Scaling multi-agent reinforcement learning to eleven aside simulated robot soccer

by

Andries Petrus Smit

Dissertation presented for the degree of

Doctor of Philosophy in Electrical and Electronic Engineering

in the Faculty of Engineering at Stellenbosch University

Promoter: Prof. H.A. Engelbrecht

Co-promoters: Prof. W. Brink, Dr A. Pretorius

December 2022

Plagiaatverklaring / *Plagiarism Declaration*

(updated March 2021)

1 Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
Plagiarism is using ideas, material and other intellectual property of another's work and presenting it as my own.

2 Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
I agree that plagiarism is a punishable offence because it constitutes theft.

3 Ek verstaan ook dat direkte vertalings plagiaat is.
I also understand that direct translations are plagiarism.

4 Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
Accordingly, all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.

5 Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel, of gedeeltelik, ingehandig het om enige kwaifikasie te verwerf nie.
I declare that the work contained in this thesis/dissertation, except where otherwise stated, is my original work, and that I have not previously in its entirety, or in part, submitted this thesis/dissertation to obtain any qualification.

Voorletters en van / *Initials and surname:* A.P. Smit

Datum / *Date:* 10 August 2022

Abstract

Robot soccer, where teams of autonomous agents compete against each other, has long been regarded as a grand challenge in artificial intelligence. Despite recent successes of learned policies over heuristics and handcrafted rules in other domains, current teams in the RoboCup soccer simulation leagues still rely on handcrafted strategies and apply reinforcement learning only on small subcomponents. This limits a learning agent's ability to find strong, high-level strategies for the game in its entirety. End-to-end reinforcement learning has successfully been applied in soccer simulations with up to 4 players. However, little previous work has been done on training in settings with more than 4 players, as learning is often unstable as well as it taking much longer to learn basic strategies. In this dissertation, we investigate whether it is possible for agents to learn competent soccer strategies in a full 22 player soccer game using limited computational resources (one CPU and one GPU), from tabula rasa and entirely through self-play. To enable this investigation, we build a simplified 2D soccer simulator with significantly faster simulation times than the official RoboCup simulator, that still contains the important challenges for multi-agent learning in the context of robot soccer. We propose various improvements to the standard single-agent proximal policy optimisation algorithm, in an effort to scale it to our multi-agent setting. These improvements include (1) using a policy and critic network with an attention mechanism that scales linearly in the number of agents, (2) sharing networks between agents which allow for faster throughput using batching, and (3) using Polyak averaged opponents with freezing of the opponent team when necessary and league opponents. We show through experimental results that stable training in the full 22 player setting is possible. Agents trained in the 22 player setting learn to defeat a variety of handcrafted strategies, and also achieve a higher win rate compared to agents trained in the 4 player setting and evaluated in the full 22 player setting. We also evaluate our final algorithm in the RoboCup simulator and observe steady improvement in the team's performance over the course of training. Our work can guide future end-to-end multi-agent reinforcement learning teams to compete against the best handcrafted strategies available in simulated robot soccer.

Opsomming

Robotsokker, waar spanne van outonome agente teen mekaar meeding, word lank reeds as 'n groot uitdaging vir kunsmatige intelligensie beskou. Ten spyte van onlangse suksesse van aangeleerde beleide teenoor heuristieke en handgemaakte reëls in ander domeine, maak huidige spanne in die RoboCup-sokkersimulasie-ligas steeds staat op handgemaakte strategieë en pas versterkingsleer slegs toe op klein subkomponente. Dit beperk 'n leeragent se vermoë om sterk, hoëvlakstrategieë vir die spel in sy geheel te vind. Eind-tot-eind versterkingsleer is al suksesvol toegepas in sokkersimulasies met tot 4 spelers. Min vorige werk is egter gedoen aan afrigting in opstellings met meer as 4 spelers, aangesien leer dikwels onstabiel is, asook dat dit baie langer neem om basiese strategieë aan te leer. In hierdie proefskrif ondersoek ons of dit moontlik is vir agente om bekwame sokkerstrategieë in 'n volle 22-speler sokkerwedstryd aan te leer deur gebruik te maak van beperkte rekenaarhulpbronne (een SVE en een GVE), vanaf tabula rasa en net deur selfspeel. Om hierdie ondersoek moontlik te maak, bou ons 'n vereenvoudigde 2D-sokkersimulator met aansienlik vinniger simulasietye as die amptelike RoboCup-simulator, wat steeds die belangrike uitdagings vir multi-agent-leer in die konteks van robotsokker bevat. Ons stel verskeie verbeterings aan die standaard enkel-agent proksimale beleid optimeringsalgoritme voor, in 'n poging om dit te skaal na ons multi-agent opstelling. Hierdie verbeterings sluit in (1) die gebruik van 'n beleid- en kritikusnetwerk met 'n aandagmeganisme wat lineêr in die aantal agente skaal, (2) die deel van netwerke tussen agente wat vinniger berekeninge moontlik maak deur gebruik te maak van groepering, en (3) die gebruik van Polyak-gemiddelde teenstanders met die vries van die teenstanderspan wanneer nodig en liga-teenstanders. Ons wys deur eksperimentele resultate dat stabiele afrigting in die volle 22-speler-opstelling moontlik is. Agente wat in die 22-speler-opstelling afgerig word, leer om 'n verskeidenheid handgemaakte strategieë te verslaan, en behaal ook 'n hoër wenkoers in vergelyking met agente wat in die 4-speler-opstelling afgerig word en in die volle 22-speler-opstelling geëvalueer word. Ons evalueer ook ons finale algoritme in die RoboCup-simulator en neem deur die loop van afrigting konstante verbetering in die span se wenkoers waar. Ons werk kan toekomstige eind-tot-eind multi-agent versterkingsleer spanne lei om mee te ding teen die beste handgemaakte strategieë wat beskikbaar is in gesimuleerde robotsokker.

Acknowledgements

I would like to thank my supervisors, Prof. Herman Engelbrecht, Prof. Willie Brink and Dr Arnu Pretorius for their valuable feedback and guidance throughout this project's development. I would like to thank Dr Arnu Pretorius in particular for allowing me to work on this project while also taking part in a two year internship at InstaDeep. Lastly, I would like to acknowledge and thank Prof. Herman Engelbrecht and InstaDeep for their generosity in providing the necessary funding and computational resources for this project.

Contents

List of figures

List of tables

List of algorithms

Nomenclature

Abbreviations

A2C	Advantage Actor-Critic
ADR	Automatic Domain Randomisation
AI	Artificial Intelligence
Dec-POMDP	Decentralised Partially Observable Markov Decision Process
HBEC	Human Based Evolutionary Computation
JIT	Just-In-Time
MA-D4PG	Multi-Agent Distributed Distributional Deep Deterministic Policy Gradient
MA-DDPG	Multi-Agent Deep Deterministic Policy Gradient
MA-PPO	Multi-Agent Proximal Policy Optimisation
MARL	Multi-Agent Reinforcement Learning
MDP	Markov Decision Process
PBT	Population Based Training
POMDP	Partially Observable Markov Decision Process
PPO	Proximal Policy Optimisation
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SVM	Support Vector Machine
TPU	Tensor Processing Unit
XLA	Accelerated Linear Algebra

Greek symbols

α	A user defined constant that specifies how fast a neural network's parameters should be updated.
$\boldsymbol{\beta}_j^k$	The best response policy of agent k for iteration j, that best exploits the other agents.
$\boldsymbol{\pi}_j^k$	The average strategy policy of agent k for iteration j.
$\boldsymbol{\pi}_j^{-k}$	The average strategy policy of every agent except agent k for iteration j.
θ^*	The optimal setting for θ that maximises an objective function $J(\theta)$.
ϵ	A threshold value that determines by what factor an output action's probability may change.
η	The sequence $a_t, s_{t+1}, a_{t+1}, \ldots, s_T, a_T$.
γ	The discount factor used to shape an agent's reward collecting behaviour.
π	The current policy of an agent.
$\pi_{\theta\text{old}}$	The output of the old policy network.
τ	The sequence $s_0, a_0, \ldots, s_T, a_T$.

Roman symbols

$\boldsymbol{K}$	The key matrix used in the multi-headed attention mechanism.
$\boldsymbol{Q}$	The query matrix used in the multi-headed attention mechanism.
$\boldsymbol{V}$	The value matrix used in the multi-headed attention mechanism.
$\mathbb{E}$	The expected value.
$\hat{A}_t$	The advantage estimate used by a reinforcement learning algorithm to determine if the episode outcome is better or worse than expected.
$\hat{r}_t$	The ratio of the probability output of the new policy over the old policy for a given action and state.

a_t	An action taken by an agent at environment time step t.
$B_k(\boldsymbol{\pi})$	The best strategy (best response) for agent k given that all the other agents' strategies, captured inside $\boldsymbol{\pi}$, are fixed.
D	The number of output values in the continuous action vector of an agent.
d_k	A normalising value used in the scaled dot product attention mechanism.
G_t	The cumulative sum of rewards experienced from time step t onward.
$I(w)$	An objective function defined over parameters w.
$J(\theta)$	An objective function defined over parameters θ.
$L(\theta, w)$	The objective function of the PPO algorithm.
$L_t^{clip}(\theta)$	The actor objective of the PPO algorithm at time step t.
$L_t^c(w)$	The critic objective of the PPO algorithm at time step t.
o_t	An observation received by an agent at environment time step t.
$o_{0:t}, a_{0:t-1}$	All observations received and previous actions performed by the agent up to time step t.
r_t	A reward received by an agent at environment time step t.
S	The entropy function for a given policy.
s_t	The state of an environment at time step t.
T	The final time step of an environment run.
$v_\pi(s_t)$	The expected cumulative reward that the agent, acting with policy π, will receive from state s_t onward in an episode.

Chapter 1

Introduction

1.1 Motivation

In May 1997, IBM's Deep Blue became the first computer chess engine to defeat a reigning world champion in a chess game under regular time controls [14]. This marked a historic moment for the field of artificial intelligence (AI). Deep Blue did not have any learning capabilities, and its chess playing ability was entirely based on manually encoded expert knowledge combined with brute force search over possible future board positions. In that same year a new competition was launched with the goal of becoming a next grand challenge for AI. This competition was named the Robot Soccer World Cup, or RoboCup for short [54]. Robot soccer has simple rules while still having many of the aspects AI struggled with at the time, such as object and scene decoding, accurate humanoid motor control and multi-agent communication. By limiting input sensors, output actuators and body shapes to be similar to humans, more focus could be placed on increasing the intelligence of such systems, as opposed to better hardware configurations. Robot soccer was therefore chosen as a challenging and fun medium in which to experiment with different algorithms, while hopefully pushing AI incrementally closer towards human level intelligence and reasoning. The official goal of RoboCup is: "*By the middle of the 21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of FIFA, against the winner of the most recent World Cup*" [53]. This goal might sound overly ambitious, but even if it is not achieved it will most likely still encourage great innovation in this field.

In 2016, the human world champion of Go was defeated for the first time by an AI system; a system created by DeepMind called AlphaGo [58]. Go is a popular board game for which up until 2016 it was considered nearly impossible for AI systems to compete with humans. What makes the win in 2016 particularly significant is that the system completely learned its own game strategy (or policy) from raw data, using reinforcement learning and human gameplay imitation. Here and in the rest of this dissertation a strategy or policy refers to a mapping for each state (or observation sequence) to an action. AlphaGo is a

strong departure from previous AI systems such as Deep Blue, which had a handcrafted (manually programmed) strategy. A later version called AlphaGo Zero learned to play Go at superhuman level without examples of human gameplay, and entirely from self-play using reinforcement learning [59]. Subsequently, DeepMind unveiled a system called AlphaZero [60] which was able to play not only Go, but also chess and Shogi, at superhuman level using the same general learning architecture and completely through self-play.

It might be hard to imagine how the successes of AlphaGo would transfer to real robots playing soccer in RoboCup. Reinforcement learning requires millions of environment steps (interactions) which are not that easy to generate when using real robots. Fortunately, recent work in automatic domain randomisation (ADR) has shown that it is possible to transfer policies trained only in simulation to real robots [46]. ADR allows agents to learn to adapt to new environments at inference time (by agent we mean a player in an environment that is learning or has learned its policy through direct interaction with the environment). When using ADR in soccer, we might be able to get away with training mostly in simulation. When the trained agents are placed in the real world they can then more easily adapt to real-world physics without needing retraining.

In RoboCup's simulation leagues, algorithms that are mostly hardcoded currently still perform better than end-to-end reinforcement learning algorithms. While more and more machine learning is being applied in RoboCup simulation leagues, recent competition winners usually rely on sets of predefined strategies with some machine learning and reinforcement learning used for fine tuning of strategies [48, 1]. This can be problematic, and prevent these strategies from being transferable to real robots as they might easily overfit to one simulation environment. Furthermore, as was seen with AlphaGo and AlphaZero, the more handcrafting that is done from human intuition, the more the approach is limited to the expertise of human designers. Once DeepMind removed the reliance on expert play, and just focused on self-play, the algorithm became considerably stronger. End-to-end reinforcement learning is the ability of a reinforcement learning algorithm to learn everything, from low-level soccer skills to high-level team tactics, directly from data generated through game simulations. In this work, we investigate the application of end-to-end reinforcement learning to the full 22 player game of 2D simulation soccer. We want our algorithms to not overfit to our specific soccer environment's dynamics, to enable ADR methods in future work.

One reason why reinforcement learning algorithms have not been so successful in robot soccer simulations to date is the amount of computing power needed to achieve good results when compared to other more handcrafted methods. Liu et al. [35, 36] used multiple computers running in parallel to train their agents in a soccer simulation with only 4 players. We will investigate how computationally efficient an end-to-end multi-agent reinforcement learning (MARL) approach can be made for simulation soccer in the full 22 player (11 vs 11) setting.

1.2 Problem statement

The goal of RoboCup is to encourage the creation of algorithms that can be utilised to play soccer well (compared to humans), but also potentially generalise to other real-world domains. Current RoboCup participants still manually encode soccer-specific knowledge into large portions of their algorithms. The reason they do this is because it takes a long time to train agents to learn competitive strategies from scratch, compared to handcrafted algorithms. We therefore investigate whether improvements can be made to allow for a learning algorithm to derive strategies, that defeat simple handcrafted opponents, in a reasonable time frame.

The problem environment we focus on is a simple 2D simulation, for reduced complexity. Top teams in recent 2D RoboCup competitions [48, 78] are relying more and more on machine learning and specifically reinforcement learning to boost performance. These teams usually subdivide the full problem into different tasks (e.g. dribbling, kicking and passing) that can be optimised individually with machine learning methods. Another line of research starts with a simplified soccer environment, such as the 4 player setting, and applies end-to-end multi-agent reinforcement learning on it [35, 36]. We found no evidence in the literature of successful attempts to apply end-to-end MARL on the full 22 player setting, where training is known to be much more unstable [13]. Here unstable training refers to high variance in the team's performance (win rate) when compared to fixed opponents at various instances throughout training. High variance usually indicates that the algorithm is struggling to learn and can even cause an unrecoverable collapse in performance. Although training in a setting with so many agents is challenging, one can imagine that at least some of the training needs to happen here for high-level strategy formation.

We therefore investigate whether it is possible to derive methods that can stabilise and speed up training in the full 22 player environment. If we can demonstrate that end-to-end learning algorithms can work in a 2D simulation domain, we could in future scale to other simulation environments or even real robot soccer using ADR [46]. We further investigate whether it is necessary to train in the 22 player environment when compared to training in the 4 player setting and then applying those agents to the full 22 player environment. Lastly, we investigate whether it is possible for agents to train stably in the full 22 player environment using only consumer-grade hardware (one GPU and one CPU).

1.3 Objectives

In this work, we focus on applying end-to-end MARL on simulation soccer, which would avoid a need for knowledge of soccer-specific strategies and lead to algorithms that could be useful in other environments.

The main objectives are as follows:

- investigate methods with which an efficient end-to-end MARL system, trained entirely through self-play, can scale to the full 22 player 2D soccer environment;
- design a custom 2D soccer environment with a faster runtime than the official 2D RoboCup simulator, for faster experimentation, that remains sufficiently close to the RoboCup simulator with regards to the MARL-specific challenges it presents;
- experimentally verify whether our solution can learn competent strategies through self-play, by evaluating the trained teams against opponents with handcrafted strategies.

1.4 Constraints

We will enforce the following constraints on our learning algorithm:

- limit the computational requirements as much as possible so that researchers with limited hardware can use it;
- refrain from training against handcrafted strategies, as many environments or problems might not have strong handcrafted strategies, and use such strategies only for evaluation;
- have independently executing policies for the individual agents on the field, as is the case in RoboCup;
- have each agent receive only partial information about its environment at every time step, as is the case in RoboCup;
- ensure that our solution remains general, without overfitting to our custom soccer environment, as we want our algorithm to be able to work in the RoboCup environment as well with a comparable number of agents.

1.5 Overview of approach

Our approach focuses on computational efficiency. Many of the existing approaches elaborated on in Chapter 2, train only in environments of up to 4 players. Liu et al. [36] specify slow training as the reason for why they do not scale to 22 players. We limit our computational resources for training to one CPU and one GPU, and to maximise what we can achieve with these resources we run parallel instances of the workers (experience collectors) on the CPU and a learner on the GPU. The workers generate experience by interacting with a custom simulation environment. This environment, presented in Chapter 4, is built for speed and compiles to machine code for faster execution. We also compile both the worker and learner execution functions to machine code. We constrain our simulation environment to operate with a continuous action space where agents receive partial observation inputs,

which are dynamic in size and dependent on what the agents can see in their respective vision ranges. This is closely aligned with the RoboCup environment and would therefore make it easier for us to transfer our final algorithm to the RoboCup environment.

With the environment in place we need a reinforcement learning algorithm to generate ever improving strategies. To further allow our algorithm to be transferable to the RoboCup competition, we constrain it to have independently executing policies. To increase throughput we constrain every agent in a team to use the same policy network. This both reduces the number of parameters that needs to be learned as well as allows for much faster inference through batching, further discussed in Chapter 5.

Training is known to be much less stable when using reinforcement learning in a relatively large multi-agent environment, such as full 22 player soccer. This instability can be exacerbated when using self-play, where agents play against opponents with strategies identical to theirs. To address this we use an on-policy reinforcement learning algorithm called proximal policy optimisation (PPO) [57]. PPO, further discussed in Chapter 3, is known to be stable as it controls how fast the policy can update during training. To address the self-play component we experiment with two methods, namely Polyak averaged opponents and league training. Both of these methods attempt to combat the problem where agents overfit to countering only their current strategies and forget past ones. Lastly, to further stabilise training we provide full state information to the critic network, which is equivalent to a team's coach in soccer. The use of full state information, instead of partially observable information, allows the critic to provide less noisy estimates of the expected game outcome and therefore allows the policy updates to be less noisy. This state information includes the time remaining, the positions and rotations of all the agents and the position and velocity of the ball. We can conveniently provide this state information because the coach is only used during training and not in evaluations or competitions, where the agents receive only partial information. When all these techniques are used together we can significantly reduce the computational requirements of stable training in the full 22 player environment, to one consumer-grade GPU and CPU setup.

We evaluate both league training and the Polyak averaged opponent algorithm by training on the consumer hardware specified. We train our algorithms in the full 22 player environment and throughout training validate them against various handcrafted opponents. We also evaluate our algorithms trained in the 22 player environment against the best algorithm trained in the 4 player environment for the same amount of time, to determine whether it is beneficial to train in the full environment. We also perform some ablation studies to determine how much each of the main components contribute to the final performance of our best algorithm. We evaluate the difference in gameplay between the team trained in the 22 player environment and the team trained in the 4 player environment. Lastly, to show transferability, we evaluate our system on the 2D RoboCup environment.

1.6 Contributions

The main contributions of this dissertation are as follows:

- providing reasons (Section 5.1) to help understand why MARL algorithms have not been so successful in the full 22 player environment to date, with reference to correlated experience, non-stationary environment dynamics and self-play strategy collapse [11];
- proposing a new algorithm (Section 5.2) that combines different stability improvement components to address the training instability issues, leading to stable training in the full 22 player environment with only one CPU and GPU;
- experimentally verifying (Section 6.3) that the proposed algorithm can learn good strategies that defeat a variety of handcrafted strategies after just over two days of training;
- experimentally verifying (Section 6.3) that agents trained in the full 22 player environment achieve a higher win rate than agents trained in the 4 player environment when evaluated in the full game, thereby validating the need to train in the full 22 player environment;
- experimentally verifying (Section 6.4) through an ablation study the effect and necessity of various subcomponents of our proposed algorithm.
- experimentally verifying (Section 6.7) that our system can be applied to the RoboCup environment and that it exhibits stable training in the full 22 player setting.

1.7 Outline

In Chapter 2, we highlight existing work on applying learning algorithms to derive strategies in soccer and other environments. In Chapter 3, we introduce some of the theory required for our proposed algorithm. In Chapter 4, we present our lightweight simulation environment for training and evaluation. In Chapter 5, we hypothesise what the challenges are in applying MARL in the full 22 player environment, and propose possible solutions. By applying these solutions we show in Chapter 6 that agents trained in the full 22 player environment can learn to outperform a variety of handcrafted strategies, as well as agents trained in the 4 player environment and evaluated in the full 22 player environment. We also perform an ablation study to demonstrate how some of the main improvements proposed in this work contribute to the final result. In Chapter 7, we conclude with a discussion and present some additional ideas for how this work can be expanded upon in the future.

Chapter 2

Related work

In this chapter we provide a brief overview of existing research on simulation soccer and algorithms that allow neural networks to be trained in this environment. This background information will serve as a guide on what parts of existing research to focus on in order to address the problem described in the previous chapter.

We start in Section 2.1 with an overview of previous research that motivates our work, where it is shown that agents training exclusively in simulation environments can directly benefit real-world robotics as well. In Section 2.2, we provide an overview of the different RoboCup leagues as an inspiration for our own custom environment. In Section 2.3, we then describe some of the approaches used in RoboCup and other soccer simulations, which vary from handcrafted algorithms to reinforcement learning algorithms. Finally in Section 2.4, we look at various algorithmic improvements that may allow neural networks to be effective in soccer environments, specifically through reinforcement learning. We end off the chapter in Section 2.5 with a summary.

