policies. The IL policies emerged as the most effective initializations for achieving the highest level of performance, underscoring their potential for enhancing the effectiveness of league training in MARL applications.

4.2. Discussion

Our results suggest that the performance difference between league training and self-play in our setup is not substantial enough to justify the significant computational cost associated with league training. League training requires the training of a set of four agents with specific goals, while self-play only requires the training of a single agent.

Several hypotheses can be proposed to explain the observed lack of significant performance differences between self-play and league training in the SimpleSoccer environment. The first possible cause could be that the SimpleSoccer game is not be complex enough to take full advantage of league training. The second possible cause could be that the initialization method used in AlphaStar, which uses SL from human expert data, is significantly different from the IL policy initialization used in this thesis. Another possible explanation could be that the use of the statistics z, which helped agents explore different play styles and strategies more effectively in AlphaStar, was absent in our setup.

4.3. Future Research Directions

Based on the insights gained from our experiments, we suggest several research directions to further explore the potential benefits of league training in MARL environments, such as the SimpleSoccer environment. One possible approach is to use the initial SL policy to condition the agent's actions during training, incorporating expert knowledge to guide agents to adopt diverse and effective strategies. In addition, developing a more sophisticated handcrafted policy that incorporates complex strategies, such as passing, positioning, and coordinated team play, could provide a richer source of expert data for initializing agents with IL. By exposing agents to more advanced strategies during the initialization phase, their subsequent training could be more effective, leading to improved performance against a broader range of opponents. In addition, modification of the reward structure through more sophisticated intermediate rewards or the introduction of pseudo-rewards (along the lines of AlphaStar's league training method) can be explored. Providing additional rewards that encourage agents to adopt complex strategies or specific behaviors may encourage them to explore a wider variety of strategies during training, resulting in the development of more adaptive and competitive agents that can better handle a diverse set of opponents. Some of these modifications may not only improve agent performance in league training, but also in self-play.

A. General Addenda

A.1. League Training Components

This appendix section discusses the components implemented for a league training setup. Some components are loosely based on the pseudocode in the supplementary data of the original AlphaStar paper, while others are built from scratch following the descriptions in the original paper [12]. The number of agents used to initialize the league was also adapted from the original paper. They used one main agent, one main exploiter, and two league exploiters per race. Since we do not have three different races, we use a total of one main agent, one main exploiter, and two league exploiters. At the end of this section is a diagram of our league training implementation.

A.1.1. League

The League class acts as a system for managing different types of agents within the competitive league environment. It initializes with a specified number of main agents, main exploiters, and league exploiters, and includes various configuration options such as checkpoint thresholds, agent log paths, and validation agent weights. The class uses the Payoff module to store and update win rates between agents. By providing methods for tracking agents' opponents, obtaining win-rate matrices, and managing learning agents, the class provides a framework for simulating competition and tracking agent performance over time. In addition, the class supports saving and loading of agent weights.

A.1.2. Player

The Player class represents an individual agent in the league. It stores the agent's properties, such as its weights, total trained steps, and checkpoint thresholds. The class provides methods for tracking the agent's progress, managing its training opponents, and determining whether it is ready for a checkpoint based on performance criteria or steps trained. The Player class serves as the parent class for specialized league training agent types, including MainPlayer, LeagueExploiter, MainExploiter, and Historical.

A.1.3. Historical

The Historical class is a specialized subclass of the Player class that represents a checkpoint of a player at a particular point in time during league training. This class is initialized with a parent player and inherits the strategy of the parent in the form of its weights and the total number of steps trained. A unique name is generated for the Historical player to indicate its parent and the number of training steps at the time of creation. The Historical class ensures that its weights remain frozen during matches, since it is not intended to actively participate in the learning process. It also overrides methods that prevent historical players from requesting matches or being considered for checkpoints, reflecting their static nature in league training.

A.1.4. MainPlayer

The MainPlayer class is a subclass of the Player class and represents the main agents in league training. The MainPlayer class has three specific functions to determine its match opponents and a primary "get_match" method that decides which of these functions to choose based on a random "coin toss". It chooses the PFSP branch 50% of the time, and the verification branch 15% of the time. For the remaining 35% or if the verification branch returns no player, the self-play branch is used.

In the PFSP branch method, the player samples an opponent from the historical players using PFSP with squared weighting. The self-play branch method is responsible for either choosing self-play or sampling a curriculum opponent from the historical players using PFSP with variance weighting. The verification branch method checks for strong historical exploiters and forgotten historical players, using PFSP with squared weighting, and samples from them, if present.