2.1 Automatic domain randomisation

If we want robots to solve complicated real-world challenges it might be necessary to incorporate, at least partially, higher-level end-to-end learning. Unfortunately, real robots cannot easily execute millions of trial and error actions in the real world. One alternative is to construct a simulation of the problem and let a robot train in this simulated environment. However, building highly realistic simulations of the real world is in many cases nearly impossible, and policies trained in simplified or idealised simulations rarely transfer well to the real world. One way around this problem is to vary the dynamics of the environments trained on so that the simulated robot can quickly learn to adapt to new physical setups.

In 2019, OpenAI demonstrated that a real robotic arm could be taught to solve a real Rubik's Cube [46], as shown in Figure 2.1. Remarkably, the network controlling this robotic arm was completely trained in simulation using reinforcement learning. This stands in contrast to the previous popular belief that agents trained in simulation generally cannot

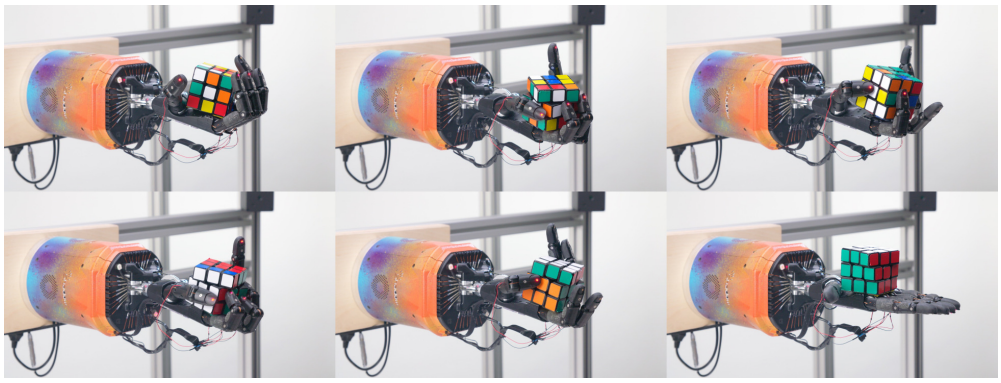


Figure 2.1: Snapshots of a robotic arm solving a Rubik's Cube, using a policy trained entirely in simulation. Source: Paino et al. [46].

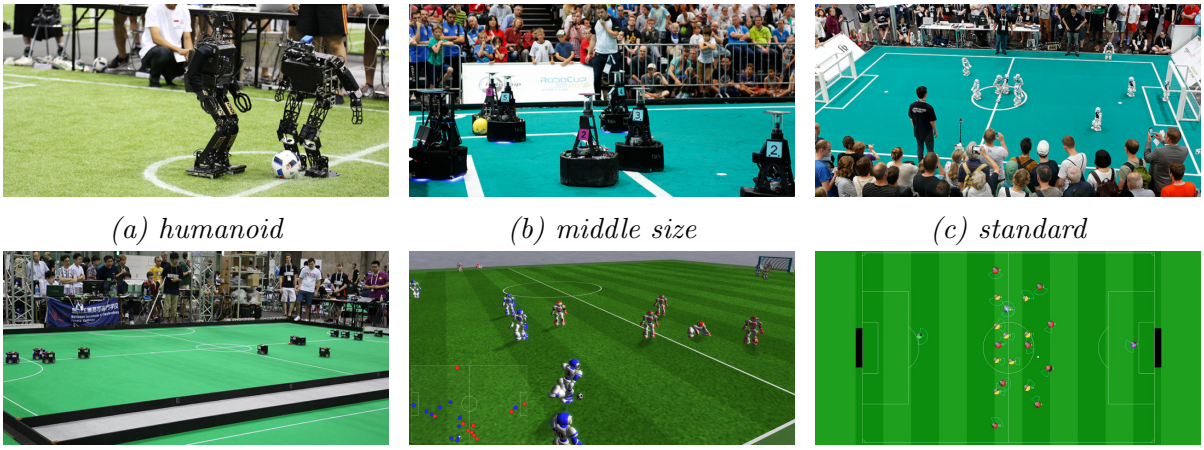
adapt to the real world, because real-world properties, e.g. friction, are difficult to simulate efficiently and accurately. The team at OpenAI sidestepped this problem by not trying to create a perfect simulation of the real world. Instead they slightly varied the simulation environment's dynamics, such as gravity and joint sizes, each time a new simulation run was initiated. The agent therefore did not only need to learn to solve the Rubik's Cube in one environment, but also needed to learn to adapt to different environment dynamics on the fly. When the agent was then instructed to control the real robotic arm, it could adapt to the real world's dynamics.

Solving a Rubik's Cube in simulation is already a challenging task, and becomes even more challenging when one adds domain randomisation. The OpenAI team therefore devised a method called automatic domain randomisation (ADR) to automatically increase the variance in the dynamics for different environments as the agent becomes more capable. Initially the agent is only tasked with solving a Rubik's Cube in one simulation environment. Then, once it starts achieving sufficiently good results, the ADR algorithm starts increasing the variance in the simulation properties between each run. This allows the agent to always have high variance in its reward signal, which encourages effective learning.

While we will not attempt to adapt our algorithm to work on real robots, our work serves as an initial step in that direction. We first focus on giving a team of agents the ability to find good strategies in one simulation environment through end-to-end learning. Then, in future work, the team of agents can be trained to dynamically adapt to various simulations with different physical setups. Once these agents are trained, they can be deployed in the real world and adapt to its physics while they are playing a soccer game.

2.2 The RoboCup leagues

The RoboCup soccer division consists of different leagues, namely the humanoid, middle size, standard, small size and simulation leagues. Depictions of these leagues are given in Figure 2.2. Each league focuses on a different set of challenges. The humanoid league, for example, most closely represents the format for which the end goal of RoboCup aspires to (having robots play against humans). However, this is also the most challenging league as one has to deal with large physical humanoid robots.



(a) humanoid *(b) middle size* *(c) standard*

(d) small size *(e) 3D simulation* *(f) 2D simulation*

Figure 2.2: The different RoboCup soccer leagues, which allow participants to focus on different aspects of robot soccer. Sources: RoboCup [51, 52], Akiyama [3].

2.2.1 3D simulation league

As seen in Figure 2.3, the 3D RoboCup simulator can be used to execute soccer games using 3D simulated humanoid robots.

A full soccer game is played with 11 players on each side. At every game time step the players receive partial egocentric observations for all the objects and stationary field markers they can see. To simplify the information processing task the players are not provided with raw image data, rather with information about each object in view. This includes distances and angles of objects relative to a player, and changes in relative velocities. Each player also receives information about itself, namely a fatigue indicator, actuator angles, acceleration measurements and audio communication data from other players. With this information, each player can then produce action values for each of its actuators. The 3D simulation environment is particularly challenging as players need to have low-level skills like walking, running and dribbling, as well as some form of team strategy.

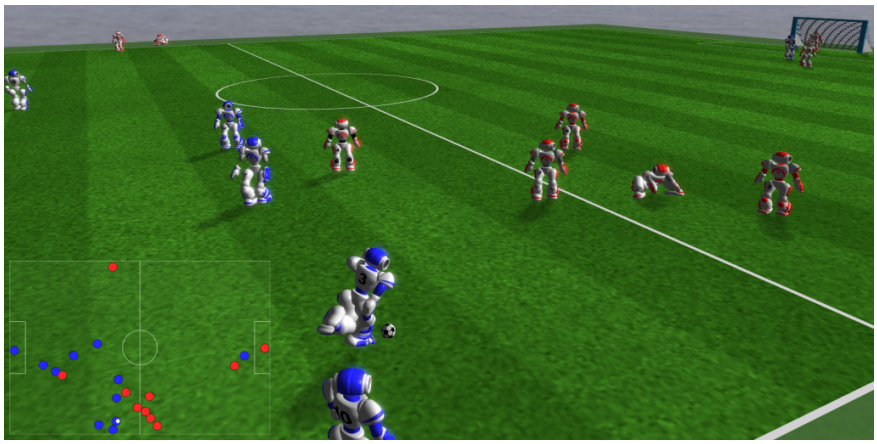


Figure 2.3: Visualisation of a game in the RoboCup 3D simulation league. Source: RoboCup [52].

2.2.2 2D simulation league

As the name suggests, the RoboCup 2D simulation league is played on a two-dimensional simulated field. Each team consists of 11 independent autonomously controlled players, with the capability to periodically communicate with one another. There are 22 players on the field, with each team having one goalkeeper. Each team has an additional coach which can observe more information about the game and then communicate with the players. There is also a digital referee which detects certain game situations, like the ball leaving the field, and then intervenes appropriately.

As can be seen in Figure 2.4, each player is represented by a circle, where one half (its front side) is coloured according to the team it belongs to and the other half is coloured in grey. Goalkeepers have separate colours to easily distinguish them from other players. Players can also temporarily change colours if they are in special game situations, as seen with the blue players.

Each player receives noisy measurements from the environment through three types of sensors, namely the aural, visual and body sensors. The aural sensor is used to receive communication messages from the coach, teammates, the referee and even the opponents. This sensor is implemented to mimic how players in the real world communicate using speech. Messages might also get lost if the player is too far from the sender, with the exception of messages from the coach and the referee. Each player receives aural messages every second simulation step, where a simulation step is typically 100 milliseconds in length.

The second input sensor that each player is equipped with is vision, allowing the player to see objects and the field boundaries within a certain vision cone. By default a player can see in a 90° cone, with new visual information arriving every 150 milliseconds. The player receives information about objects in its vision range, which include their relative positions

Figure 2.4: Visualisation of a game in the RoboCup 2D simulation league, using the game monitor. Source: Akiyama [3].

and velocities. The player can dynamically adjust the view quality and view angle. However, these values are inversely proportional to the viewing frequency. Thus, having high view quality and a wide view angle will result in long waiting times before new visual information is provided to the player. To better emulate a real sensor, noise is added to all the visual information provided to the player.

The last set of sensors available to each player is the body sensors. These sensors measure information like the game time remaining, the neck angle relative to the player's body, the speed and direction of the player, the stamina of the player and other factors like if a collision is occurring with another object. On the default setting the body information is presented to the sensors every 100 milliseconds.

The players also have different output methods to influence the world. Firstly, they can move forward or move backward in the direction their bodies are facing. They can also rotate around their centre to reorient their bodies. Each player has a neck which can be rotated within a certain range from the forward direction of its body. Therefore, the player can look in a different direction than the direction it is moving in. When the player decides to kick the ball, if it has the ball, the ball will travel in the direction its neck is facing. This allows for more versatile play, as each player has more accurate control over where it is looking and where it wants to pass the ball, without having to stop or redirect its motion.

2.3 Approaches to solving simulation soccer

We next investigate some approaches used by previous RoboCup participants as well as prior research on other simulated soccer environments.

2.3.1 2014's 3D simulation league winner

A robotic agent playing 3D simulation soccer faces many distinct (but not mutually exclusive) tasks, like dribbling, kicking the ball, recovering from a fall, etc. One way of approaching this large problem set is to compartmentalise the tasks into hierarchies. The agent can then learn lower-level behaviours like running and getting up before learning higher-level skills such as passing the ball. One of these hierarchical machine learning paradigms is called layered learning [61] and has been successfully applied in the RoboCup 3D simulation league, leading to 2014's winning team [38]. One problem with layered learning is that when lower-level modules are learned and frozen, agents might not be able to learn optimal behaviours overall. Moreover, learning all lower-level modules simultaneously might be too slow or even unstable. MacAlpine and Stone [38] address this problem with overlapping layered learning, where lower-level modules are partially updated during learning, for increased final performance. They successfully learned 19 layers of behaviours and optimised over 500 parameters. Up to two agents were trained together in a soccer environment, and learned behaviours were evaluated in the full 22 player game. In our work, we opt to use a simpler 2D environment with less complicated environment dynamics. We do not use layered learning and aim to derive everything, from individual agent skills to high-level team strategies, directly from competitive self-play. We also aim to train our team in the full 22 player environment, which we believe can enable better team strategy formation.

To simplify the soccer problem we next consider the 2D simulation league. Our goal is to first achieve 11 player end-to-end training in a 2D domain, before moving to the more complicated 3D domain in future work.

2.3.2 2019's 2D simulation league winner

The winner of the 2019 RoboCup 2D simulation league was a team called Fractals2019 [48]. The main strategy of this team was to combine an evolutionary algorithm with guided self-organisation. They combined artificial evolution with human innovation, a method also called human based evolutionary computation (HBEC). As mentioned in their work, Prokopenko and Wang [48] rely on methods such as particle swarm based self-localisation, tactical interaction networks, dynamic tactics with Voronoi diagrams, bio-inspired flocking behaviour, and diversified opponent modelling. HBEC is used to determine what combination of these and other techniques provides the best results by manually (or automatically) adjusting hyperparameters and measuring the agent's fitness, e.g. using goal difference with respect to other agents. This drastically decreases the amount of computing power needed, compared to generic evolutionary or other reinforcement learning algorithms, as the agents are provided with a strong prior over what good strategies are. The task of the evolutionary algorithm is now reduced to only fine tune how much and when to use each component, while also evolving other attributes like when to move fast and when to conserve stamina. While these methods provide relatively fast training times, they rely heavily on human intuition

for what good soccer strategies are.

2.3.3 Multi-year 2D simulation league winner

We now describe the algorithm HELIOS [4, 5], which won the RoboCup 2D simulation league in 2010, 2012, 2017 and 2018.

HELIOS uses a team formation model that implements Delaunay triangulation [32], to generate points that are sufficiently spread out from one another. This allows the agents in the HELIOS team to cover greater areas of the field, without bunching. The algorithm also employs a tree search method to search through sequential ball kicking actions among multiple players on the same team, in order to determine which actions the agents should take. A goal score criterion is used as an evaluation to weigh different action sequences.

Searching through all possible combinations of ball passes quickly becomes intractable. To help the search algorithm, Akiyama et al. [4] prune all actions that are not part of the intended tactics that they want their team to exhibit. To do this, they use a support vector machine (SVM) classifier to determine whether a given action is part of the intended tactic or not. To generate the label set for this SVM, they extract game data from different training runs where they manually label actions, using a graphical user interface, as being part of the intended tactic or not.

In the 2019 version of their algorithm they focused on dynamic adaptation of the team strategy based on what strategy the agents observe their opponents executing. In RoboCup a team is provided the log files of previous games played in a competition. By using this log data the HELIOS team could predict the team strategy based on their previous actions, and better exploit their opponents. To do this they used methods adapted from natural language processing, such as Word2Vec [40], which translate actions and players into real-numbered vectors. They then use clustering to measure the similarity between the current opposing team and team strategies in their database. Once they find a matching strategy they play a specific counter strategy.

2.3.4 Teaching a 3D simulated robot to run

The recent surge in interest in the field of reinforcement learning is largely due to the now famous 2013 paper on playing Atari games with deep reinforcement learning by DeepMind [41]. While most of the concepts in the paper, like combining neural networks with reinforcement learning, were not new, it was the scale at which they could get it all working together that was astounding. The authors showed how the same algorithm could be applied to many Atari games and achieve good, and in some cases superhuman, performance. They did this by combining the theory behind reinforcement learning with the function approximation capabilities of deep neural networks, while also including ideas such as experience replay.

The basic idea behind reinforcement learning is that one or multiple agents can interact

with an environment. The agent receives information from the environment, e.g. what it is seeing or feeling, and also if it has received any rewards. It then learns a function mapping, typically represented by a neural network, that takes as input observations up until the current time step in the environment and tries to output the best action to maximise the expected reward it will receive. Gradient based reinforcement learning uses gradient descent, or gradient ascent in the case of our work and further discussed in Section 3.1.1, to directly optimise the neural network to become better at predicting this expected reward and predicting the best action to maximise the reward function. We will use one optimiser to train all our networks on a combined objective function.

While the RoboCup soccer leagues have not seen widespread use of end-to-end reinforcement learning, where the team's complete strategy is learned from playing games, there has been application in some aspects of the game. One such example is in teaching the NAO robot to run faster in 3D simulation [1]. The authors use reinforcement learning to enable an agent to run faster than could be manually programmed. In a game such as soccer, stable running speed can be an important indicator in determining who will win a game. The average speed of their agents increased from the manually programmed version with a speed of 1.46m/s to 2.45m/s. They also stated that this was the fasted recorded straight line running speed of any agent in the RoboCup 3D simulation competition. The algorithm used to train these agents is called proximal policy optimisation (PPO), which we discuss in Section 3.3.5 as it is also the main reinforcement learning algorithm in our work.

2.3.5 2021's 2D simulation league winner

We now discuss the algorithm used by the 2021 2D simulation league winner, called CYRUS-2021 [78]. The CYRUS team also placed fifth, fourth, second and third in 2014, 2017, 2018 and 2019, respectively. Similar to HELIOS [5], the CYRUS team used a search algorithm to predict the best actions to take on the field. Zare et al. [78] state that in recent years they have focused predominately on incorporating more machine learning into their agents, in lieu of hardcoded strategies. They mention that they improved the defensive decision-making process of their agents using reinforcement learning, used neural networks for better prediction of an opponent's behaviour into the future, and optimised their agents' ball shooting skills. To learn to shoot and pass the ball to specific targets, the CYRUS team uses a one layer feedforward neural network. This network was trained using supervised learning on multiple scenarios where the agent needs to shoot the ball to a specific target location.

In the latest update to the CYRUS algorithm, Zare et al. [78] investigate how noise in the measurements provided by the simulation affects the agents. They showed that this was a dominant factor causing their team to lose to other top teams. In their 2021 paper, Zare et al. [78] address this issue by using a feedforward neural network as a denoiser to predict the full state properties of agents from noisy observations. They then exploit this neural

network by providing the predicted full state information, instead of the raw observations, to the behaviour module. This update resulted in a 11.3% increase in win rate against the 2019 HELIOS team. In the finals of the 2021 competition they faced off against HELIOS and won the game 1–0.

2.3.6 DeepMind's simplified soccer environment

One of the best examples of end-to-end reinforcement learning being applied to simulation soccer was demonstrated by a team from DeepMind in 2019. Liu et al. [35] investigated if it was possible to train a team of two soccer agents using only self-play. To do this they used the MuJoCo physics engine [67], visualised in Figure 2.5. Each agent has the ability to move forward, move backward, jump and rotate about its vertical axis. An agent can kick the ball by either rotating about its vertical axis and let its feet hit the ball, or dribbling it with the circular part of its body. The agents cannot touch each other, which incentivises them to not intentionally block or hit opponents. The ball is also placed back into play at a random position if kicked out-of-bound.

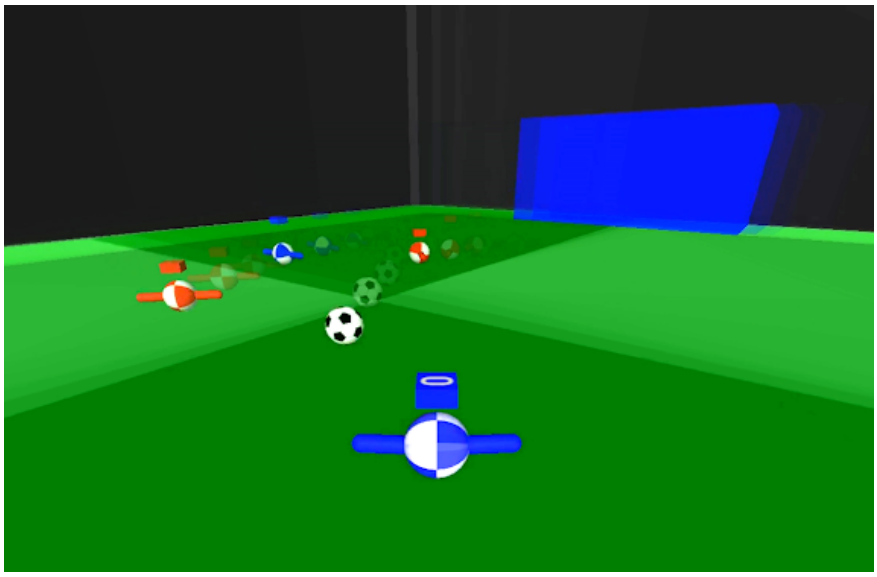


Figure 2.5: Game window of a soccer game in DeepMind's simplified soccer environment. Source: Synced [64].

The inputs provided to the agents are all the players' positions, velocities and accelerometer information, and also egocentric ball position, velocity, and angular velocity, and the corner positions. This makes the complete input a 93-dimensional vector. An important thing to note is that in the 2D RoboCup simulation league there is only egocentric viewpoint information provided on position, velocity, etc., as opposed to some absolute values provided here. How much this absolute information affects the end results is however still unclear.

Figure 2.6: Game window of a soccer game in DeepMind's humanoid soccer environment. Source: Geeky.news [22].

One commonly observed problem in the multi-agent reinforcement learning setup is that the best performing team typically overfits to one opponent's strategy and usually cannot adapt to new strategies. Liu et al. [35] therefore decided to use the population based training (PBT) [30] algorithm to maintain different sets of agent networks with possibly different strategies. This allows the best agents to train on a wider variety of strategies. Another advantage of PBT is that it allows for hyperparameter search, and continuous adaptation of hyperparameters. PBT is also completely decentralised and therefore can be scaled to run on multiple computers in a trivial manner. The authors chose a population size of 32 agents, which all trained in parallel on multiple GPUs.

At the end of training the agents could competently play a game of soccer. While it is challenging to estimate how good a specific team is, the authors noted that the agents could be seen scoring goals and passing the ball to one another. The latter is an important result as it shows that cooperative behaviour can be learned using current reinforcement learning algorithms.

In this work, we opt to not use PBT and will rely on only one GPU for training in the full 22 player environment. This should make it easier for other researchers, who might have limited computational resources, to build upon our work.

2.3.7 DeepMind's humanoid soccer solution

More recently, a form of layered learning has been applied on a simulated humanoid body with 56 degrees of freedom [36]; see Figure 2.6.

Some of the complexity of the soccer environment was reduced, by making the ball

bounce back from the sidelines and removing penalties altogether. Agents were trained in three stages. Firstly, agents would perform imitation learning for low-level movement skills, using motion capture data from real human players. In the second phase, individual agents would learn mid-level skills by performing basic drills, e.g. running and dribbling. In the third phase, multi-agent reinforcement learning was used with population based training in a 4 player (2 vs 2) environment, with 16 independent learners that trained in parallel on multiple GPUs. The agents learned to play competent soccer in this environment and the authors also presented evidence that the agents could pass the ball and recover from falls. A multi-headed attention mechanism [70] was used, which we also use in our work. Unfortunately, the computational requirements for agent policies scale quadratically with the number of agents, making it expensive to compute agent actions in the full 22 player environment. In our work, we adapt the network to scale linearly with the number of players on the field, which enables significantly faster training in the 22 player environment.

2.4 Reinforcement learning for soccer environments

In this section we discuss some additional techniques used in the literature that are of interest to our work.

2.4.1 OpenAI Five

In Figure 2.7 a game snapshot of the OpenAI Five [11] team can be seen playing Dota 2, a popular real-time strategy game. In Dota 2 a team of five players, each controlling a hero unit, faces off against an opposing team of five players. Players try to defend their team's home base while attacking the enemy's units and buildings. The objective of the game is to take down the enemy team's Ancient, which is a glowing building in their home base. Once the Ancient has been destroyed the game is won by the attacking team. In 2019, Berner et al. [11] created an artificial team that learned through reinforcement learning and self-play to defeat the reigning human world champions in the game.

Figure 2.7: A Dota 2 game window showing the OpenAI Five team. Source: Alvarez [7].

This research is of interest to us as our soccer environment also consists of multiple agents and has a self-play component to it. Furthermore, as will be discussed in Section 3.3.5, we also use proximal policy optimisation [57] which is a popular reinforcement learning algorithm. While Berner et al. [11] opted to train one controller to control all five players simultaneously we opt to have independently executing policies, to better align with the rules of the 2D RoboCup league. The most useful aspect that we take from OpenAI Five is their league training, where the latest team plays not only against itself, but also against previous versions of itself. AlphaStar [42] employs a similar strategy by placing a league of independently learning agents against each other. This prevents a common pitfall when using self-play called strategy collapse, where a team only tries to find good strategies against its current strategy and forgets all past strategies. In doing so, its final strategy can become brittle as it does not work against a variety of other opponents. We provide a more detailed explanation of league training in Section 3.4.5.

2.4.2 Dealing with dynamic inputs to neural networks

Neural networks are function approximators used in reinforcement learning. Standard neural networks usually take in a fixed size input or in the case of recurrent neural networks, further discussed in Section 3.1.2, an order dependent input. However in our case we need each agent to be able to process information from all the players it can see, which might change in number from observation to observation. We also know that the order in which we present the players to the agent should not change its action output. One promising approach that addresses both these issues is to use an attention mechanism.

In recent years, attention mechanisms [16] have become an important part of deep learning. They are loosely inspired by biological systems in the human brain that tend to focus

on distinct parts when processing large amounts of information. A seminal research paper by Vaswani et al. [70] popularised attention mechanisms in deep learning by introducing the Transformer architecture, shown in Figure 2.8. Before their paper, dominant sequence transduction models would rely on complex recurrent or convolutional neural networks that included encoder and decoder networks. Vaswani et al. [70] proposed a simple network architecture based on attention mechanisms for both the decoder and encoder networks, which achieved state-of-the-art results in many sequence modelling tasks. They achieved this by adding positional encodings to input sequences and then applying many layers of multi-headed attention modules followed by feedforward layers. Using attention allowed this model to extract information from much larger sequences which previous recurrent or convolutional models struggled with.

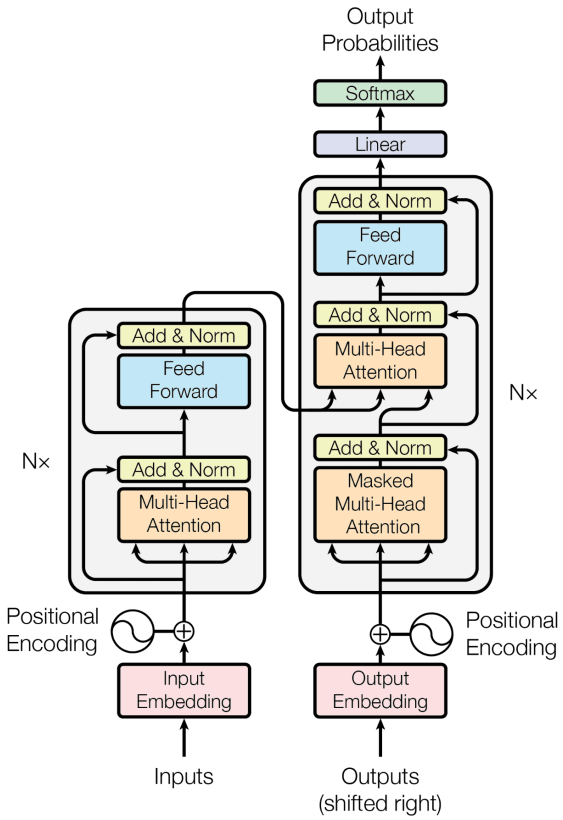


Figure 2.8: Transformer architecture using the multi-headed attention mechanism for both the encoder and decoder module. Source: Vaswani et al. [70].

In our work, we use multi-headed attention modules to both allow our agents to receive a variable input data size from the environment that is order invariant, and also allow them to learn to focus on what is important to process in that moment.

2.4.3 Multi-agent PPO

Recent work by Yu et al. [76] showed that the proximal policy optimisation algorithm [57] can work well in the cooperative multi-agent setting, with slight adaptations such as improving inputs to the value function using agent-specific global states, improving training data usage, and setting the PPO clipping value lower. We also use these recommendations in our work, and add a few additional improvements as described in Section 5.2.

2.5 Summary

At the start of this chapter we discussed why addressing simulation soccer first might be beneficial to not only the RoboCup simulation leagues, but potentially also the leagues using real robots. The RoboCup 2D simulation league is seen as a natural starting point for the development of end-to-end reinforcement learning in robot soccer, as it has the most sensory abstraction. While machine learning is being applied more frequently in 2D soccer each year, end-to-end reinforcement learning is not yet used widely. Current winners seem to rely on human based intuitions and rules, with various machine learning components on top for strategy fine tuning. Reinforcement learning is, however, starting to show promise in robot soccer. It has been successfully applied by Abreu et al. [1] to create the fastest humanoid runner in the 3D simulation league at the time. It has also been used by the winning participants of the 2021 2D simulation league competition [78] to improve their team's defensive decision-making ability. End-to-end reinforcement learning has also recently been applied to train agents in a simplified soccer environment completely from scratch [35]. However, they trained 32 agents in parallel, with only two players on a team. Furthermore, Liu et al. [36] required 16 independent agents that trained in parallel on multiple GPUs, as well as imitation learning on expert demonstrations to generate good priors before applying reinforcement learning. To successfully apply reinforcement learning to robot soccer, with limited resources, we need to investigate more computationally efficient solutions.

We also described various tools we can utilise to make it easier to apply reinforcement learning to simulation soccer. One problem that arises in soccer is that there is no simple fixed opponent to use for the agents to train against. The opponent should not be too easy or too difficult to play against, as both would hinder learning. However, we also do not want to explicitly provide handcrafted opponents as they might not always be available in other domains. Therefore self-play, where the agents play against versions of themselves, seems like a natural fit. Self-play has known challenges, for example the learning team might only learn good strategies to counter its current strategy and forget about its previous strategies. To combat this, Berner et al. [11] use league training where the opponents are semi-randomly selected from previous version of the current agents. Therefore the current agents have to learn new strategies that defeat all its previous ones, and this generates a more general strategy. We employ a variation of this league training in our work.

Liu et al. [36] suggested using multi-headed attention to give each agent the ability to focus only on aspects of other players that are important to it at the current moment. However, Liu et al. [35, 36] provide full state information to their agents. As is described in Chapter 4, we align our environment with RoboCup by providing only partial observations to the agents. Here attention also provides benefits as we can feed in a varied input size depending on the number of other players that are seen by an agent.

With this background on related work we now move on to theory on various components for the construction of our final algorithm. This theory is discussed in the next chapter.

Chapter 3

Theoretical background

In this chapter we provide some theoretical background for the algorithms used in our work. In Section 3.1, we provide a quick overview of the different neural network architectures used to construct our agents. We introduce reinforcement learning in Section 3.2 and describe why it is a suitable framework for our objectives. In Section 3.2.1, we discuss the Markov decision process (MDP) which is commonly assumed in reinforcement learning. In Section 3.2.2, we introduce the partially observable Markov decision process (POMDP) for the single-agent setting. In Section 3.3.1, we derive the policy gradient learning algorithm. We then introduce various improvements to this algorithm in Sections 3.3.2, 3.3.3 and 3.3.4. In Section 3.3.5, we specify the main reinforcement learning algorithm used in our work, called proximal policy optimisation (PPO). We then move on to multi-agent reinforcement learning in Section 3.4, and introduce a number of game theoretical topics that influence our final solution, namely Nash equilibrium, best response strategies, stochastic vs deterministic policies, and self-play. We also introduce the decentralised POMDP (Dec-POMDP) for the multi-agent setting, which our environment (and the RoboCup environment) operates under. We conclude the chapter in Section 3.5 with a summary.

3.1 Neural network architectures

An important part of many reinforcement learning systems is the use of neural networks to form expressive policies and critics for the agents. Within the context of reinforcement learning, networks are typically trained on a reward signal from the environment. We want to maximise the expected reward received by agents. In our environment, an agent also receives partial observations which include the agent's position as well as all the other agents that are visible to it. This input therefore varies in size depending on how many other players the agent sees. The agent also receives information on where the ball is, if it can see the ball. This poses a problem for standard neural networks that require a fixed input size. Secondly, our agents also need some form of memory to allow them to incorporate historical observations when deciding on an action. In this section, we describe the various

deep learning architectures used to address these problems. We start by providing a brief introduction to standard feedforward neural networks and their training.

3.1.1 Feedforward neural networks

Artificial neural networks are powerful function approximators that are loosely inspired by how the brain works. Many recent breakthroughs in machine learning use some form of neural network. They are made up of computational units referred to as artificial neurons, each performing a nonlinear operation on its inputs to produce a scalar output. The computations in a neuron are determined by its internal parameters, and by changing these parameters one can change the computation it performs.

As depicted in Figure 3.1, an artificial neuron takes as input a vector $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]$ which gets multiplied (element-wise) by a weight vector $\boldsymbol{w} = [w_1, w_2, \ldots, w_n]$ representing the neuron's parameters. An extra parameter called the bias (w_0) is also added and the result is summed. A nonlinear activation σ is then applied to produce the output of the neuron.

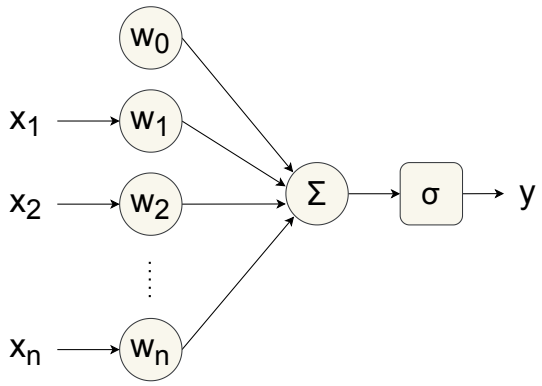


Figure 3.1: Image of an artificial neuron's internal computation.

Modern neural network architectures often use the rectified linear unit (ReLU) activation [21] as it is simple to compute and has advantages when calculating gradients such as preventing vanishing gradients (discussed later in this section). The ReLU activation is defined as

$$\sigma(z) = \max(0, z). \tag{3.1}$$

By combining multiple artificial neurons one can create an expressive function approximator, called a neural network. A neural network has multiple layers and components, namely the input layer where the input vector is presented to the network, a certain number of hidden layers which sequentially transform the input into progressively more complex

features, and lastly the output layer which returns the output of the network. Each neuron in a layer takes as input the outputs of the previous layer, performs an internal computation as explained above, and passes its output to the next layer. Such a network is said to be fully connected as all neurons in successive layers are connected to each other. A neural network can be trained to approximate different functions by incrementally updating the parameters (weights and biases) of all the neurons. By increasing the number of hidden layers and also the number of neurons in each hidden layer one can increase the expressiveness of the neural network with regards to the complexity of the functions it can represent. The larger the neural network, however, the more computational steps it takes to produce an output. The entire network can be seen as representing a function of the form

$$\boldsymbol{y} = f_\theta(\boldsymbol{x}), \tag{3.2}$$

where θ represents all the parameters in the network, $\boldsymbol{x}$ the input vector, $\boldsymbol{y}$ the output and f the network architecture.

With this function approximator in place we would like to find appropriate values for θ so that the neural network performs some useful computation, which can for example be to classify inputs into different output categories as is done in supervised learning, or to take observations from an environment as input and output actions as is done in reinforcement learning. To tune a neural network's parameters most modern approaches rely on some objective function $J(\theta)$, that measures how good the neural network is at approximating some function. If the objective function increases (or decreases if it is a loss) during training, the network becomes better at approximating the function.

One example of supervised learning is where we take some input vector and predict the class that it belongs to. To do this we can use a fully connected feedforward neural network with the number of output neurons equal to the number of classes. A forward pass through the network produces one scalar value as output per class. To turn these output values into probabilities over the classes, we use the softmax function:

$$p(Y = y_k | X = \boldsymbol{x}_k, \theta) = \frac{e^{f_\theta(\boldsymbol{x}_k)_{y_k}}}{\sum_{j=1}^{|Y|} e^{f_\theta(\boldsymbol{x}_k)_j}}, \tag{3.3}$$

where Y is a discrete random variable over the class labels. $f_\theta(\boldsymbol{x}_k)_{y_k}$ and $f_\theta(\boldsymbol{x}_k)_j$ represent numerical values in the neural network's output vector at indices y_k and j, respectively. $p(Y = y_k | X = \boldsymbol{x}_k, \theta)$ is the predicted probability of the input vector $\boldsymbol{x}_k$ belonging to class y_k.

We want to tune the parameters of the network to increase the probability of correct labels given the inputs. A commonly used objective function for this problem is the categorical

cross-entropy[1] [79], defined as

$$J(\theta) = \frac{1}{N}\sum_{k=1}^{N} \log(p(Y = y_k | X = \boldsymbol{x}_k, \theta)), \tag{3.4}$$

where y_k is the true label and $\boldsymbol{x}_k$ the input vector associated with that label. By maximising this objective the network moves the predicted probability of the correct label closer to 1 for each of the N inputs in the training set. Therefore the higher the value $J(\theta)$ the better the network represents the data.

The standard method to tune the parameters of a neural network is called gradient descent, where one uses gradients to estimate the direction to update each parameter in order to minimise an objective function. In this work we instead use gradient ascent to maximise an objective function.

As demonstrated in Figure 3.2, gradient ascent is an iterative algorithm that updates a network's parameters to maximise some objective function by moving the parameter values in a direction dictated by first-order derivatives.

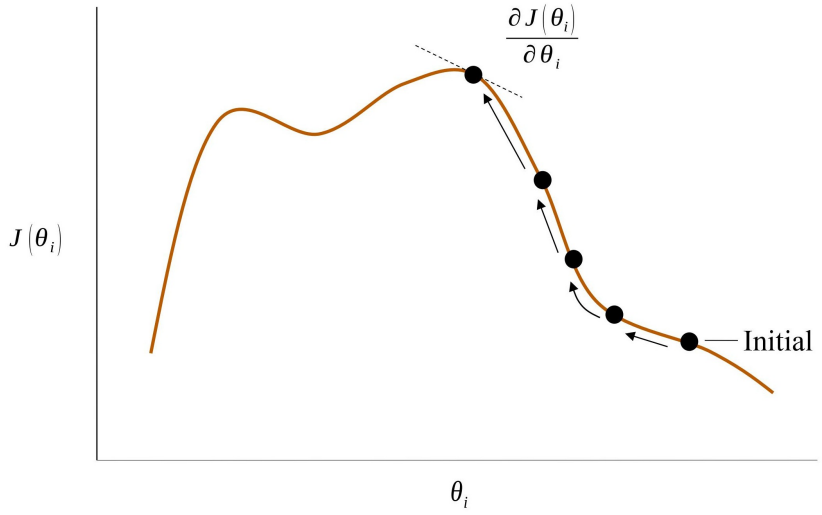


Figure 3.2: Depiction of the gradient ascent process that is used to iteratively tune a parameter θ_i to maximise some objective function $J(\theta_i)$.

The figure indicates how one parameter θ_i can be updated. We can update all the parameters of a network simultaneously by using the gradient vector $\frac{\partial J(\theta)}{\partial \theta}$ containing the first-order partial derivatives of the objective function with respect to each parameter. These derivatives can be calculated for a given set of values in θ by performing a forward pass through the network with all the training inputs, to determine the value of the objective function, and then a backwards pass to calculate the gradient and update the values in θ.

[1] Alternatively, some supervised learning problems use Kullback-Leibler divergence [23] which generally leads to a similar optimisation objective [18].

Due to the neural network being a composite function where every layer depends only on the previous layer, we can apply the chain rule to calculate gradients efficiently. We first calculate $\frac{\partial J(\theta)}{\partial \boldsymbol{y}}$, where $\boldsymbol{y}$ is the output vector from the final layer in the network. Next we can calculate the gradient of the last hidden layer with respect to the output, $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{h}_n}$, where $\boldsymbol{h}_n$ is the output of the last hidden layer. We can now recover the gradient of the objective function with respect to the last hidden layer $\boldsymbol{h}_n$,

$$\frac{\partial J(\theta)}{\partial \boldsymbol{h}_n} = \frac{\partial J(\theta)}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{h}_n}, \tag{3.5}$$

and compute the gradient of the objective function with respect to any hidden layer using the chain rule:

$$\frac{\partial J(\theta)}{\partial \boldsymbol{h}_k} = \frac{\partial J(\theta)}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{h}_n} \prod_{i=0}^{n-k-1} \frac{\partial \boldsymbol{h}_{n-i}}{\partial \boldsymbol{h}_{n-i-1}}, \tag{3.6}$$

where k is the index of a hidden layer.

Once we have the gradient vector $\frac{\partial J(\theta)}{\partial \theta}$, evaluated at the current set of parameter values, we have the direction in which to update each parameter in order to increase the objective function. Due to this being a first-order approximation we perform only small updates. The parameter update rule can be expressed as

$$\theta_{j+1} = \theta_j + \alpha \left. \frac{\partial J(\theta)}{\partial \theta} \right|_{\theta=\theta_j}, \tag{3.7}$$

where α is a small user defined scalar value called the learning rate, typically between 10^{-5} and 10^{-3}, that controls how fast a network's parameters should be updated. If we apply gradient ascent updates over many iterations the neural network might converge to a function that fits the training data reasonably well.

Feedforward neural networks allow us to learn a functional mapping from a training dataset of input-output pairs. Generally these input and output vectors are required to have fixed sizes. However, for our work we need functional mappings that take variable length inputs, for two reasons. Firstly, we need to somehow take into account observations over multiple time steps to generate an action. We therefore need our agent's network to have some form of memory of past observations. Secondly, the observations received at each time step might change in size depending on how many objects an agent can see. In the next two sections we present architectures that, when combined, can address these two issues.

3.1.2 Recurrent neural networks and LSTMs

In the game of soccer a player has to remember approximately where the ball and every other player is even if they cannot all be seen in the current observation. Recurrent neural networks (RNNs) offer one way of distilling memory of a sequence of inputs. An RNN can

be viewed as a feedforward neural network with an internal memory. At every time step an RNN takes as input a fixed vector and produces a fixed output vector, just like feedforward networks, but then also passes along internal information that can be used together with the input at a next time step.

An example RNN is shown in Figure 3.3. It receives as input both an external observation $\boldsymbol{x}_t$ and an internal vector.

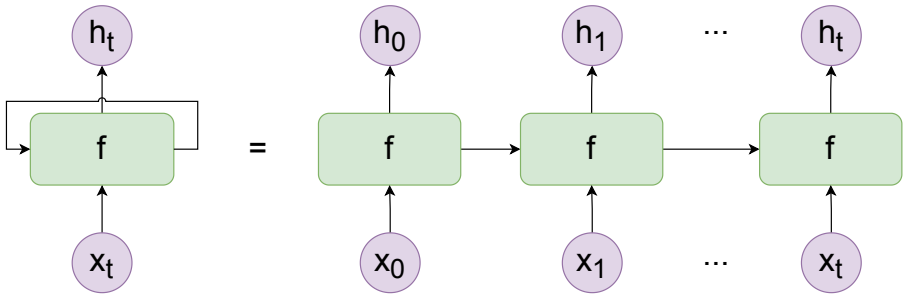


Figure 3.3: An RNN unrolled through time. The green blocks represent the network, the purple circles represent the input ($\boldsymbol{x}_t$) and output ($\boldsymbol{h}_t$) for a given time step. On the left of the figure is the RNN architecture with a looping arrow back into the network. On the right this loop is unrolled through time and one can see the internal state being passed through time. Image redrawn from Olah [45].

The internal vector allows the network to process and pass along information from previous time steps. The network itself can be a feedforward network that takes a concatenation of the external observation and internal state as input and produces an output. This output can then be split into the output at that current time step and the next internal state of the network. So at every time step the RNN takes a fixed sized input vector and produces a fixed sized output vector. However, if we now consider the entire unrolled RNN as our function approximator we have

$$\boldsymbol{h}_t = f_\theta(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t). \tag{3.8}$$

Notice that in equation 3.8 the size of the complete input $\boldsymbol{x}_0 : \boldsymbol{x}_t$ increases as the time step t increases, where in the case of feedforward neural networks they remained fixed (equation 3.2). Therefore, an RNN seems like a good solution to our problem where the agents require a memory of observations from previous time steps. However, in this setting gradient ascent can suffer from a problem called vanishing gradients, which makes it challenging for RNNs to learn long-term dependencies. Vanishing gradients is a known problem that can occur when applying the chain rule, which requires repeated multiplication, over many time steps. If many of the factors in this multiplication are small the gradient can become very weak or even zero in the case of numerical underflow. Therefore the gradient signal between the RNN output and inputs further back in time might be negligible when compared to inputs

closer to the output, which results in the network never learning long-term dependencies and only focusing on immediate observations.