A.1.5. LeagueTrainer

The LeagueTrainer class serves as an orchestrator for training and evaluating the performance of agents within a league. Upon initialization, it takes a configuration object, an environment, and the league instance. It manages various training and evaluation processes, such as validating the league and comparing the performance of the main agent with validation agents. The main method plays a specified number of matches, iterates through the learning agents, and initiates their training through the TrainingMatchCoordinator. The class uses the PayoffUpdater module to update the win-rate matrix after each match, ensuring that agent performance is accurately tracked throughout the training process. In addition, the LeagueTrainer class handles checkpointing, creating historical agents as needed. Overall, the LeagueTrainer class manages the training and evaluation of agents, ensuring that agent interactions and learning are coordinated.

A.1.6. PayoffUpdater

The PayoffUpdater is responsible for updating the payoff matrix of agents in a league. It takes a configuration, an evaluation match environment, and a league as input when initialized. It contains several methods, i.e., one method updates the payoff matrix for a given agent by playing evaluation matches against other agents in the league if the evaluation is necessary. An evaluation is only necessary against agents that are potential training partners. Another method logs the results of each evaluation run. Evaluations are crucial for evaluating the agent's performance and finding the best opponent.

A.1.7. TrainingMatchCoordinator

The TrainingMatchCoordinator class is responsible for coordinating training matches between agents in a league. It is initialized with a configuration, an environment, an agent, and a match ID.

This class has a method to play the match between the two agents. It sets up the match using another method that prepares the agents' learners by enabling or disabling their learning, setting their weights, and updating their TensorBoard paths and steps. The match is then played, and the method returns the agents, match rewards, and match lengths. After the match is played, a method is called to finish the match, which checks if the agents' weights are valid after the match and updates the agents' weights and trained steps if they are not historical agents.

Several helper methods are used to set up the agents' learners for a training match. They handle tasks such as setting weights, enabling or disabling learning, updating TensorBoard paths, and configuring logger settings.

A.1.8. EvaluationMatchCoordinator

The EvaluationMatchCoordinator class is responsible for coordinating evaluation matches between agents in a league. It is initialized with a configuration, an environment, the agents to be evaluated, and the current evaluation run. The main method plays a match between the two agents. Several helper methods are used to prepare the learners of the agents for an evaluation match. They handle tasks such as setting weights, disabling learning, updating TensorBoard paths, and configuring logger settings. The match is then played, and the method returns match rewards and match lengths for the evaluation.

A.1.9. LeagueValidator

The LeagueValidator class is responsible for evaluating the performance of the main agents. It initializes with a configuration object, an environment, and the league instance. The class provides a method to validate the league, calculate the performance of the validation agents, and compare the performance of the main agent to them. Its main method periodically validates performance of the main agent by playing validation matches against validation agents and logging the results. Another method creates a matrix of win rates between validation agents to establish performance thresholds to be met. Finally, another method compares the performance of the main agent against validation agents and determines whether the main agent has outperformed the validation agents. The class provides tools to evaluate the performance of agents within a league, helping to determine their progress relative to validation agents.

A.1.10. ValidationMatchCoordinator

The ValidationMatchCoordinator class is responsible for coordinating validation matches between learning agents and validation agents in a league. The class is initialized with a configuration, a validation environment, a learning agent to be validated, the current validation run number, and additional parameters related to the validation agents.

The main method of this class plays a validation match between the learning agent and the validation agent. The setup method prepares the learning agent and validation agent by disabling their learning, setting their weights, and updating their log paths and trained steps. The match is then played, and the match statistics are logged and returned.

A.1.11. PFSP

The PFSP function calculates the selection probability distribution for agents based on their win rates and the specified weighting method. The function applies the weighting function to the win-rates given to it. After calculating the probabilities, it normalizes them by dividing each probability by the sum of all probabilities. If the sum of the probabilities is very small (less than 1e-10), the function returns an array of equal probabilities for each agent. This ensures a uniform distribution of probabilities in cases where the calculated probabilities are very close to zero, preventing any bias in the selection process.

In conclusion, this appendix chapter has provided a detailed overview of the various components implemented to enable league training in the existing codebase. These components were designed based on the AlphaStar paper and adapted to the specific requirements of the project.