To combat this problem, Hochreiter and Schmidhuber [28] proposed the long short-term memory (LSTM) unit. It uses trainable logic gates that control whether internal state information gets updated or not. The gates are constructed in such a way that important information can be preserved over many time steps to influence the current output, and this makes the gradient signals much stronger over longer sequences than in standard RNNs. The architecture of an LSTM unit is shown in Figure 3.4. Here the three network blocks represent computations performed over three time steps.

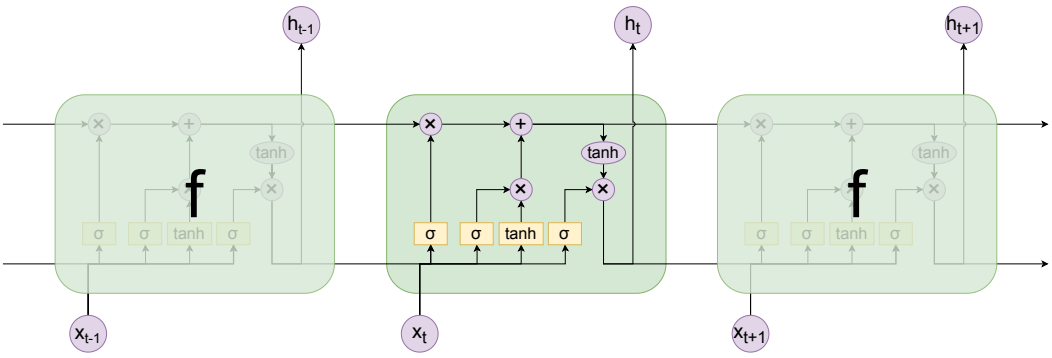


Figure 3.4: An LSTM unit unrolled over three time steps. Here f represents the LSTM unit with its internal computation, and again the purple circles represent the input ($\boldsymbol{x}_t$) and output ($\boldsymbol{h}_t$) vectors. Image redrawn from Olah [45].

The internal computation performed at every time step can be seen in the middle of Figure 3.4. Arrows that join represent concatenations and arrows that split represent duplication of the vector. The yellow blocks represent neural network layers and the internal purple circles represent point-wise operations. The top horizontal black line represents the internal state of the LSTM. The bottom black line represents the output of the LSTM which is also passed along to the next time step. The neural network layers (yellow blocks) have either a sigmoid activation or a tanh activation. The sigmoid activation is defined on elements z of an incoming vector as

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \tag{3.9}$$

with output between 0 and 1. It essentially determines whether to keep or destroy information captured by another vector, which allows them to act as continuous switches. The output of the first (left-most) sigmoid component in Figure 3.4 is multiplied by the current internal state, to determine whether to keep or forget information from the past. The bottom horizontal line, which also includes the new input vector, is then subjected to a tanh activation to transform and normalise it to values between -1 to 1. The second sigmoid

determines whether to add this new information to the state vector of the LSTM, thereby determining whether information stored in the LSTM's state should be kept, forgotten or updated. The last (right-most) sigmoid decides how much of the normalised internal state should be passed to the output and how much should be blocked. The top horizontal line has very few computations associated with it. If the first sigmoid is set to 1 and the second to 0, information can be stored indefinitely. This also means that the gradient signal degrades much slower and therefore relationships over much longer sequences can be learned, compared to standard RNNs.

While our agent can now process multiple observations to produce an action, these observations must still be of the same size. To allow for a variable observation size at each time step we next investigate the multi-headed attention mechanism.

3.1.3 Multi-headed attention

LSTMs can process sequences of observations, where the order of the observations over the different time steps influences the network's output. However, within the same time step an agent can observe a variable number of players. In this case, the order that these inputs (other players' information) are presented to the network should not influence its output action. We could use an LSTM, but we know that LSTMs are not input order invariant. To enable effective learning we want an architecture with an order invariant property, while also allowing for a variable input size. The multi-headed attention mechanism has exactly these properties.

Multi-headed attention was popularised by Vaswani et al. [70] when they showed that their network architecture could achieve state-of-the-art results on many natural language processing tasks. As show in Figure 3.5, the multi-headed attention layer takes as input three matrices ($\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$) representing the query, key and values.

The multi-headed attention mechanism is analogous to searching for information on the internet. The query might be some string of letters, and let us assume the search engine simply operates by comparing the user's query to all the websites it knows. Let us also assume each website has a set of keywords (key) associated with it. The search engine finds the closest matching website (value) to the query. An attention mechanism allows for the search to be performed on some set of values to find relevant information. This search process is done without taking the order of the values into account. Therefore one can add more values, which are captured in the $\boldsymbol{V}$ matrix, and the same network will still be able to produce outputs.

To describe how this search process works we define the scaled dot product attention mechanism as follows:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V}, \tag{3.10}$$

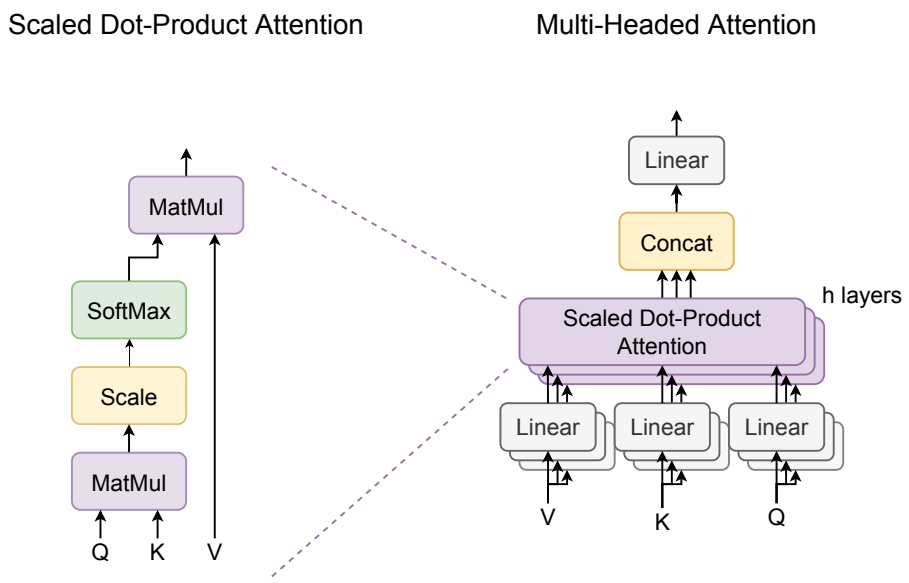


Figure 3.5: Graphical depiction of the computation performed in a multi-headed attention mechanism. The diagram on the left represents the computation performed inside the scaled dot-product attention operator. The diagram on the right depicts multiple parallel computations (one for each of the h heads) which get concatenated in the Concat layer. The bottom three linear layers are feedforward networks without activations and have learnable parameters which transform the input matrices separately for each head. Each head can therefore learn different representations of the input matrices. Image redrawn from Tamura [65].

where the softmax function is defined in equation 3.3. The softmax function normalises each row so that the elements are positive and the sum equals 1. Here $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ are matrices of sizes (d_q, d_k), (d_{vi}, d_k) and (d_{vi}, d_{vo}), respectively. The query matrix $\boldsymbol{Q}$ consists of d_q queries. Each query is a vector of dimension d_k. The $\boldsymbol{K}$ matrix has a set of keys which are associated with a set of values (websites in our search analogy). By calculating $\boldsymbol{Q}\boldsymbol{K}^T$ we are essentially trying to find scores which indicate how much each query matches each key. After dividing by $\sqrt{d_k}$ and normalising by a softmax we get weights indicating how much of each value should be represented in the output for each of the queries. This matrix $(\text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right))$ is referred to as the attention score matrix. Its rows are normalised, so by multiplying by $\boldsymbol{V}$ we get a weighted average for each query where the weights are determined by how closely the keys match.

Vaswani et al. [70] noted that the division by $\sqrt{d_k}$ prevents the dot product $(\boldsymbol{Q}\boldsymbol{K}^T)$ from becoming too large. When feeding large values into the softmax operator the gradients become very small. This may hinder training when using gradient ascent.

As can be seen in Figure 3.5, there are multiple copies of the scaled dot product operation. Each of these operations is referred to as a head, and for each of them the query, key and value matrices are first passed through separate linear feedforward layers. Different queries are therefore generated, and matched with different sets of key-value pairs. The outputs

with each head's scaled dot-product operations are concatenated and passed through a final linear feedforward layer to reduce the vector output dimension. The idea behind using multiple heads is that it allows the network to perform multiple queries on different value sets, which further increases the expressivity. The internal parameters of this multi-headed attention network are inside each of the linear feedforward layers. The feedforward layers are also applied row-wise to the $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ matrices and therefore the number of rows need not be fixed for a given parameterisation of the network.

One advantage of the multi-headed attention mechanism is that the row dimensionality of the value matrix does not influence the output shape of the attention function defined in equation 3.10. In our work, this dimension can change depending on how many other players the agent sees in a particular time step. The second advantage we also want is input order invariance. The order of the rows in the $\boldsymbol{K}$ and $\boldsymbol{V}$ matrices does not influence the output, as long as each row in $\boldsymbol{K}$ corresponds to the correct row in $\boldsymbol{V}$.

With network architectures that can possibly work well in our soccer environment, we need a learning algorithm that can find good parameters for the networks. Next we look at reinforcement learning as a possibility for this.

3.2 Reinforcement learning

Reinforcement learning is the process of learning through trial and error to act optimally in an environment. The basic premise is that an agent, or multiple agents, perform actions in a provided environment and get rewarded or punished for those actions. The environment provides numerical rewards to each agent for an indication of how well it is performing. In our case, the environment could represent a soccer game where a positive reward is given to a team of agents (players) when they score a goal, and a negative reward is given when a goal is scored against them. Another example of a reinforcement learning environment is a chess game, where a reward of 1 could be given for a win, 0 for a draw and -1 for a loss. An agent can play many games and learn to become proficient through maximising its reward. Reinforcement learning is a powerful framework for problems where agents can repeatedly and safely experiment within the environment. There also needs to be a clear reward structure that is positively correlated with how well the agents are achieving a certain objective.

In this work, we investigate methods to automatically derive high-performance strategies in the full 22 player soccer environment without manually encoding any significant soccer-specific knowledge. Our aim is that the methods would be applicable to other multi-agent competitive environments. We therefore need an algorithm to learn everything from low-level (basic) skills to high-level team strategies from only interacting with the environment. Reinforcement learning seems a natural fit for this problem as we can simulate soccer games with relative ease, as is done in the RoboCup simulation leagues. Soccer also has a well

defined reward structure. As will be discussed in Section 3.2.1, we divide our environment into a fixed number of time steps within which agents can perform actions and receive rewards. An obvious approach would be to just provide for each team at the final step a reward of 1, 0 or -1 depending on whether they won, drew or lost the game. This ensures that by maximising the rewards the team learns to win games, which requires them to improve both low-level and high-level strategies. In this work, we provide a slightly more expressive reward structure of 1 when the team scores a goal, -1 when they are scored against and 0 for the rest of the time. This reward structure provides a trade-off between a slight misalignment of the main objective (winning the game) and making it easier for the agents to learn, as more rewards are provided. These rewards directly incentivise goal scoring and defending behaviour. The slight misalignment is due to the agents attempting to maximise the number of goals they score in relation to their opponents (which can result in more offensive play even when they are ahead) instead of directly maximising their ability to win a game (which can result in more defensive play when they are ahead).

Before we derive learning algorithms we first need some understanding of the dynamics of reinforcement learning environments. In the next section we introduce these environmental dynamics.

3.2.1 Markov decision processes

In reinforcement learning one usually assumes or constructs the environment in such a way that it can be modelled as a Markov decision process (MDP). An MDP is a mathematical framework for sequential, discrete-time decision making in environments that can be partially stochastic and partially under the control of the decision maker.

For every time step, we define a state value $s_t \in S$ and a set of possible actions $A(s_t)$ from which the agent can select an action a_t to take. In this work, we assume that all states have the same action space, such that $A(s_t) = A$, i.e. the action space is independent from the agent's current state. The states and actions can be discrete or continuous in nature. For discrete states and actions there exist $|S|$ possible states and $|A|$ possible actions. In the continuous case, the states and actions can be real numbers (or vectors of real numbers). In this work, we use continuous states and actions, which are generally more expressive and allow for more precise decision making. It is however usually more difficult to learn in environments with continuous actions as a reinforcement learning algorithm has to deal with an infinite number of possible actions. In the continuous case, states and actions can be represented by one or multiple scalar values.

At each time step t, an agent (or team of agents) can select an action. The state then gets updated using a transition probability distribution over the next state s_{t+1} given the current state s_t and the action a_t taken. s_0 and a_0 represent the starting state in the environment and the first action made by the agent. The initial state can either be fixed or randomly sampled from some initial state distribution $p(s_0)$. In an MDP we assume the

Markov property, which specifies that the next state in an environment is independent of the previous states, given that the current state is known, i.e.

$$p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = p(s_{t+1}|s_t, a_t). \tag{3.11}$$

We use a lowercase p indicating that this could be either a discrete probability over a finite number of states or a probability density function over continuous states.

We further define the reward r_t that the agent receives in each state after it performs an action, with $r_t = r(s_t, a_t) \in R$, and $R : S \times A \to [r_{\min}, r_{\max}]$. Some literature assigns the reward for the previous state action pair only in the next time step $(t+1)$, but for notational simplicity we assign it at the current time step. Furthermore, we assume that the reward function is deterministic in nature given a state-action pair. Although this reward is only dependent on the current state-action pair, the current state is dependent on the previous state, as seen in equation 3.11. Therefore actions taken earlier in the sequence generally have an influence on future rewards. The agent must choose its actions carefully at every time step even though it might not experience any immediate reward from it. Note that we use the shorthand notation $p(S_{t+1} = s_{t+1}|S_t = s_t, A_t = a_t) = p(s_{t+1}|s_t, a_t)$, where S_{t+1}, S_t and A_t are random variables, and s_{t+1}, s_t and a_t are instantiations of those random variables.

In the reinforcement learning framework a distinction is made between an environment and an agent (or multiple agents) acting in the environment. We first consider the single-agent setting and will generalise to the multi-agent setting in Section 3.4.

Figure 3.6 illustrates the information flow between the agent and the environment.

At every step the agent receives an observation from the environment. This observation, o_t, can be the full internal state of the environment, in which case $o_t = s_t$, or a partial observation of the environment (e.g. a robot's visual observation in the real world). The agent then performs some computation on this information and produces an action a_t. The function that takes as input an observation or sequence of observations and produces an

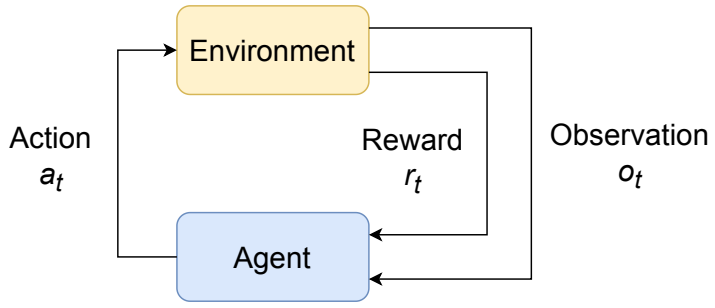


Figure 3.6: An illustration of how a reinforcement learning agent interacts with an environment.

action, is referred to as the agent's policy π. The agent also receives a reward r_t from the environment. As mentioned above, t indicates the current time step in the environment which increments from 0 up until the experiment terminates (if it terminates at all). We use discrete time steps, which is generally acceptable even in environments with continuous time, such as in real world robotic tasks, as long as the intervals between the discrete steps are sufficiently small.

We now define an objective for the agent, which is to maximise the expected cumulative reward. We generally consider two types of environments. The first focuses on episodic tasks, where there is a terminal state, e.g. a game of soccer that is completed after a finite number of time steps. One complete run through such a task environment is called an episode, and episodes are assumed to be independent of each other as the states get reset after each episode's terminal time step. The agent must try to maximise an objective function defined as

$$v_\pi = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=0}^{T} r_t \right], \tag{3.12}$$

where v_π represents the expected cumulative reward of the agent over an episode. $\tau \sim \pi_\theta(\tau)$ represents the probability of sampling a trajectory given the policy's parameters θ. Here τ represents the trajectory $s_0, a_0, \ldots, s_T, a_T$, and $r_t = r(s_t, a_t)$ represents the reward the agent receives at time step t. We bound r_t to be in $[r_{\min}, r_{\max}]$. T represents the final time step in which the agent can take an action and receive a reward.

The second type of environment is continuing task environments that do not have an end or terminal state. In these environments we cannot simply sum up the expected reward to infinity. If the rewards do not decrease in magnitude the agent has no incentive to learn optimal actions. We therefore need to add a discount factor to the rewards, such that

$$v_\pi = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right], \tag{3.13}$$

where γ is a real number in $[0, 1)$. If the rewards are bounded, we can show that the total reward v_π remains bounded by

$$\frac{r_{\min}}{1-\gamma} \leq v_\pi \leq \frac{r_{\max}}{1-\gamma}. \tag{3.14}$$

Therefore, every action an agent takes becomes important again if an agent wants to maximise this cumulative reward.

In this work, we focus on environments with episodic tasks. This is because our soccer environment has a clear terminal state that is always reached when the game concludes. Episodic task environments do not explicitly require a discount factor (γ) and therefore we do not include it in most of our derivations. However, we do make use of equation 3.13

in the design of our main reinforcement learning algorithm as it helps when training our agents, as discussed in more detail in Section 3.3.5.

For episodic task environments the probability of a trajectory is defined as [34]

$$p(\tau) = p(s_0, a_0, s_1, a_1, \ldots, s_T, a_T) = p(s_0) \prod_{t=0}^{T} p(a_t|s_t, \theta) p(s_{t+1}|s_t, a_t). \tag{3.15}$$

As mentioned above, τ represents the trajectory followed in a given episode. Each agent has a policy π, parameterised by θ, which it uses to act in the environment. The policy is usually written as

$$\pi_\theta(a_t|s_t) = p(a_t|s_t, \theta). \tag{3.16}$$

The policy can be represented in different ways, from a simple table lookup to a deep neural network. The policy can also be written in terms of a function that takes in the state and outputs an action:

$$a_t = \pi_\theta(s_t). \tag{3.17}$$

Following from equation 3.12, the objective of an agent in an episodic task environment is to find an optimal policy as defined through its optimal set of parameters

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=0}^{T} r_t \right]. \tag{3.18}$$

We therefore aim to find an optimal set of parameters θ^* which results in the agent achieving the highest expected cumulative reward. The optimal policy is written as $\pi^* = \pi_{\theta^*}$.

Next we define the value function as

$$v_\pi(s_t) = \mathbb{E}_{\eta \sim \pi_\theta(\eta)} \left[\sum_{k=t}^{T} r_k \right], \tag{3.19}$$

which is the expected cumulative reward an agent will receive from state s_t up until the terminal state. π_θ represents the probability of sampling η given the current policy π, where η is the random variables $a_t, s_{t+1}, a_{t+1}, \ldots, s_T, a_T$. The value function can also be defined recursively as

$$v_\pi(s_t) = \mathbb{E}_{\eta \sim \pi_\theta(\eta)} \left[G_t \right] = \mathbb{E}_{\eta \sim \pi_\theta(\eta)} \left[r_t + G_{t+1} \right], \tag{3.20}$$

where G_t is cumulative sum of rewards from time step t onward, defined as

$$G_t = \sum_{k=t}^{T} r_k. \tag{3.21}$$

Estimates of these value functions are used throughout reinforcement learning. In Section 3.3.2, we will derive a learning algorithm that explicitly models both the policy π and the value function $v_\pi(s_t)$ for faster convergence.

3.2.2 Partially observable Markov decision processes

In many environments, including our soccer environment, agents do not have access to the full underlying state information and receive only partial information from the environment. In our custom soccer environment an agent can observe only what is in front of it and has to remember or estimate what is happening behind it. In this case, the agent operates in a partially observable Markov decision process (POMDP). Due to the environment being partially observable, agents can use internal memory to remember information from previous observations. Therefore, in a POMDP agents select actions as follows:

$$a_t = \pi(o_t, o_{t-1}, a_{t-1}, \ldots, o_0, a_0), \tag{3.22}$$

where $o_t \in O(S_t)$ is the observation received by an agent at time step t which typically contains less information than what is present in s_t. As we discussed in Section 3.1.3, the size of the observation might be different at every time step, as the number of other players an agent observes depends on where the agent is looking. Notice that we also include previous actions as input to the policy. These are sometimes useful in environments where observations might be ambiguous, e.g. a vision based robot exploring in a dark room. Even though it might not be able to exactly determine where it is currently located, based on its vision, it can still estimate its position based on previous movement actions. We can also write the policy in equation 3.22 as a probability distribution, with internal parameters θ, as

$$p(a_t|o_t, o_{t-1}, a_{t-1}, \ldots, o_0, a_0, \theta) = p(a_t|o_{0:t}, a_{0:t-1}, \theta) = \pi_\theta(a_t|o_{0:t}, a_{0:t-1}). \tag{3.23}$$

3.2.3 Deep reinforcement learning

According to equation 3.22, we need some function that takes as input previous observations and actions and produces a new action as output. In the simple case of fully observable environments ($o_t = s_t$) with only a limited number of states, this function can be a table with each possible state as a row and the corresponding distribution over actions in columns. In the case of discrete actions the columns contain probabilities. In the continuous case the columns might contain the mean and standard deviation of a Gaussian distribution

over actions. Learning in these scenarios is equivalent to updating the values in this table according to some update rule, as will be described in Sections 3.3.1 and 3.3.3.

There is a clear problem with the table approach when working with larger environments. Using a table requires that each state needs to be visited at least once, so that the agent experiences rewards for each state and can update its policy accordingly. Real environments often have state spaces for which exhaustive exploration is intractable. The state might also be continuous, requiring an almost infinite number of rows in the table. If agents cannot explore every possible state we need them to generalise experience from encountered states to a potentially large set of states not encountered. We therefore need a function approximator that can learn a compressed yet informative representation of state-action data it receives, in order to generalise to new states. Neural networks (as discussed in Section 3.1) offer this kind of function approximation, and so it seems fitting to try and combine reinforcement learning with neural networks. Indeed, this combination is in large part responsible for recent progress in reinforcement learning.

We can provide the state information to a feedforward neural network, or observation-action sequences to a recurrent neural network, which can be trained to produce an action as output. Because standard neural networks are deterministic in nature we let them output probability scores over a specified action set, and sample from this distribution to generate an action. In the discrete case, one can use the softmax operator, defined in equation 3.3.

For continuous actions a common modelling choice is the Gaussian distribution. One may start by creating a neural network function approximator that produces estimates for the mean μ_t and standard deviation σ_t of the action distribution at time step t. To prevent the neural network from outputting zero or negative numbers for the standard deviation, one may use an appropriate activation function like ReLU or sigmoid. An action can now be sampled using

$$p(A_t = a_t|S_t) = \frac{e^{-0.5\left(\frac{a_t-\mu_t}{\sigma_t}\right)^2}}{\sigma_t\sqrt{2\pi_n}}, \tag{3.24}$$

where π_n is not the policy but the number $\pi_n \approx 3.1416$. One can therefore use neural networks to produce deterministic output vectors and then sample using these vectors to select actions. This results in policies being stochastic in nature.

For both of the above examples the neural network can take in either a continuous state vector directly for a continuous state distribution or a discrete state input as a one-hot encoded vector. One-hot encoding entails creating an input vector of size $|S|$, and setting all entries except the current state to zero. Using a one-hot encoding for categorical variables (like discrete states) generally works well as it makes no correlation assumptions between the states and makes it easy for the neurons in the first hidden layer to specialise to different states.

Now that we have a function approximator we need a reinforcement learning algorithm in

order to find parameters that maximise the cumulative reward function defined in equation 3.18. We discuss our first reinforcement learning algorithm in the next section.

3.3 Single-agent reinforcement learning

We proceed to derive the single-agent reinforcement learning algorithms used in this work. In Section 3.4, we will investigate additional tools needed to enable these algorithms to function properly in the multi-agent setting. The following single-agent algorithm is the first step in creating our main algorithm. It attempts to directly optimise the policy using experience from the environment.

3.3.1 Policy gradient method

As discussed, the aim of an agent is to maximise the cumulative reward it expects to receive in an episode. As function approximators, neural networks can be tuned to achieve this aim. We also have gradient ascent, with which to tune the network's parameters given that we have a differentiable objective function with respect to the parameters. The maximisation problem can be written as

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{t=0}^{T} r_t\right] = \arg\max_\theta J(\theta). \tag{3.25}$$

The objective function is

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{t=0}^{T} r_t\right], \tag{3.26}$$

i.e. the cumulative reward the agent receives as a function of the policy parameters. Recall that $r_t = r(s_t, a_t)$ is the reward received at time step t when performing action a_t in state s_t.

The gradient ascent update rule to be derived has the form

$$\theta_{j+1} = \theta_j + \alpha\nabla_\theta J(\theta_j), \tag{3.27}$$

where $\nabla_\theta J(\cdot)$ is the gradient of the objective function with respect to the parameters θ. To simplify the notation we write

$$r(\tau) = \sum_{t=0}^{T} r_t, \tag{3.28}$$

where τ is the trajectory. The objective function becomes [39]

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[r(\tau)] = \int_\tau \pi_\theta(\tau) r(\tau)\ d\tau, \tag{3.29}$$

with gradient

$$\nabla_\theta J(\theta) = \int_\tau \nabla_\theta \pi_\theta(\tau) r(\tau)\ d\tau. \tag{3.30}$$

We can manipulate this expression to turn the integral back into an expectation. To this end, we make use of the log-derivative trick which states that

$$\nabla_x \log f(x) = \frac{\nabla_x f(x)}{f(x)}, \tag{3.31}$$

and therefore implies that

$$\nabla_\theta \pi_\theta(\tau) = \pi_\theta(\tau)\frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau). \tag{3.32}$$

We now expand the objective function in equation 3.30 as

$$\nabla_\theta J(\theta) = \int_\tau \nabla_\theta \pi_\theta(\tau) r(\tau)\ d\tau = \int_\tau \pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau) r(\tau)\ d\tau = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]. \tag{3.33}$$

As stated in equation 3.15, the probability distribution over τ is defined as

$$\pi_\theta(\tau) = p_\theta(\tau) = p_\theta(s_0, a_0, \ldots, s_T, a_T) = p(s_0)\prod_{t=0}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t). \tag{3.34}$$

Taking the log on both sides yields

$$\log \pi_\theta(\tau) = \log p(s_0) + \sum_{t=0}^{T} \left(\log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)\right), \tag{3.35}$$

and the gradient becomes

$$\nabla_\theta \log \pi_\theta(\tau) = \nabla_\theta \left[\log p(s_1) + \sum_{t=0}^{T} \left(\log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)\right)\right] \tag{3.36}$$

$$= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t). \tag{3.37}$$

We can now substitute equations 3.28 and 3.37 into equation 3.33 to get

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\right)\left(\sum_{t=0}^{T} r_t\right)\right] \tag{3.38}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{k=0}^{T} r_k\right], \tag{3.39}$$

where we moved the summation over rewards into the summation of $\nabla_\theta \log \pi_\theta(a_t|s_t)$.

The motivation behind the derivation up until this point was to get a formulation for the gradient that can be evaluated, i.e. we are able to compute what is inside the expectation in equation 3.39. The steps in equations 3.35 to 3.37 show that we can remove the dependency of knowing the model of the underlying MDP when computing gradients. The expectation in equation 3.39 can be evaluated by means of Monte Carlo approximation, which relies on the following equality:

$$\lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N} f(x_i)_{x_i\sim p(x)} = \mathbb{E}[f(x)]. \tag{3.40}$$

That is to say, if we sample from $p(x)$ a large but finite number of times we can approximate the expected value of $f(x)$. Therefore equation 3.39 can be approximated as

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}, s_{i,t})\sum_{k=0}^{T} r(s_{i,k}, a_{i,k})\right], \tag{3.41}$$

where $s_{i,t}$ and $a_{i,t}$ represent the state and action, respectively, that was recorded in datapoint i for time step t. The same holds for $s_{i,k}$ and $a_{i,k}$. Equation 3.41 provides a simple rule for updating the policy parameters directly using experience sampled from the environment. It is important to note that this first-order gradient is valid only for a given instantiation of the parameters θ with which the N trajectories were sampled. After performing the update rule in equation 3.27, we would need to collect N new trajectories, sampled under the new network parameters, before performing another update.

Updating the policy in this way is often referred to as a policy gradient method, as we directly approximate the gradient of the policy with respect to the objective. However a remaining problem is that the approximation in equation 3.41 can have high variance. This is because for each trajectory that is sampled, $T + 1$ samples are taken from both the policy and the environment transition function. These are usually stochastic in nature and may therefore introduce high variance in the sampled trajectories, and a large number of experiences need to be collected for every gradient ascent update in order to accurately approximate the expectation. According to equation 3.41, the cumulative reward is used to determine how much to update the policy towards the action taken in the environment. Therefore, actions that produced higher rewards become more likely in the future.

In the above, we use the complete cumulative reward to update each action probability. However, the policy at a given time step cannot influence previous rewards in that same episode. To remove those previous rewards and the extra variance they introduce, we use the expected grad-log-prob lemma. From probability theory we know that

$$\int p_\theta(x)\,dx = 1. \tag{3.42}$$

Taking the gradient on both sides gives

$$\nabla_\theta \int p_\theta(x)\,dx = 0, \tag{3.43}$$

and, using the log derivative trick from equation 3.31, we have

$$\begin{aligned} 0 &= \nabla_\theta \int p_\theta(x)\,dx && (3.44)\\ &= \int \nabla_\theta p_\theta(x)\,dx && (3.45)\\ &= \int p_\theta(x)\nabla_\theta \log p_\theta(x)\,dx && (3.46)\\ &= \mathbb{E}_{x\sim p_\theta(x)}\left[\nabla_\theta \log p_\theta(x)\right]. && (3.47) \end{aligned}$$

We now expand equation 3.39 as follows:

$$\begin{aligned} &\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{k=0}^{T} r_k\right] && (3.48)\\ &= \sum_{t=0}^{T}\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{k=0}^{T} r_k\right] && (3.49)\\ &= \sum_{t=0}^{T}\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{k=0}^{t-1} r_k\right] + \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{k=t}^{T} r_k\right] && (3.50)\\ &= \sum_{t=0}^{T}\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\right]\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{k=0}^{t-1} r_k\right] + \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{k=t}^{T} r_k\right] && (3.51)\\ &= \sum_{t=0}^{T}\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{k=t}^{T} r_k\right], && (3.52) \end{aligned}$$

where $r_k = r(s_k, a_k)$. The simplification in the second last line is possible because in the t^{th} term of the outer sum, $\nabla_\theta \log \pi_\theta(a_t|s_t)$ and $\sum_{k=0}^{t-1} r_k$ are independent given that we know the state s_t [2]. The first term in the second last line can be discarded as equation 3.47 shows that $\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\right] = 0$.

By combining equations 3.52 and 3.40 we find

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}, s_{i,t}) \sum_{k=t}^{T} r(s_{i,k}, a_{i,k}) \right]. \tag{3.53}$$

This gradient approximation forms the basis of the classic policy gradient method called REINFORCE [72].

3.3.2 REINFORCE with baselines

To further reduce the variance in the estimates of $\nabla_\theta J(\theta)$ we can subtract the expected value for a particular state from the cumulative reward. Using the same derivation process as in equations 3.48 to 3.52, we can show that

$$\sum_{t=0}^{T} \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{k=t}^{T} r_k - b(s_t) \right) \right] \tag{3.54}$$

$$= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{k=0}^{T} r_k \right], \tag{3.55}$$

where b is a deterministic function that takes the state s_t as input and produces a scalar output. The final expectation remains the same, but this form allows for better control over the variance. By setting $b(s_t)$ to be the expected cumulative reward we can eliminate the offset. This reduces the variance in the estimates significantly, which speeds up convergence. One way of estimating $b(s_t)$ is by another neural network, $v_w(s_t)$. The policy network $\pi_\theta(a_t|s_t)$ and the so-called critic network $v_w(s_t)$ can be trained in parallel. The critic network attempts to predict the expected cumulative reward for every state given the current policy. The policy in turn attempts to produce actions that maximise the expected cumulative reward, while using the critic to reduce the variance in its network updates. This forms the basic premise of actor-critic methods, as discuss in the next section.

3.3.3 Actor-critic architecture

While policy gradient methods have straightforward update rules, they usually have slow convergence times during training and require a large number of time steps per update to counteract the high variance in the gradient estimates. As mentioned in Section 3.3.2, it is possible to include a critic network that reduces the variance in these updates, and the best performing reinforcement learning algorithms usually train both policy and critic networks for faster convergence.

Actor-critic methods, illustrated in Figure 3.7, work by learning both a value estimator (critic) that predicts the expected future reward, as well as a probability distribution (policy) over actions that the agent should take. The critic is updated by estimating the expected

rewards received in the environment, such that

$$\nabla_w I(w) \approx -\frac{1}{N}\sum_{i=1}^{N}\nabla_w \sum_{t=0}^{T}\left(\sum_{k=t}^{T} r(s_{i,k}, a_{i,k}) - v_w(s_{i,t})\right)^2, \tag{3.56}$$

where w contains the parameters of the critic $v_w(\cdot)$. Here $I(\cdot)$ represents the critic's objective function. The critic is usually represented by a neural network that takes a state as input and outputs a scalar value indicating the expected cumulative reward from this state onward. Therefore maximising this objective requires that the critic becomes more accurate at predicting the current policy's expected cumulative reward.

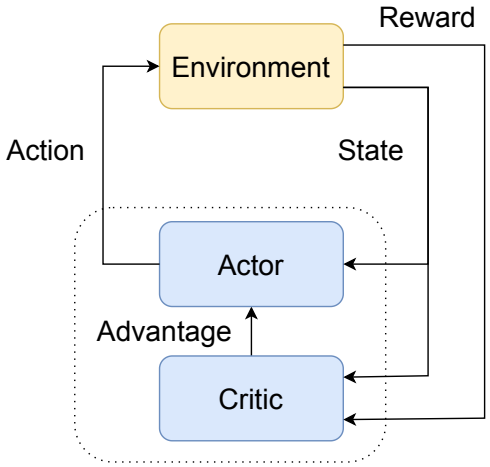


Figure 3.7: Illustration of the actor-critic interaction with the environment. The critic takes as input the state and reward information and updates its internal parameters to better predict the expected cumulative reward for the given policy. The critic passes an advantage value (difference between the actual received cumulative reward and the predicted cumulative reward) to the actor which indicates how good the given actions were. The actor takes as input state (or observational) information and produces an action which is passed to the environment. The actor updates its parameters using the advantage value.

The gradient of the policy's objective function can be derived by combining equations 3.40 and 3.54, and by swapping $b(s_{i,t})$ for $v_w(s_{i,t})$:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\left[\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{k=t}^{T} r(s_{i,k}, a_{i,k}) - v_w(s_{i,t})\right)\right]. \tag{3.57}$$

The difference between the actual sum of rewards and estimated sum of rewards is often referred to as the advantage value, and this actor-critic method is often referred to as the advantage actor-critic (A2C) method, given in Algorithm 1. In the algorithm, G_t is an episode's cumulative reward from time step t and must be sampled using the current policy. The networks can be updated based on this experience. A slight improvement over

Algorithm 1 Advantage actor-critic algorithm.

Require: π, θ, v, w, α_θ, α_w ▷ α_θ and α_w are step size parameters greater than zero.

for each iteration **do**
  Generate an episode τ_i following π_θ.
  for t **in** environment steps (in τ_i) **do**
    $G_t \leftarrow \sum_{k=t}^{T} r(s_{i,t}, a_{i,t})$ ▷ Cumulative reward from time step t onward.
    $\hat{A}_t \leftarrow G_t - v_w(s_{i,t})$ ▷ Advantage estimate.
    $w \leftarrow w - \alpha_w \hat{A}_t^2$ ▷ Update the critic.
    $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \hat{A}_t$ ▷ Update the policy.
  end for
end for
return θ, w

this algorithm is to work with batched data, where many episodes are first collected. The networks are then updated over a batch of experience as in equations 3.56 and 3.57. This averaging over a batch of data can help to reduce variance in the updates.

We next discuss an improvement which allows for faster training by parallelising different components of A2C.

3.3.4 Distributed advantage actor-critic

A common enhancement of the A2C method is to have more than one experience generator running in parallel. This allows actor-critic algorithms to better utilise different processors on modern computing hardware. Here the word "distributed" means that the experience generators and learner are now running in parallel and not synchronous with each other.

As illustrated in Figure 3.8, there are multiple workers interacting with different copies of the same environments. The key insight is that collecting experience from the environment is usually the main computational bottleneck in the batched A2C method. However, each episode can technically be generated independent from the others, as updates are performed over batched data. The distributed A2C method therefore has n independently running processes called workers with each having their own copy of the policy and critic networks. They generate trajectories and send them to a learner process which runs parallel to the workers and uses the generated trajectories to perform updates to both networks. The workers frequently request copies of the latest learner networks so that the experience they generate remains relevant to the optimisation procedure.

In practice, the workers are usually run on central processing units (CPUs), which are good at executing sequential programs such as agents interacting with an environment. The learner usually runs on a graphical processing unit (GPU) or a tensor processing unit (TPU) which can perform parallel computations, such as working with batched data, much faster than a CPU. The distributed A2C method has much faster throughput than A2C and agents learn faster in this setup.

In this work we make use of our multi-agent reinforcement learning framework called

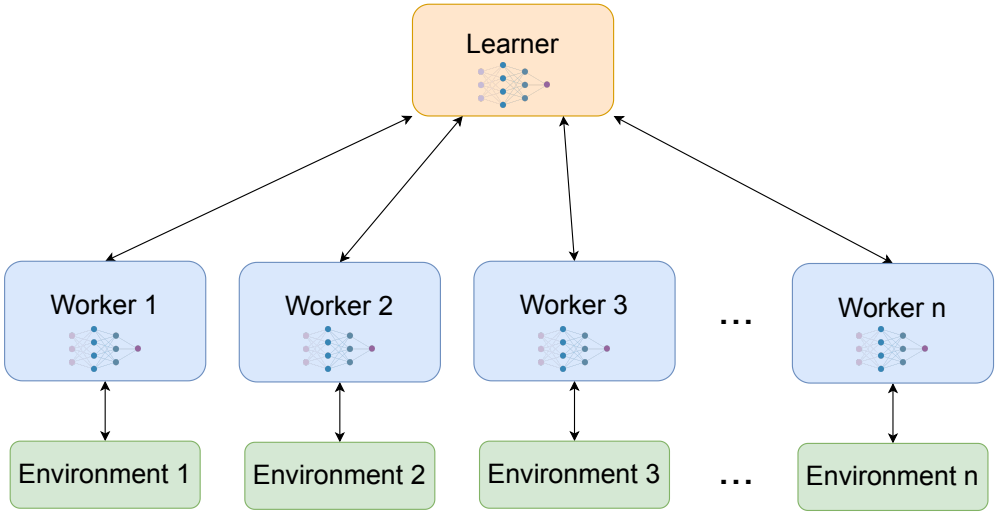


Figure 3.8: A graphical illustration of the multiple workers and learner in the distributed A2C setup. At the bottom of the image n workers can be seen interacting with their own copies of the environment in parallel. Each worker has their own copy of the learner's networks which they sync regularly. Image redrawn from Anagolum [8].

Mava [47], further discussed in Section 5.3. This framework also has workers executing in parallel, together with a learner for faster training. In the next section, we derive the main learning algorithm used in this work.

3.3.5 Proximal policy optimisation

A popular reinforcement learning algorithm by Schulman et al. [57] is called proximal policy optimisation (PPO). It incorporates the same ideas as the actor-critic method, but is designed to be more stable by limiting how much an agent can update the policy's parameters per datapoint. Policy based methods approximate the expected cumulative reward of a given state to determine in which direction to update the policy. Without a value function, an approximate method uses the actual cumulative reward of the episode, as in equation 3.41, which may introduce a significant amount of additional noise in the estimates. Training policy based methods are therefore slow compared to actor-critic methods. As discussed in Section 3.3.3, actor-critic methods allow the policy to be updated based on the agent's own value estimate, while updating the agent's value prediction (critic) to be closer to the observed rewards, and PPO also follows this design paradigm. A problem that actor-critic methods in general still face is called policy collapse, which can occur when the agent's policy updates too quickly for the critic network to model accurately. The critic's expected reward estimates can therefore become outdated which in turn causes the policy's updates to become more noisy, to the point that the policy might rapidly deteriorate in performance. PPO tries to combat this problem by directly implementing a clipping function to limit how

much the optimiser can update the parameters per batch of trajectories. This clipping functionality is especially useful in the multi-agent setting where other agents are also sampling stochastic actions and updating their policies over time.

As before, the policy is denoted by $\pi_\theta(a_t|s_t)$, and the predicted value of being in the given state is $v_w(s_t)$. We can now derive our main reinforcement learning algorithm and reintroduce the reward discount factor γ, as described in Section 3.2.1. This factor, which is a real number fixed between 0 and 1, determines the reward horizon on which an agent focuses. The true value function can be defined as

$$v_\pi(s_t) = \mathbb{E}_{\eta\sim\pi_\theta(\eta)}\left[\sum_{k=t}^{T}\gamma^{k-t}r_k\right], \tag{3.58}$$

which is the expected discounted cumulative reward when the current policy π_θ is used. Each future reward is discounted by γ^{k-t}, where $k-t$ is the number of steps into the future in which the agent receives that reward, to incentivise the maximisation of immediate rewards more than future rewards. If γ is close to 1 the policy will favour long term rewards, while a discount factor close or equal to 0 will make the policy favour immediate rewards. It may seem that a larger discount factor would be preferable, however with a larger discount factor the agent has to take the entire episode's rewards into consideration, in equal measure. This makes the reward estimates less reliable and may slow down the learning process. In many episodes each individual reward might be correlated with rewards received after it. Therefore, maximising for short term rewards might align well with maximising the total reward, while introducing less noise to the learning process. In robot soccer, for example, it is almost as good to optimise the scoring of goals (short/medium term rewards) over winning a game (long term reward). To update the agent's neural network, given new rewards and observations, an objective function must first be defined.

The PPO objective function is formed from three different objectives, and is defined as

$$L(\theta, w) = \sum_{t=0}^{T}\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[L_t^{clip}(\theta) - c_1L_t^c(w) + c_2S\left[\pi_\theta(A_t|S_t = s_t)\right]\right], \tag{3.59}$$

where the expected value can again be approximated by taking the average over a batch of training experiences, as specified in equation 3.40. $L_t^{clip}(\theta)$ represents the clipped objective of the policy function. $L_t^c(w)$ represents the objective of the critic, with importance weight c_1. $S\left[\pi_\theta(A_t|S_t = s_t)\right]$ represents the entropy of the policy output, with importance weight c_2. The entropy objective is used to encourage the agent to explore, by making less likely actions more likely. For discrete action spaces the entropy objective is defined as

$$S\left[\pi_\theta(A_t|S_t = s_t)\right] = \sum_{k=1}^{|A|}\pi_\theta(A_t = a_k|S_t = s_t)\log(\pi_\theta(A_t = a_k|S_t = s_t)), \tag{3.60}$$

where $|A|$ is the number of discrete actions available to the agent. In Section 3.3.6, we will discuss calculating the entropy term for a clipped Gaussian distribution over a continuous action space.