B. Figures

B.1. League Training UML diagram

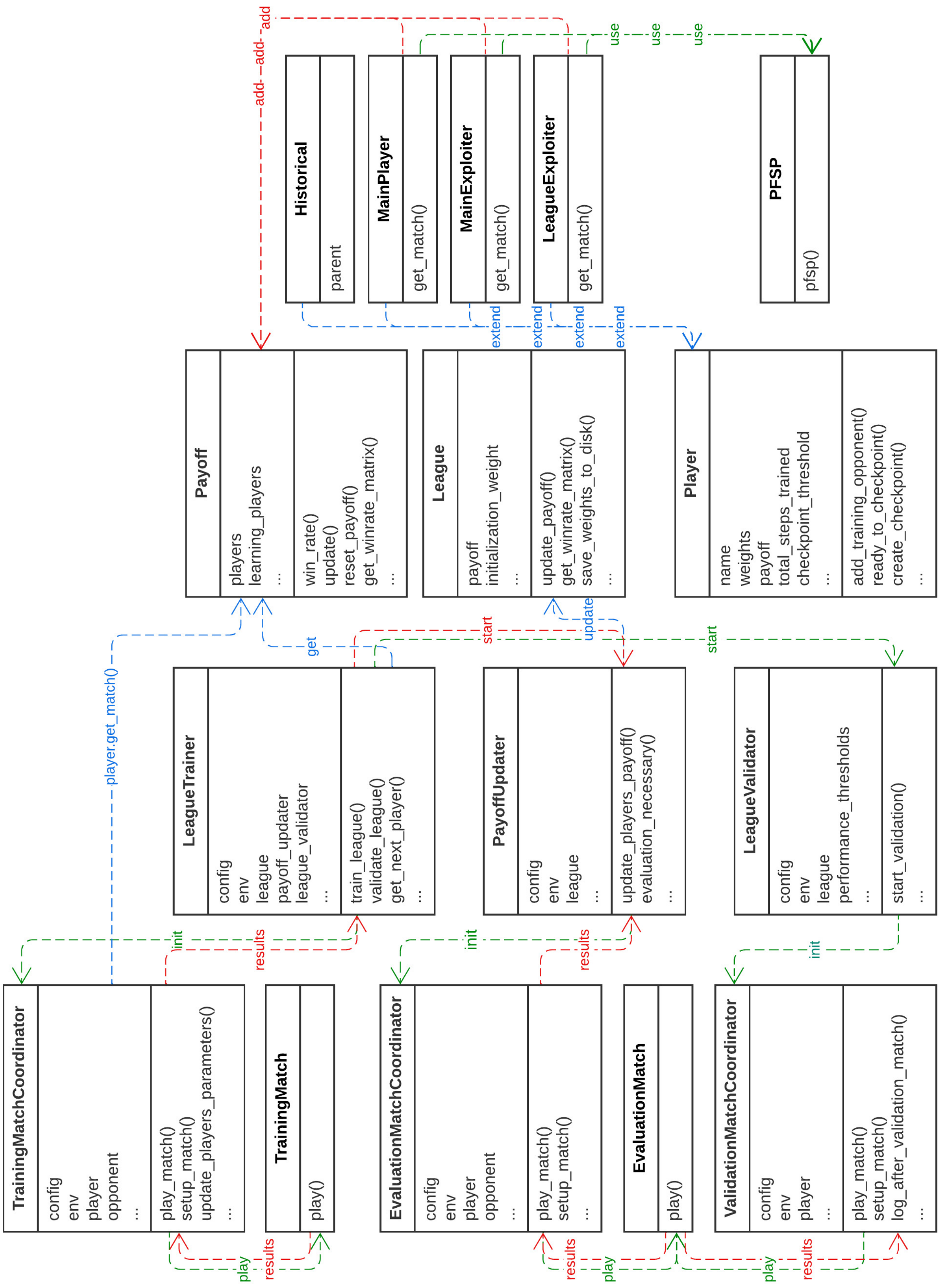

Figure B.1.: UML diagram of the implemented League Training setup.

List of Figures

Acronyms

Bibliography

[1] P. Hernandez-Leal, B. Kartal, and M. E. Taylor. "A Survey and Critique of Multiagent Deep Reinforcement Learning". In: (2018). DOI: 10.48550/arXiv.1810.05587.

[2] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, and J. Chung. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* 575.7782 (2019), pp. 350–354. DOI: 10.1038/s41586-019-1724-z. URL: https://www.nature.com/articles/s41586-019-1724-z.

[3] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[4] V. Konda and J. Tsitsiklis. "Actor-Critic Algorithms". In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[5] I. Grondman, L. Busoniu, G. A. D. Lopes, and R. Babuska. "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1291–1307. DOI: 10.1109/TSMCC.2012.2218595.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/arXiv.1707.06347.

[7] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. "A survey of robot learning from demonstration". In: *Robotics and Autonomous Systems* 57.5 (2009), pp. 469–483. ISSN: 09218890. DOI: 10.1016/j.robot.2008.10.024.

[8] F. Torabi, G. Warnell, and P. Stone. *Behavioral Cloning from Observation*. 2018. DOI: 10.48550/arXiv.1805.01954.

[9] Stable Baselines Team. *Stable Baselines3 - Pretraining with Behavior Cloning*. URL: https://colab.research.google.com/github/Stable-Baselines-Team/rl-colab-notebooks/blob/sb3/pretraining.ipynb#scrollTo=ShofuYy-8dLs.

[10] Alexander Neitzal. *SimpleSoccer*. 2022. URL: https://github.com/neitzal/simplesoccer.

[11] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[12] Google Deepmind. *Grandmaster level in StarCraft II using multi-agent reinforcement learning: Supplementary Data*. URL: `https://static-content.springer.com/esm/art%5C%3A10.1038%5C%2Fs41586-019-1724-z/MediaObjects/41586_2019_1724_MOESM2_ESM.zip`.