The critic's objective can be calculated as

$$L_t^c(w) = (G_t - v_w(s_t))^2, \tag{3.61}$$

where G_t is the cumulative reward experienced from time step t onward and $v_w(s_t)$ the value estimated by the critic network. Minimising this loss pushes the critic network's output towards the true cumulative expected reward for the given policy. The target value can be estimated for each environment step as

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k, \tag{3.62}$$

using the rewards experienced through the environment run. As can be seen by the summation over all the rewards, this is a Monte Carlo version of the PPO algorithm. The original PPO implementation uses a truncated generalised advantage estimation [56], which generally results in faster training. However, we found Monte Carlo sampling to work better in our multi-agent setting where the value estimator needs to adapt faster than in the single-agent case. The reduction in bias in the value estimation seems to make up for the increase in noise. To calculate the $L_t^{clip}(\theta)$ objective, two intermediate values must first be calculated. The first is

$$\hat{A}_t = G_t - v_w(s_t), \tag{3.63}$$

which is the same advantage estimate defined in the advantage actor-critic algorithm (Algorithm 1) with the additional discount factor. If $\hat{A}_t$ is positive, action a_t was better than the critic expected (higher cumulative discounted reward). If it is negative the cumulative discounted reward was lower than expected. The probability of action a_t can now be increased if $\hat{A}_t$ is positive, or decreased if it is negative. The second value needed to define L_t^{clip} is

$$\hat{r_t} = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta\text{old}}(a_t|s_t)}, \tag{3.64}$$

which is the probability ratio of the updated network over the old network for taking action a_t at time step t. In the continuous case, this is just the ratio of the density function outputs as the probability densities for a specific action is zero. This ratio $\hat{r}_t$ should not be confused with r_t, which is the reward received at time step t. If $\hat{r}_t$ is larger than 1 the action is more likely, and if $\hat{r}_t$ is smaller than 1 the action is less likely. The use of a fraction allows the optimiser to focus equally on correcting low and high probabilities.

For the standard actor-critic architectures, as shown in equation 3.54, the gradient of

the objective function for the policy would be

$$\nabla_\theta J(\theta) \approx \sum_{t=0}^{T} \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t]. \tag{3.65}$$

If $\hat{A}_t$ is greater than 0 (the agent did better than expected), the optimiser will try to increase the probability of the action the agent took. If $\hat{A}_t$ is less than 0 (the agent did worse than expected), the optimiser will try to decrease the probability of this action in the future. Using equation 3.31, the gradient can be written as

$$\nabla_\theta J(\theta) \approx \sum_{t=0}^{T} \mathbb{E}_{\tau\sim\pi_\theta(\tau)} \left[\frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} \hat{A}_t \right]. \tag{3.66}$$

One problem with the expression $\frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}\hat{A}_t$ inside the expectation, is that the network may sometimes update an action probability by too much, which can lead to policy collapse. The collected experience predicts only the expected reward for the actions that were taken in the environment, and if we update the policy actions to be too far from those sampled actions the reward signal can become unreliable. In an effort to avoid this problem, PPO limits the amount by which the optimiser can update each action's probability, by means of a clipping function. The PPO policy objective is defined as

$$L_t^{clip}(\theta) = \min(\hat{r}_t\hat{A}_t, \text{clip}(\hat{r}_t, 1-\epsilon, 1+\epsilon)\hat{A}_t), \tag{3.67}$$

where the clip function clips $\hat{r}_t$ if its value is below $1-\epsilon$ or above $1+\epsilon$. The threshold ϵ determines by what factor an output action probability may change. In the original paper, Schulman et al. [57] propose an ϵ of 0.2.

The policy objective function is plotted in Figure 3.9. The PPO algorithm essentially allows the optimiser to change the policy only within a certain range of the original policy, which further helps reduce the chances of policy collapse.

In our own simulation environment, as in the RoboCup simulators, we will provide partial observations to the agents. The critic still relies on full state information as it is not used during evaluation. After training, our agents will not need the critic anymore and can only operate with partial observations of the full state. We can therefore replace all the above state based inputs with the policy setup presented in equation 3.23 and use a recurrent neural network to incorporate previous information.

3.3.6 Bounded continuous actions

We want to use PPO with bounded continuous actions [25, Appendix C]. In our soccer environment, agents output a 2-dimensional action vector, with both elements bounded between -1 and 1. The first element controls the forward or backward speed of the agent

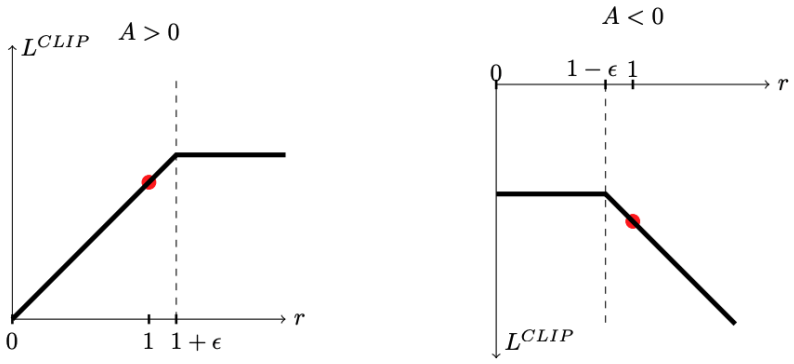


Figure 3.9: Graphical depiction of the policy objective function used in PPO. ***Left****: The case where the advantage estimate is positive. Here the optimiser is incentivised to increase* $\hat{r}$*, represented as* r *in the image, but receives no additional incentives after* $\hat{r}$ *reaches* $1+\epsilon$*.* ***Right****: The case where the advantage estimate is negative. Here the optimiser is incentivised to decrease* $\hat{r}$*, but receives no additional incentives after* $\hat{r}$ *reaches* $1-\epsilon$*. Source: Schulman et al. [57].*

and the second the rotation of the agent. A standard multivariate Gaussian distribution would not be applicable, since it does not have a natural sample bound. Instead, we squash the samples of the Gaussian distribution using a tanh function element-wise such that $a = \tanh(u)$, where u is a sample from the Gaussian distribution. Next we determine both the probability $\pi(A_t = a_t|S_t = s_t)$ of selecting a given action and the entropy $S[\pi(A_t|S_t = s_t)]$ of the policy at that state.

From probability theory [75], we know that if we have a collection of continuous variables $(U_1, \ldots, U_D)$ with probability density function f_U and a transformed set of variables $(A_1, \ldots, A_D)$, we can construct a new valid probability density function f_A. Here U and A represent the sets $U_1, \ldots, U_D$ and $A_1, \ldots, A_D$. In our case, we apply the same transformation function (tanh) element-wise to $U_1, \ldots, U_D$ to produce $A_1, \ldots, A_D$. For monotonically increasing transformations of probability distributions it is known that

$$f_A(A) = f_U(U) \left| \det \left(\frac{\partial A}{\partial U} \right) \right|^{-1}, \tag{3.68}$$

where $\frac{\partial A}{\partial U}$ represents the Jacobian matrix defined as

$$\frac{\partial A}{\partial U} = \begin{bmatrix} \frac{\partial a_1}{\partial u_1} & \cdots & \frac{\partial a_1}{\partial u_D} \\ \vdots & \vdots & \vdots \\ \frac{\partial a_D}{\partial u_1} & \cdots & \frac{\partial a_D}{\partial u_D} \end{bmatrix}. \tag{3.69}$$

In our case, the non-diagonal entries of the Jacobian are zero, because we apply the trans-

formation element-wise. The Jacobian thus becomes

$$\frac{\partial A}{\partial U} = \begin{bmatrix} \frac{\partial a_1}{\partial u_1} & \dots & 0 \\ \vdots & \vdots & \vdots \\ 0 & \dots & \frac{\partial a_D}{\partial u_D} \end{bmatrix}. \tag{3.70}$$

With $a_i = \tanh(u_i)$ we have

$$\frac{\partial a_i}{\partial u_i} = \frac{\partial \tanh(u_i)}{\partial u_i} = \operatorname{sech}^2(u_i) = 1 - \tanh^2(u_i). \tag{3.71}$$

Therefore, the required density function becomes

$$f_A(A) = f_U(U) \left| \det \left(\begin{bmatrix} 1 - \tanh^2(u_1) & \dots & 0 \\ \vdots & \vdots & \vdots \\ 0 & \dots & 1 - \tanh^2(u_D) \end{bmatrix} \right) \right|^{-1} \tag{3.72}$$

$$= f_U(U) \left| \prod_{i=1}^{D} (1 - \tanh^2(u_i)) \right|^{-1}, \tag{3.73}$$

where u_i is the i^{th} element in the sampled vector from U. In our case, $f_A(A)$ represents the policy probability density function $\pi(A_t = a_t | S_t = s_t)$. We can use equation 3.73 to get the density function output for an action by providing the Gaussian density function $f_U(U)$ and the sample from the Gaussian distribution related to that action.

For the entropy, we need to turn to the definition of the entropy of a continuous distribution:

$$S\left[\pi(A_t | S_t = s_t)\right] = -\int_{-1}^{1} f_A(A_t = a) \log(f_A(A_t = a))\, da. \tag{3.74}$$

Even with the aid of integral solving software[2], we could not arrive at a closed form solution to this integral. Therefore, we approximate the entropy over our bounded distribution to be the same as the entropy of the underlying Gaussian distribution which has a closed form solution [24]. Fortunately, PPO does not actually need the exact entropy of the distribution to work, and we found that approximating it with a loosely correlated entropy from another distribution also works. Thus we set

$$S\left[\pi(A_t | S_t = s_t)\right] \approx S\left[\pi(U_t | S_t = s_t)\right] = \frac{1}{2} \ln |\Sigma| + \frac{D}{2}(1 + \ln(2\pi)), \tag{3.75}$$

where Σ is the covariance matrix of the Gaussian and D the dimension of the action vector. We now have all the functions necessary to allow our PPO algorithm to use a bounded action space.

[2]Link: https://www.wolframalpha.com/

3.4 Multi-agent reinforcement learning

In Sections 3.2.1 and 3.2.2, we introduced the Markov decision process and partially observable Markov decision process for a single agent in an environment. However, in our soccer environment we have 22 agents all performing actions simultaneously. In this setting, we use the decentralised partially observable Markov decision process (Dec-POMDP) formulation with a set of K agents. We still have an underlying state vector $s_t \in S$ of fixed size. Each agent receives an observation, $o_{k,t} \in O_k(S_t)$, from the environment, where k is the agent's index and t the time step. Each agent takes actions $a_{k,t} \in A_k$. Different agents can therefore have different action spaces. We assume the action spaces remain constant throughout an episode. The state gets updated using a transition probability distribution over the next state s_{t+1} given the current state s_t and the actions $a_{1,t}, \ldots, a_{K,t}$ of all the agents. In a Dec-POMDP, all agents perform actions simultaneously per environment step.

We can still use single-agent reinforcement learning algorithms as derived in the previous sections, but we need to pay special attention to extra complexities in the multi-agent setting. As each agent tries to maximise the rewards it receives, the environment it experiences changes because of the presence of other agents that are also learning. Furthermore, in competitive environments such as soccer, the rewards a team receives are inversely proportional to the rewards the other team receives. The teams are competing for the same rewards, and a strategy that works well in one environment against one team might not work well against other teams.

3.4.1 Game theory

Game theory deals with decision making in environments where other actors are competing with or against each other to achieve some goal. We again define the reinforcement learning objective, adjusted from equation 3.18, to be the optimal policy for agent k:

$$\pi_k^* = B_k(\boldsymbol{\pi}) = B(\pi_1, \ldots, \pi_{k-1}, \pi_{k+1}, \ldots, \pi_K) = \arg\max_{\pi_k} \mathbb{E}_{\tau \sim \pi_k} \left[\sum_{t=0}^{T} r_k(s_t, a_{k,t}) \right]. \tag{3.76}$$

All the policies combined are represented by $\boldsymbol{\pi}$. Here we assume all agents except k are part of the environment and have fixed policies. Therefore this objective is valid for only one set of policies for the other agents. In game theory this is referred to as a best response strategy for agent k. It is represented by B, and is the best strategy or policy of one agent given that all the other agents' strategies are fixed. It is important to note that there might be many policies that represent the best response to a given setup, and a best response strategy for one set of opponents might not be optimal for all others. We therefore require a better framework to derive a policy that works against many variations of the policies for other agents.

Game theory does provide us with an objective in this multi-agent setting, called a

Nash equilibrium, which is an optimal solution where no agent has any incentive to change its strategy given the other strategies. In game theory there can be multiple strategies (policies) that are optimal in terms of Nash equilibrium. It is important to note that a Nash equilibrium policy will not necessarily have the highest expected reward in any setting of the opponent's policies. It only means that if the other agents have optimal policies, there is no other policy an agent can select to achieve a higher expected cumulative reward.

A simple example to explain this is with the game rock-paper-scissors[3]. It is well known that the optimal strategy for this game (without trying to read the opponent) is to select each of the three options with probability $\frac{1}{3}$, because any other strategy can be exploited by the opponent. The strategy $(R = \frac{1}{3}, P = \frac{1}{3}, S = \frac{1}{3})$, where R, P and S represent the probability of selecting rock, paper and scissors, is therefore a Nash equilibrium strategy as the opponent cannot generate a better strategy that wins more often. However, this is not to say that the strategy is the best against any opponent strategy. If the opponent always plays $(R = 1, P = 0, S = 0)$ the best response would be $(R = 0, P = 1, S = 0)$. However, if this strategy is played again, the opponent can counter with $(R = 0, P = 0, S = 1)$. If both sides keep on iterating on their strategy they can only hope to reach an equilibrium at $(R = \frac{1}{3}, P = \frac{1}{3}, S = \frac{1}{3})$, at which point no further improvement is possible.

Some environments do not have Nash equilibrium solutions. However, one popular setup in game theory is the zero-sum game, which always has a Nash equilibrium given that mixed strategies are allowed[4]. A mixed strategy is a stochastic strategy that consists of sampling from a set of deterministic strategies. In zero-sum games one agent's (or team's) reward gains is equivalent to another's loss. Therefore both sides are competing for the same resources and the sum of all rewards handed out to the two teams in one episode is 0. This applies well to a soccer game where one side scoring a goal receives a reward of 1 and the other side a reward of -1. We also assume a fair zero-sum game where, given optimal play, both sides have an expected reward of 0. The expected cumulative reward is always 0 for Nash equilibrium strategies in these environments, and if the players are allowed to play a mixed strategy, the game always has a Nash equilibrium.

Next we discuss a number of reinforcement learning based solutions that might help our teams to converge to Nash equilibrium strategies.

3.4.2 Deterministic vs stochastic policies

In the reinforcement learning framework, agents take actions based on a policy probability distribution,

$$\pi_\theta(a_t|s_t) = p(a_t|s_t, \theta). \tag{3.77}$$

[3]The official rules of rock-paper-scissors: https://wrpsa.com/the-official-rules-of-rock-paper-scissors/.

[4]For these environments to have a guaranteed Nash equilibrium, they should be finite in nature. However, because computers use bits, our soccer environment with continuous state and action spaces technically is finite.

For the rest of this section omit the θ variable for notational simplicity. Policies are usually (partially) random during training and all the actions usually have a non-zero probability of getting selected. This encourages exploration while the agent is learning. However, after training the agent can select the most probable action for the highest expected cumulative reward, given a particular state. A policy that always outputs the same action given the same state is said to be deterministic. For any single-agent environment that does not change dynamics between episodes, there exists at least one deterministic policy that is optimal:

$$a_t = \arg\max_a \pi^*(A_t = a|S_t = s_t). \tag{3.78}$$

In the multi-agent setting, however, we cannot assume that the dynamics of the environment (which includes other agents) does not change between episode generations. In this case a deterministic policy is often not optimal. In Section 3.4.1, we mentioned the game of rock-paper-scissors, with optimal strategy $(R = \frac{1}{3}, P = \frac{1}{3}, S = \frac{1}{3})$. This is a stochastic strategy that cannot be exploited by an opponent. An example of when a stochastic policy is optimal in soccer is with a penalty goal kick, where both the striker and goalkeeper decide in which direction they are going to kick and dive. If one of them follows a deterministic strategy it becomes easy for the other to counter and always win. The optimal strategy for both players would therefore be stochastic.

We take this into account when deriving policies for our agents. We allow them to have a stochastic strategy both at training time and when they are evaluated. In game theory a deterministic policy is often call a strategy. If we use a stochastic policy we are essentially sampling strategies over some distribution. As described in the previous section, this is called a mixed strategy as we now have a probability distribution over different deterministic strategies.

3.4.3 Self-play

One idea to train in a multi-agent setting would be to convert the multi-agent problem into a single-agent problem. We give all the agents a fixed handcrafted strategy except one agent. The environment would be static for this one agent, and we can use standard single-agent reinforcement learning algorithms to maximise the expected reward. This could lead to a policy that represents the best response to the handcrafted strategy of the other agents. However it may not be easy to design an optimal Nash equilibrium policy by hand, and so those other agents may act far from optimal. Furthermore, the trained agent will probably not do well when the strategy of the other agents change, as it was trained against only that one strategy. We therefore need some way of updating the other agents as well, to allow the stochastic policy to converge to at least one Nash equilibrium optimal policy.

Self-play is where reinforcement learning agents compete against copies of themselves

over many episodes. The agents might start out with weak strategies and gradually learn better strategies over time. In a competitive setting this self-play training has a number of favourable properties. Firstly, opponents are always just strong enough to produce both wins and losses with approximately 50% probability, leading to balance in the rewards received. Secondly, as the training agents become stronger, so do their opponents. This forms a natural curriculum learning setup where agents can first learn basic skills before more complex team strategies. Curriculum learning [71] is a method of training agents by first providing them with an easy environment and then as they become more capable gradually increasing the difficulty.

A classical approach to self-play in reinforcement learning is to train K independent agents. Each agent has its own policy and tries to maximise its own expected reward, thereby iteratively improving its policy to produce a best response to the other agents' strategies. The idea is that if each agent's updates are small enough, they might eventually converge on a Nash equilibrium policy. This has been shown to work in small environments, but seems to break down in larger environments. If one agent updates its policy too quickly, the dynamics of the environment might change too much for the other agents, and cause policy collapse. If all the policies update at the same rate they might also enter a loop and never converge to a Nash equilibrium. We encountered such a loop in the rock-paper-scissors example, when each player tries to counter the other with a deterministic strategy. We therefore need a more sophisticated approach from which to derive our final solution.

3.4.4 Fictitious self-play

Fictitious play [12] is a game theoretic concept where agents are guaranteed to converge to a Nash equilibrium strategy in 2 player zero-sum games. If we treat each of our soccer teams as an agent we can use fictitious play to try and converge towards a Nash equilibrium strategy. As we saw in rock-paper-scissors, if an agent finds only the best response to its opponent's current strategy, a cycle might emerge where none of them learn optimal strategies. To combat this, fictitious play requires that each agent should find the best response strategy to the average over the opponent's history of strategies. We can define the average strategy to be a mix over the players previous actions. In rock-paper-scissors, for example, we can calculate the average strategy of the opponent by taking the probability for each action to be the number of times that action was executed in the past divided by the total number of actions. As the iterations progress each agent is developing a more general strategy that converges to a Nash equilibrium. This algorithm is often referred to as the classic fictitious play algorithm. The update rule can be expressed as

$$\pi_{k,n+1} = \left(1 - \frac{1}{n+1}\right)\pi_{k,n} + \frac{1}{n+1}B_k(\boldsymbol{\pi}), \tag{3.79}$$

where $\pi_{k,n}$ represents the policy of agent k at iteration step n. At each iteration this update is applied to each agent's policy to derive a new strategy. $B_k(\boldsymbol{\pi})$ again represents the best response for agent k to the current setup containing all the other agent policies.

Simply taking the average over all the opponent's previous strategies might not converge that fast when the opponent is also learning. This is because the opponent's initial strategies might be quite weak and it might be better to place more emphasis on strategies encountered later in training. We however would still like to be guaranteed of converging to a Nash equilibrium.

Leslie and Collins [33] proposed an update to fictitious play, called generalised weakened fictitious play, which retains the Nash equilibrium convergence guarantee of fictitious play but allows for more flexibility in update rules. They define the update rule as

$$\pi_{k,n+1} = (1 - \alpha_{n+1})\,\pi_{k,n} + \alpha_{n+1}(B_{k,\epsilon_n}(\boldsymbol{\pi}) - M_{n+1}), \tag{3.80}$$

where $\alpha_n \to 0$ and $\epsilon_n \to 0$ as $n \to \infty$. Here we have

$$B_{k,\epsilon_n}(\boldsymbol{\pi}) = \left\{ \pi_k : \mathbb{E}_{\tau \sim \pi_1 \ldots, \pi_k, \ldots, \pi_n} \left[\sum_{t=0}^{T} r_k(s_t, a_{k,t}) \right] \geq \mathbb{E}_{\tau \sim \pi_1 \ldots, B_k(\boldsymbol{\pi}), \ldots, \pi_n} \left[\sum_{t=0}^{T} r_k(s_t, a_{k,t}) \right] - \epsilon \right\}. \tag{3.81}$$

$B_{k,\epsilon_n}(\boldsymbol{\pi})$ therefore represents the set of policies for agent k which are within ϵ reward from the optimal best response policies. In equation 3.80, M_{n+1} represents a set of perturbations such that for any constants $C > 0$ and $m > n$,

$$\lim_{n \to \infty} \sup_{m} \left\{ \left|\left| \sum_{i=n}^{m-1} \alpha_{i+1} M_{i+1} \right|\right| : \sum_{i=n}^{m-1} \alpha_i \leq C \right\} = 0. \tag{3.82}$$

The proof that equation 3.80 converges to a Nash equilibrium given these constraints is presented in Leslie and Collins [33].

With equation 3.80 we can now adapt our strategies towards a possible perturbed ϵ-best response, instead of the exact best response specified in the original fictitious play algorithm (which can be derived by setting $\epsilon_n = M_n = 0$ and $\alpha_n = \frac{1}{n}$).

This update rule gives more flexibility, and allows us to work with approximate best responses that should become more accurate over time. We can also use a variety of weightings (α_n) over historical strategies instead of being limited to a simple average. We can for example use values for α_n that prioritise actions at later iterations over those at the start of training. One way of accomplishing this is $\alpha_n = (C + n)^{-p}$, where C and p are user defined constants. This generally leads to faster convergence while still ensuring that $\alpha_n \to 0$ as $n \to \infty$.

Heinrich et al. [27] extended the work of Leslie and Collins [33] by proposing the first self-play hybrid algorithm, that combines supervised learning and reinforcement learning,

for approximating the update rule in equation 3.80. It is an open question whether this algorithm inherits the guarantee of converging to a Nash equilibrium. A simplified version of their general fictitious self-play algorithm is presented in Algorithm 2.

Algorithm 2 General fictitious self-play.

Require: $\boldsymbol{\pi}_0$, $\boldsymbol{\alpha}$, s and f ▷ Here $\boldsymbol{\pi}_0$ is the initial setup for the average policies of each agent, and s (slow) and f (fast) are user defined constant integer values.

$\boldsymbol{\beta}_0 \leftarrow \boldsymbol{\pi}_0$
for j incrementing from 0 **do**
  $\mathcal{D} = \text{GenerateData}(\boldsymbol{\pi}_j, \boldsymbol{\beta}_j, s, f, \alpha_j)$
  for each player k **do**
    $\mathcal{M}^k \leftarrow \mathcal{M}^k \cup \mathcal{D}^k$
    $\beta^k_{j+1} = \text{ReinforcementLearning}(\mathcal{M}^k)$
    $\pi^k_{j+1} = \text{SupervisedLearning}(\mathcal{M}^k)$
  end for
end for
return $\boldsymbol{\pi}_j$

In Algorithm 2, the ReinforcementLearning function takes agent k's memory $\mathcal{M}^k$ and performs offline reinforcement learning on that data. In this offline setting we essentially calculate the best response β^k_{j+1} to the opponent's actions presented in the data for iteration j. The SupervisedLearning function works on the same memory but in turn learns an average policy π^k_{j+1} for the current agent. The GenerateData function is given in Algorithm 3.

Algorithm 3 Data generation for the general fictitious self-play algorithm.

Require: $\boldsymbol{\pi}$, $\boldsymbol{\beta}$, α, s and f

$\boldsymbol{\sigma} \leftarrow (1-\alpha)\boldsymbol{\pi} + \alpha\boldsymbol{\beta}$
$\mathcal{D} \leftarrow$ sample s episodes from strategy profile $\boldsymbol{\sigma}$
for each player k **do**
  $\mathcal{D}^k \leftarrow$ sample f episodes from strategy profile $(\boldsymbol{\beta}^k, \boldsymbol{\sigma}^{-k})$
  $\mathcal{D}^k \leftarrow \mathcal{D}^k \cup \mathcal{D}$
end for
return $\{\mathcal{D}^k\}_{1 \le k \le N}$

As can be seen in Algorithm 3, we use a sampling strategy profile $\boldsymbol{\sigma}$ that uses both $\boldsymbol{\pi}$, the weighted average strategies of each agent, and $\boldsymbol{\beta}$, the best responses of each agent with respect to the other agents' weighted average strategy. In 2 player games $\boldsymbol{\sigma}$, $\boldsymbol{\pi}$ and $\boldsymbol{\beta}$ include two strategies each. For every iteration j, each agent k adds s episodes from $(\boldsymbol{\pi}^k_j, \boldsymbol{\pi}^{-k}_j)$ and f episodes from $(\boldsymbol{\beta}^k_j, \boldsymbol{\pi}^{-k}_j)$. Here $\boldsymbol{\pi}^{-k}_j$ indicates that every agent except agent k is using their average strategy policy from $\boldsymbol{\pi}_j$.

Heinrich et al. [27] use an off-policy reinforcement learning algorithm which can use all the generated episodes, because the opponents always use their average strategies $\boldsymbol{\pi}^{-k}_j$. Off-policy algorithms do not require that the data that is being trained on only come from the current policy, and therefore they can train on all the data as long as the opponents use a

consistent policy. In the supervised learning algorithm we need to set appropriate values for s and f to achieve optimal performance. If we want to train on the true average target distribution we would only train on $(\boldsymbol{\pi}_j^k, \boldsymbol{\pi}_j^{-k})$, i.e. $f = 0$, but this requires us to have a very large memory pool containing all the data the agent generated so far. If we assume that our updating procedure is incremental we could alternatively only use episodes from $(\boldsymbol{\beta}_j^k, \boldsymbol{\pi}_j^{-k})$, i.e. $s = 0$, as we are slowly incorporating more of the latest policy into the average strategy. Depending on the problem, a balance between these extremes is usually selected.

3.4.5 League training

The general fictitious self-play algorithm (Algorithm 2) suffers from a few practical shortcomings when using neural networks. Heinrich et al. [27] assume that one policy can easily represent a range of mixed strategies. While this is true for small games like tic-tac-toe, it generally does not work well for large environments where each policy might focus on a specific way of playing a game. To address this problem, algorithms like AlphaStar [42] and OpenAI Five [11] rely heavily on the idea of league training, where a team that is training plays episodes not only against itself, but also against various other policies. By playing against different policies, instead of using one network to approximate or average over different policies, agents can learn general strategies in a much more stable manner. In the case of AlphaStar these other policies are also learning, but our limited computational resources prevent us from training more than one team at a time. We therefore focus on OpenAI Five which uses a static league of opponents that do not update. OpenAI Five operates by training one PPO agent, allowing it to play against itself for 80% of games and, to prevent strategy collapse, allowing the learning policy to play against past versions of itself for the remaining 20% of the time. The policy is saved every 10 iteration steps, and forms a new opponent. By now sampling from these policies the current agent can generate a more robust strategy. Instead of sampling uniformly from past strategies, OpenAI Five assigns a quality score q_i to each opponent i. Opponents are sampled with probability p_i which is proportional to e^{q_i}. The quality score for a given opponent is updated only if they are defeated in an episode, according to

$$q_i \leftarrow q_i - \frac{\omega}{N p_i}, \tag{3.83}$$

where ω is a constant learning rate (set to 0.01 by default) and N is the number of opponents. When a new opponent is added its quality score is set to the highest quality score of all other opponents. The update rule in equation 3.83 lowers the quality score of weaker opponents faster than stronger opponents, and results in weaker opponents being sampled less often than stronger ones. This speeds up learning while still allowing a wide variety of strong opponents to be sampled from. Furthermore, in the update rule there is a division by the probability of being sampled, which prevents stronger opponents from being punished too

harshly as they are being sampled more often.

In our work we also experiment with league training, as further discussed in Chapter 5, and our league training algorithm is closely related to OpenAI Five.

3.5 Summary

In this chapter we provided theoretical background for the various algorithms used in our work. We started with an overview of general neural network architectures. We then moved on to reinforcement learning, Markov decision processes and deep reinforcement learning. We believe reinforcement learning is a good fit for our problem, because of the clear reward signal in our environment, namely scoring a goal. Soccer also naturally fits into the Markov decision process framework.

We derived a number of single-agent learning algorithms relevant to this work. We discussed policy gradient methods and introduced various enhancements to arrive at the actor-critic setup. We then described the proximal policy optimisation (PPO) algorithm [57], which is the main algorithm for our work. We also discussed a method to allow PPO to work with continuous bounded actions as required in our environment.

Lastly, we investigated properties and complexities of multi-agent reinforcement learning that are typically not present in the single-agent setup. We started by briefly introducing the field of game theory and why we might want to learn stochastic policies over deterministic ones. We then considered fictitious self-play which seems a promising direction to learn ever improving strategies from scratch. We also considered an opponent league based learning strategy that is commonly used for practical multi-agent reinforcement learning algorithms.

In the next chapter we introduce our custom soccer environment.

Chapter 4

Simulation environments

In this chapter we introduce the two environments used in our experiments. We start in Section 4.1 by introducing our custom soccer environment, which is used in most of our experiments. In Section 4.2, we discuss the initial shaped reward that is required for agents to learn in this environment. After the agents learn to score goals, the shaped reward is removed. In Section 4.3, we discuss the code acceleration library called JAX, which we use in our custom environment. This allows us to prototype much faster as training can be performed faster. In Section 4.4, we present code for our updated training setup. We then introduce our wrapped 2D RoboCup environment in Section 4.5. We end the chapter with a summary in Section 4.6.

4.1 Custom soccer environment

A future aim of this work is to create a team of 11 players that can compete in the 2D RoboCup simulation league. To this end, we adapted the 2D RoboCup simulator to work with a Python interface and trained simple players using reinforcement learning in this adapted simulator. The complete Python wrapper[1] is available inside the multi-player reinforcement learning framework called Mava, further described in Section 5.3. This wrapper can process around 61 environmental steps per second for 22 players, by taking away all time pauses in the RoboCup simulator. This is relatively slow when compared to our custom environment, which outputs around 1533 steps per second with 22 players. Firstly, in RoboCup competitions the players are allowed to communicate with one another only through the RoboCup environment interface. This interface allows for a few bits of information to be broadcast between players on the field at each step. Each player is also limited to the same computational constraints, to make the competitions more fair. To enforce these constraints each player runs in a separate process and communicates exclusively with the environment using its own network socket. In our 2D RoboCup wrapper we can have

[1] Our Python wrapper for the 2D RoboCup environment is available inside Mava [47], which can be found here: https://github.com/instadeepai/Mava.

all the players on the same process, but communication with the server is still expensive as we need to communicate each player's action with the environment through a separate connection, and this additional overhead slows down each environment step. Secondly, the RoboCup environment has a number of advanced rules such as throw-ins, penalties and the modelling of player stamina, which take up simulation time. To focus on training a full team using reinforcement learning, we require a simpler soccer environment with fewer of these complexities, without losing the core aspects which make the 2D RoboCup simulation league difficult to master, such as continuous actions, partial observations and 22 players acting simultaneously in the same environment. Our custom environment allows us to iterate different ideas and algorithms much faster, and then later apply our best algorithm to the 2D RoboCup environment.

To this end, we constructed a custom 2D soccer environment[2], running at over 1533 steps per second[3], which is more than 25× faster than the fastest we could get from the 2D RoboCup simulator (more on this in Section 6.2.1). We achieved this speed by combining parallel matrix operations on limited game physics using the NumPy[4] library with an XLA accelerator provided by the JAX[5] library. We discuss these speedups in more detail in Section 4.3. Our environment is also significantly lighter than the 2D RoboCup environment in terms of CPU computation requirements.

Our environment is similar to the one created by Liu et al. [35]. However theirs is more than 100× slower than ours when running 22 players, and provides full state information to the players which would require additional processing to convert to partial observations. Our environment provides partial observations to players from the outset, similar to the RoboCup simulator. Our environment is able to run anything from 1 vs 1 player games to the full 22 player (11 vs 11) setting. Players can hit only the ball, and not each other, which reduces the number of collision calculations necessary per step. The ball bounces from the sidelines, which eliminates the need for throw-ins. Generated views of our 2D environment are provided in Figure 4.1.

In this 2D simulator, a player can kick the ball using its side legs. It can also dribble the ball by repeatedly hitting it with the central circle part of its body. If the ball has a non-zero velocity for a given time step, its position is updated and its speed decreased, if no player is touching it, thus simulating friction. The short yellow line on each player indicates the direction it is facing. When the ball crosses a goal line the score of the game (displayed at the top centre of the screen) is updated. A game consists of a predefined number of steps in which players can execute actions, and the current environment step is displayed at the top

[2]A pure NumPy implementation of our custom 2D soccer simulator is available on GitHub and can be found here: https://github.com/DriesSmit/MARL2DSoccer.

[3]These measured step speeds are for taking random actions in the environment with only one process running. When parallel workers are used with neural network policies, the average environment step speed is slower. See Section 6.2.2.

[4]NumPy: https://NumPy.org/

[5]JAX: https://jax.readthedocs.io/en/latest/notebooks/quickstart.html

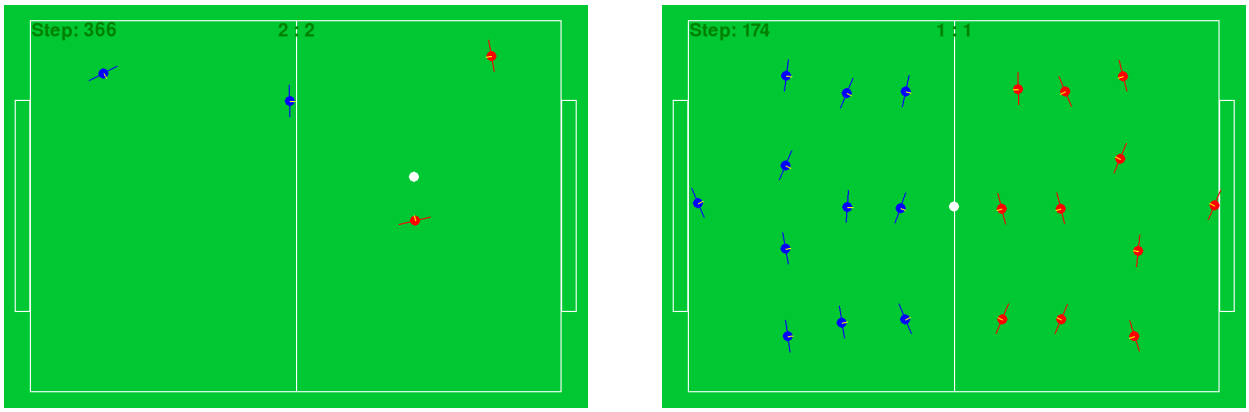


Figure 4.1: Top-down graphical views of our 2D soccer environment. ***Left****: 4 player game.* ***Right****: full 22 player game. The blue players' goalpost is on the left side of the field, and the red players' on the right. The goalposts are made larger to prevent players from completely blocking off their own goals.*

left of the screen. After a goal is scored the players respawn to their starting positions and orientations, with some noise added. The ball can also be spawned at a random location. After a game is completed, the score determines the winner or whether it is a draw.

The players act in the environment by providing a continuous valued action vector of size two. The first entry signifies movement along the axis a player is facing, with forwards being positive and backwards negative. The second entry allows the agent to rotate, where a negative number represents anti-clockwise rotation, 0 no rotation and positive clockwise rotation. These actions update the position and rotation of each player over short distances. Stamina and fatigue, which are included in the RoboCup simulator, are omitted in this simplified environment. Direct player-to-player communication is also omitted. The players receive a visual input and an indication of how many steps remain before the game is completed. To keep the setup close to RoboCup's, each player is limited to a partial observation through a 180° field of view, as shown in Figure 4.2. This means that a player can see only what is in front of it. We decided that the vision system should be egocentric (from the player's perspective), even though this generally leads to a more challenging problem than a fixed top-down vision system. The egocentric view is closely related to how the 2D RoboCup simulator is set up and also seems more realistic. The players are provided with the egocentric coordinates (distance and angle) and velocities of all objects (ball and other players) in their 180° field of view. This is similar to the environment of Liu et al. [35], although in their case complete state information is provided to the players. In our environment each player also receives the absolute coordinates of its location on the field, a normalised measure of the time remaining in the game, and the last actions of all the players it can see. Lastly, each player receives its starting position in coordinate form. This may encourage agents to learn differentiated behaviours depending on what position they

are playing.

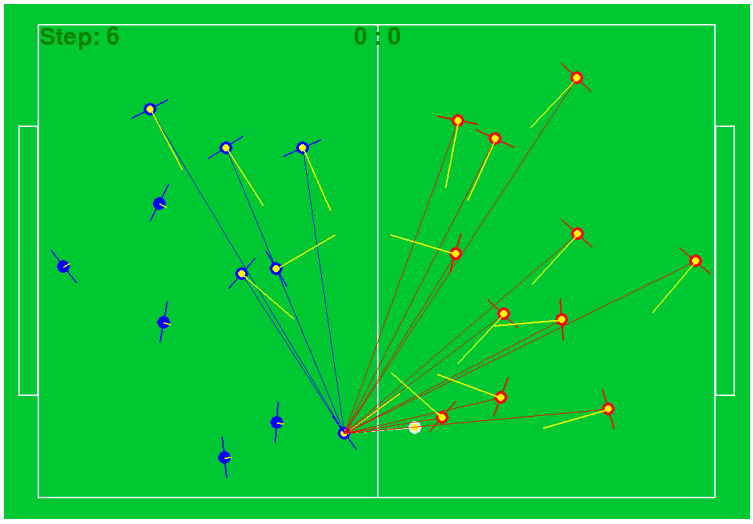


Figure 4.2: Top-down graphical view of an agent's partial observation in our 2D soccer environment. The lines extending from the one agent represent the teammates, opponents and ball that the agent can see in its 180° field of view.

4.2 Reward shaping

To facilitate initial learning, a temporary shaped reward signal is provided to the players. We found this to be necessary because initially, with actions being mostly random, the likelihood of scoring a goal is quite low. The formula for this reward is

$$r_{i,t} = g_{i,t} + 0.1b_{i,t} + 0.05a_{i,t}, \tag{4.1}$$

where $r_{i,t}$ represents the reward received by player i at a given time step t. Here $g_{i,t}$ represents 0 if no goal has been scored in step $t-1$, $+1$ if the player's team scored and -1 if they were scored against. The variable $b_{i,t}$ represents the magnitude of the ball's velocity towards the opponents' goal and $a_{i,t}$ the magnitude of the player's velocity towards the ball's coordinates. This incentivises the players initially to move towards the ball and kick it towards the opponents' goal. Once the players score at least one goal in more than 75% of the games, the reward shaping is removed. From that point on the players rely only on the $g_{i,t}$ term for training. We chose 0.1 and 0.05 as we would like the ball moving towards the goal to provide more reward than the ball moving away from the agent. Other than this constraint, the values are quite arbitrary and we found that the agents learn with other weightings too.

This initial reward shaping signal is considered the main handcrafted part of our algorithm, necessary for the agents to learn to score goals before they can rely fully on the sparse

rewards received by the environment. However, we found that after less than 2% of the total training time the agents can already score goals and the reward shaping is removed.

4.3 JAX acceleration

To achieve a fast environment throughput we need to perform some optimisations on the players-environment interaction code. A simplification already present in our environment, as well as the 2D RoboCup environment, is that players cannot collide with each other[6]. We still need to compare player positions with one another when calculating the observations, since each player receives egocentric information of each other player it can see. We therefore have $22 \times 21 = 462$ comparisons to compute, which is quite a lot if executed sequentially. We parallelise this computation by using the NumPy library, but even this can be quite slow as we are still executing Python instructions. However hotly debated, Python is often regarded as being slower than lower-level languages such as C++ due to its use of bytecodes instead of machine code. Bytecodes provide the advantage of making Python platform-independent, but requires extra computation time to convert instructions to a form that can be executed on a machine. This is in contrast to C++ which compiles code directly to low-level machine code. For additional speedup in our simulation environment it would be advantageous to somehow convert Python code directly to machine code without needing to do it at every step of the execution process.

A Python library call JAX can do exactly this, and uses almost exactly the same interface as NumPy. JAX has just-in-time (JIT) compilation capabilities, allowing for a program to be compiled as it is being run. Once the program is compiled it can be executed without interfacing with Python instructions again. This provides a good trade-off between remaining a higher-level language like Python and being capable of compiling functions such as the environment step function to machine code. Under the hood JAX uses accelerated linear algebra (XLA) which is a domain-specific compiler for linear algebra, and is specifically designed to accelerate parallel matrix computations. By using JIT compilation on our environment step function we achieve a 7× speedup over using pure Python operations alone, and more than a 3× speedup over NumPy.

4.4 Python code overview

We now present an overview of the training code that can be used with the custom environment, as well as the wrapped RoboCup environment. This setup differs from usual reinforcement learning where each agent executes its actions sequentially. It does allow for sequentially action selection, but also for batched execution of team policies. We found

[6]There are some other player to player calculations performed in the 2D RoboCup environment, such as whether a foul has been committed.

this to drastically improve performance. Furthermore, experience is stored and processed in batches which reduces data transmission overhead. A Python code snippet for a simple reinforcement learning training routine in our simulation environment is presented below.

```
# Initialise the soccer environment.
soccer_env = SoccerEnvWrapper(num_per_team=11)

# Initialise the teams.
rl_team = RLTeam()
random_team = RandomTeam()
teams = [rl_team, random_team]
while True:
    # Reset the agents' internal states.
    for t_i in range(len(teams)):
        teams[t_i].reset_brain()

    # Start a new game.
    observations, states, rewards = soccer_env.reset_game()

    # Store the initial observations, states and rewards received
    # from the environment.
    rl_team.observe_first(observations[0], states[0], rewards[0])

    done = False
    while not done:
        # Get the agents' actions.
        actions = []
        for t_i in range(len(teams)):
            actions.append(teams[t_i].get_team_actions(observations[t_i]))

        # Step the environment.
        observations, states, rewards, done = soccer_env.step(actions)

        # Display the soccer screen.
        soccer_env.display_screen()

        # Store the observations, states and rewards for this
        # environment step.
        rl_team.observe(observations[0], states[0], actions[0],
                        rewards[0], done)

    # Perform an agent update step.
    rl_team.learner_step()
```

The above routine shows how the environment can be used to train a team of agents using reinforcement learning. In this environment the players receive zero reward until a goal is scored. If the team that is training scored the goal, they receive a reward of $+1$. If they were scored against, they receive a reward of -1. The code for this reinforcement

learning team can be placed inside the `RLTeam` class. The team collects experience through the `observe_first()` and `observe()` functions and updates its internal networks when `learner_step()` is called. One can experiment with different algorithms for the `RLTeam` class, from independent policies per agent (as is done in this work) to one policy controlling the entire team.

4.5 2D RoboCup simulation environment

Our main focus is on training and evaluating in our custom environment with its simpler dynamics and faster runtime. However, we would also like to see whether it is possible to apply our learning algorithm to the 2D RoboCup simulation environment. To do this we constructed a Python wrapper for the RoboCup environment so that it has the same application programming interface as our custom environment. A monitor for the 2D RoboCup simulator is shown in Figure 4.3. This monitor differs from the one shown in Figure 2.4, but both can connect to the same RoboCup server and can be used to visualise the same game.

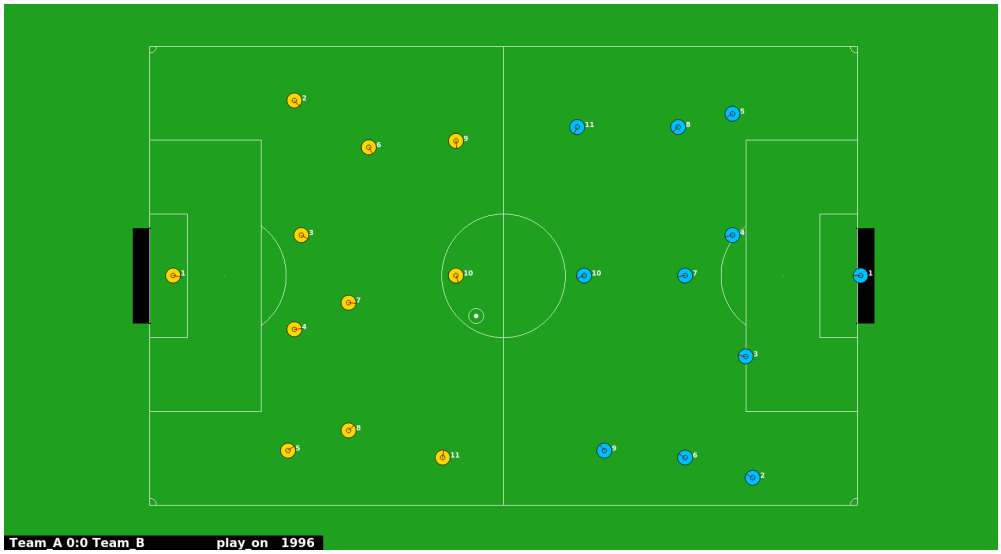


Figure 4.3: Top-down field view of the wrapped 2D RoboCup environment.

In this environment players receive partial egocentric observations of all the objects they can see in their 90° field of view (half the visibility provided in our custom environment). Furthermore, the RoboCup server does not provide players with their absolute coordinates as observations, but rather relative coordinates of the landmarks on the field visible to each player. To simplify learning we opt to not provide raw landmark coordinates to each player, and instead we calculate the translation and rotation necessary to map these relative coordinates to absolute coordinates. The calculated translation and rotation are then used

to estimate the player's absolute coordinates (with added noise provided by the simulator), which we provide to the player.

In our wrapper we scaled all the RoboCup actions to be between -1 and $+1$, as that is the range of our algorithm's output. In the RoboCup environment the action space is more complicated than in our custom environment. Players are allowed to perform one of five main actions per step, namely dash, kick, tackle, turn or not moving. Dash allows a player to move around in any direction relative to its body, where moving in the forward direction consumes only half the stamina than moving backwards. Each player has a stamina bar that increases by a set number of points each step. When a player executes any action, except the no move action, stamina gets used. When a player's stamina reaches zero, the effectiveness of actions (accuracy and power) is drastically reduced. The player can then choose to either wait for more stamina or move with this reduced capability. Furthermore, a player can move faster in the forward direction than sideways or backwards.

The kick action allows a player to kick the ball in some chosen direction relative to its body, if the ball is within the body radius. The player's kicking power reduces as the kicking angle moves from the forward direction ($0°$) to the backward direction ($\pm 180°$). The tackle action is used to kick the ball in a direction chosen by the player and can be used if the ball is just outside the player's body radius, but more noise is added to the direction the ball moves compared to normal kicking. The tackle action can also be used to foul another player that is close by, which might result in a penalty if the referee observes it (with some probability). The turn action allows the player to turn in some chosen direction relative to its body.

Only one of the above five actions can be performed at a time, which is different to our custom environment. Players can simultaneously perform additional actions such as turning their heads to look in a different direction than their body is facing, and changing their viewing quality to receive visual information more often (e.g. every step) but in a more narrow field of view, or less often (e.g. every second step) but in a wider field of view. To simplify training we fix the head to face in the direction of a player's body, and we also fix the viewing angles. We replace the goalkeeper (who can catch the ball) with another normal player to keep the action space consistent for all players.

RoboCup uses a hybrid action space where both discrete and continuous actions can be selected. Each of the main actions has continuous controls associated with it that determine in which direction and, when applicable, with what power to execute that action. However, our learning algorithm works only with continuous actions and we therefore need to convert this hybrid action space to a continuous one. To do so, we add five additional continuous actions to determine which of the main actions to take. The highest of these five values will indicate which action to take, along with its corresponding continuous control outputs. The complete action output vector is shown in Table 4.1.

Additional rules that we keep in our wrapped RoboCup environment are goal side and sideline throw-ins. If a player kicks the ball out of play the ball will respawn at either a

Table 4.1: The continuous action space used by the wrapped RoboCup environment. All actions are scaled and normalised to have continuous values between -1 *and* 1.

Action index	Action description
0	Selection score for the dash action.
1	Selection score for the kick action.
2	Selection score for the turn action.
3	Selection score for the tackle action.
4	Selection score for the no move action.
5	Dash power (-1 backwards to $+1$ forwards).
6	Dash direction ($-90°$ to $90°$).
7	Kick power (0 to 1).
8	Kick direction ($-180°$ to $180°$).
9	Turn direction ($-180°$ to $180°$).
10	Tackle foul (0 for not foul and 1 for foul).
11	Tackle direction ($-90°$ to $90°$).

sideline or in front of one of the goals. The team that last touched the ball is not allowed within a certain radius of the ball. This allows the other team to get to the ball, and as soon as they kick the ball the game continues as normal.

To enable initial learning we introduce reward shaping in exactly the same manner as used in our custom environment. The RoboCup simulator does not allow for automatic restarting of a game and therefore we reset the positions and stamina of all the agents using the RoboCup server, while resetting the environment step counter and number of goals scored inside the wrapped code.

We also accelerate the wrapped RoboCup environment by removing all time pauses. As mentioned before, this allows the environment to process around 61 steps per second in the 22 player setting. For each of the RoboCup games a server needs to be created with which each agent can communicate using a socket connection. We developed code to enable the receiving of state information that closely matches up with player observations. This presented some issues as communication over these socket connections, especially the ones used in RoboCup, is inherently unstable. This results in missing and misaligned packets that can cause problems for a reinforcement learning algorithm. To counteract this we implemented an additional flag that indicates to the agents whether a specific piece of data is missing. We also implemented additional measures to combat data misalignment (where the state and observation information differ due to new information not arriving on time) such as checking that the observations and states match up at training time.

4.6 Summary

In this chapter we explained both our custom soccer environment and wrapped RoboCup environment. We decided to design our own 2D simulation environment and use it for most of our experiments, instead of using the 2D RoboCup simulator with its higher computational overhead and extra game dynamics that might distract from the core focus of our work. We do pay special attention to retain the major challenges in 2D soccer in our environment, namely training with 22 independently executing policies in a partially observable setting. Our environment also provides an initial shaped reward signal, that we consider handcrafted information, but this is quickly removed once the agents start learning. The agent observation and action setup are also similar to RoboCup with regards to providing a variable size list of observed players and ball, and having the players output continuous actions. We also implemented a wrapped version of the 2D RoboCup environment. While our main focus is on the custom environment, we will use this wrapped environment to show that it is possible to transfer our algorithm to the RoboCup environment.

In the next chapter, we turn our attention to the design of our main reinforcement learning algorithm for agents in our custom soccer environment.

Chapter 5

System design

In this chapter we investigate what is needed to scale end-to-end reinforcement learning systems to the full 22 player setting in simulated soccer. We wish to encode as little as possible domain-specific knowledge into our learning algorithm. If we want to train a competitive team using algorithms tailored specifically to the soccer environment, we could encode individualised rewards for each agent based on their position, e.g. defenders getting penalised for being in front of strikers, or we could punish players for bunching together. We could also initialise the policy with imitation learning, where agents learn to copy some handcrafted strategy, and apply reinforcement learning after that. Techniques like these are used by participants in RoboCup competitions, but will not generalise to or be of much use in other multi-agent domains. Instead, in our work we set a minimal reward for goals scored, and provide no agent-specific reward after initial reward shaping. This brief initial reward shaping is needed so that the agents can experience non-zero rewards in order to kick-start their learning. We believe that even this initial reward shaping can potentially be removed by instilling a form of artificial curiosity in the agents [62]. We begin this chapter with some of the challenges of scaling reinforcement learning algorithms to the 22 player setting, and then provide possible solutions.

5.1 Challenges in the 22 player setting

Our aim is to get a reinforcement learning system to learn, from scratch, how to play competitive soccer in an environment with 22 players. Soccer is already a challenging problem when training only, say, 4 agents [35, 36] and it becomes significantly more difficult for standard reinforcement learning algorithms to achieve good results when training in the full 22 player setting. Reasons for this are presented next.

5.1.1 Dynamic environment

Training in environments where many agents are learning simultaneously is known to be a challenging problem in reinforcement learning. In multi-agent reinforcement learning, each

agent perceives the environment to be changing over time and this can destabilise training. Assigning value to the action an agent takes at a particular time step can also be challenging, as many other agents are also performing actions at the same time.

5.1.2 Partial observability

Agents can only see in front of them and therefore might not know where all the objects in the environment are. This makes training more difficult as agents may need to learn some form of memory in order to remember important information which might not be observable at every time step.

5.1.3 Sparse rewards and credit assignment

Environments with sparse rewards also complicate learning. In soccer, agents typically receive only a few non-zero rewards over an entire game. This makes it challenging for agents to determine which (if any) of their previous actions were responsible for these rewards.

5.1.4 Self-play

Agents playing against copies of themselves can be an effective way to introduce curriculum learning into the training process. The agents might start out against weaker opponents and as they improve, the opponents also improve, so that they gradually learn better strategies over time. Naive self-play where one network plays only against itself generally leads to strategy collapse [11], where the current network learns to beat only one strategy while forgetting all the previous strategies it learned over the course of training. Furthermore, a network update leading to a worse strategy might be challenging to detect, as both teams will play with this new strategy and the reward might not change to indicate deterioration.

5.1.5 Computational cost

When training with 22 agents it takes significantly longer to process an episode, because at each step 22 neural network forward passes need to be computed. Every agent's interaction with the environment also needs to be calculated, along with updated state and observation data. It may be argued that there are more agents generating experience per episode, which could counteract episodes taking more time. However, experiences generated in the same episode are significantly more correlated than across different episodes, which can lead to catastrophic forgetting [9] and deterioration of learning.

5.2 Proposed solutions

As discussed in Chapter 3, we focus on using multi-agent proximal policy optimisation (MA-PPO), where policy updates can be controlled more easily and the likelihood of policy collapse from the non-stationary nature of multi-agent environments is reduced. MA-PPO has been shown to work well in cooperative settings [76], and we would like to extend MA-PPO to also work in the mixed cooperative-competitive setting. To address the challenges listed in Section 5.1 we propose Algorithm 4, where each iteration adds one episode's experience to the data buffer and performs updates on the networks based on the collected experiences. An episode is played out between the learning team and either the Polyak averaged opponent [68] or a league of opponents. The experience of the learning team is sent to the learner and is used to update the learning team's networks. The learner uses the standard PPO training setup (Section 3.3.5) for each agent, with a clipped policy objective function and entropy term. The critic is used to derive an advantage function which determines how to update the policy. The opponent's networks are updated by slowly adjusting them towards the learner's networks. In Algorithm 4, discounted_return(r_1) calculates the sum of discounted future rewards at every time step, and average_score(r_1) calculates the average game outcome (win: 1.0, draw: 0.5, lose: 0.0) over all episodes in r_1. Lastly, entropy($\pi_{\theta_1}(a \mid o_1)$) calculates an entropy value for the action (a) distributions of the policy given the observations o_1. Our various improvements over the standard PPO implementation in the context of multi-agent reinforcement learning are presented next.

5.2.1 Clipping value

As recommended in previous work [76] we set a small clipping value for the policy objective function in order to avoid strategy collapse. The clipping value controls how far a policy can deviate from the original policy that was used when the experience was collected. If the policies deviate too much the experience might become unreliable, since the expected cumulative reward was estimated for the old policy. This is especially important in the multi-agent setting where 22 actions are taken per step instead of just one. Furthermore, by making only small updates to the team policies we try to keep the state distribution close to what the critic has experienced in the past. The critic therefore provides better estimates of the discounted cumulative reward, which reduces noise in the learning process. We use a clipping value of 0.1 instead of the default value of 0.2 for PPO.

5.2.2 Training data usage

A powerful trick in PPO is mini-batching, where a batch of experience is split into smaller mini-batches to train on multiple times, allowing experience to be used more than once. However, as noted by Yu et al. [76], too many updates on the same experience can lead to degraded performance in the multi-agent setting. We therefore use a reduced number

Algorithm 4 Our multi-agent PPO algorithm.

Require: θ_1, ϕ, λ_π, λ_v, $\alpha_v = 0.5$, $\alpha_p = 1.0$, $\alpha_e = 0.01$, $\alpha_s = 0.01$, $\epsilon_p = 0.1$, $\epsilon_s = 0.55$, $epochs\_per\_step = 5$ ▷ We use a smaller ϵ_p clipping value and a smaller $epochs\_per\_step$ value than in the standard PPO implementation, for increased training stability, and a small α_s value to slowly update the opponent strategy. We also use a different set of weights for the opponent agents (θ_2) than for the learning agents (θ_1).

$\theta_2 \leftarrow \theta_1$
$\mathcal{D} \leftarrow \emptyset$
for each iteration **do**
  for t **in** environment steps **do**
    $s_t \sim p(s_t \mid a_{1,t-1}, a_{2,t-1}, s_{t-1})$
    $s_{1,t}, s_{2,t} \leftarrow s_t$ ▷ Each team has its own state value.
    $o_{i,t} \sim p(o_{i,t} \mid s_t)$ **for** $i \in \{1, 2\}$
    $a_{i,t} \sim \pi_{\theta_i}(a_{i,t} \mid o_{i,t})$ **for** $i \in \{1, 2\}$ ▷ Perform two batched forward passes (one per team) for faster inference.
    $r_{1,t} \sim p(r_{1,t} \mid s_t, a_{1,t}, a_{2,t})$
    $\mathcal{D} \leftarrow \mathcal{D} \cup (o_{1,t}, s_{1,t}, a_{1,t}, r_{1,t})$
  end for
  for each learner step **do**
    $o_1, s_1, a_1, r_1 \sim \mathcal{D}$ ▷ Sample experience from the learning team.
    for epoch **in** range($epochs\_per\_step$) **do**
      $V_1 \leftarrow V_\phi(s_1)$ ▷ The value function uses full state information.
      $V_{target} \leftarrow \text{discounted\_return}(r_1)$
      $A_1 \leftarrow V_{target} - V_1$
      $V_{obj} \leftarrow -A_1^2$
      $u_1 \leftarrow \frac{\pi_{\theta_1}(a_1|o_1)}{\pi_{\theta_1 \text{old}}(a_1|o_1)}$
      $p_{obj} \leftarrow \min(u_1 A_1, \text{clip}(u_1, 1-\epsilon_p, 1+\epsilon_p) A_1)$
      $J(\theta_1, \phi) \leftarrow \mathbb{E}\left[\alpha_v V_{obj} + \alpha_p p_{obj} + \alpha_e \text{entropy}(\pi_{\theta_1}(a \mid o_1))\right]$
      $\theta_1 \leftarrow \theta_1 + \lambda_\pi \nabla_{\theta_1} J(\theta_1, \phi)$
      $\phi \leftarrow \phi + \lambda_v \nabla_\phi J(\theta_1, \phi)$
    end for
    if average_score(r_1) $> \epsilon_s$ **then** ▷ When necessary, freeze the opponents' weights to stabilise training.
      $\theta_2 \leftarrow \alpha_s \theta_1 + (1 - \alpha_s)\theta_2$ ▷ Slowly update the opponents' weights.
    end if
  end for
end for
return θ_1, ϕ

of epochs per step of 5, instead of 64 as in the original PPO implementation [57]. We found experimentally that this provides the best benefit ratio between experience reuse and training stability.

5.2.3 Environment-provided global state

Another improvement we borrow from Yu et al. [76] is to provide the global state as input for each agent's critic. We use the popular decentralised execution with centralised training

architecture [17, 43], allowing full state information to be fed into the critics and only partially observed inputs into the policies (the critics are discarded at test time). Full state information allows the critics to better model the value function because no information is hidden from them.

5.2.4 Parallel computation

Reinforcement learning is known to require a large number of episode iterations, in challenging environments, before agents achieve good results [77, 74]. This poses a problem as agents can take a long time to train. The process of testing different algorithms therefore becomes slow, and one way around this issue is by trying to parallelise computations. A typical reinforcement learning algorithm iterates between two operations, namely collecting new experience by running agents in an environment and updating the agents' network using this new experience. The agents collecting new experience are usually called workers, and the experience is sent to a learner which updates the network. Workers normally run on a computer's CPU, while the learner normally uses a GPU (or in some cases a TPU). As described in Section 3.3.4, it is possible to get the workers and learner to also run in parallel. If more than one CPU is available, different workers can simultaneously collect experience, and a speedup of the whole reinforcement learning process can be achieved using Algorithm 5.

Algorithm 5 Training reinforcement learning agents by attempting to maximise the use of a standard desktop computer's processing power, through its GPU and CPU threads.

Require: n ▷ Where n is two less that the number of CPU threads available, reserving one for the learner and one for the base operating system.

```
Create n workers that each uses their own CPU thread.
Initialise experienceBuffer.
Initialise learner with a default network.
for worker in workers do
    worker.network = learner.network
    worker.start()
end for
Wait for workers to finish.
for i in training steps do
    Reset experienceBuffer.
    for worker in workers do
        experienceBuffer += worker.experience
    end for
    for worker in workers do
        worker.network = learner.network
        worker.start()
    end for
    learner.start(experienceBuffer)
    Wait for workers and learner to finish.
end for
```

Algorithm 5 drastically decreases training times and therefore more reinforcement learning algorithms can be evaluated in a shorter time span. For our experiments we run multiple workers (CPU threads) and one learner (GPU) on the same machine. Further specifications on the computational hardware and number of workers found to be suitable are described in Sections 6.1 and 6.2. To achieve the parallelisation we make use of the Mava framework [47], discussed in Section 5.3. This framework, which is based on Acme [29], uses Launchpad [73] to asynchronously train and run separate processes for the workers and learner.

5.2.5 Code compilation acceleration

As mentioned in Section 4.3, Python code can be inherently slower than compiled languages like C++, because it is a dynamic language, and multiple computational steps that normally happen during compilation in C++ are moved to the runtime in Python. To overcome this obstacle we uses JAX's JIT functionality to compile the code when we first call the environment step function. Our policies and learner are written in TensorFlow[1] which has a conversion tool called tf.function. This tool transforms a Python function to Python-independent dataflow graphs. By applying this method we can accelerate both our policy network forward passes, used to calculate new actions, and the entire learner step function, which is used to perform updates to our networks. Therefore by using tf.function along with the JIT compiled environment step function, most of our code can execute outside of Python. This greatly increases performance, as will be demonstrated in Section 6.2.

5.2.6 Polyak averaging with opponent freezing

While the above methods help stabilise and speed up training, they do not directly address the problem of strategy collapse inherent in self-play. The learning team might overfit on defeating a current strategy, instead of finding a more general (balanced) strategy that works against different opponent strategies.

For our first self-play algorithm we decide to simplify the problem by training only a best response strategy, using reinforcement learning, for one team against an average strategy for the other team. This reduces the need to generate additional experience for both teams, as described in Section 3.4.4. Furthermore, instead of running two sets of reinforcement learning algorithms and two sets of supervised learning algorithms, described in Algorithm 2, we run only one of each. This is possible because the environment is identical, with identical optimal strategies, for both sides, and we can use the same best response and the same average strategy for both of them.

We would like to use an on-policy algorithm as it is generally known to result in more stable training [26]. This is especially important in our case, as we might have instabilities caused by 22 agents updating their policies, as well as the self-play setting. Therefore we

[1] TensorFlow: https://www.tensorflow.org/

cannot assume that the reinforcement learning algorithm can train on samples from both the best response strategy, $(\boldsymbol{\pi}_j^k, \boldsymbol{\pi}_j^{-k})$, and the average strategy, $(\boldsymbol{\beta}_j^k, \boldsymbol{\pi}_j^{-k})$. In the online case we need on-policy data and therefore we need to set $s = 0$ in Algorithms 2 and 3, i.e. we are completely in the best response setting. This would only be possible if we assume that the supervised learning algorithm is incremental in its updates.

In practice, direct use of supervised learning would require an extremely slow learning rate and an optimiser without momentum, such that the learning rate does not change dynamically per parameter. If the supervised learner updates too quickly, self-play training would collapse because the learner would start overfitting to recent strategies. To reduce the number of required computations we forgo training in a supervised learning setting entirely. Instead we make the approximation that a policy trained iteratively over a batch of data can be approximated by the average parameters of the policies for each iteration. This leads to a number of benefits. Firstly, we can now store only the parameters of our policies over various iterations instead of needing a large memory over the actions of those policies. Secondly, we now need to calculate only an average, instead of performing supervised learning, which is much less computationally expensive.

The last simplification we make is to not directly calculate the average of all previous policies, but instead rely on a moving average calculation using Polyak averaging [68]. This theory is based on 2 player games. We can regard our soccer setup as a 2 player game by viewing each team as a player, where the action of each team is just the aggregate of all players in that team. We therefore assume a sparse network setup for the team where there are 11 different modules that break the team observation into 11 parts. We also assume each team uses the same networks for every agent in that team. This has computational benefits, as discussed later in this section.

We update the average strategy by using Polyak averaging, defined as

$$\theta_2 \leftarrow \alpha_s \theta_1 + (1 - \alpha_s)\theta_2, \tag{5.1}$$

where θ_2 is the slow-moving average network parameters, and θ_1 the current network parameters that are being updated using the reinforcement learning algorithm. The value of α_s is a small positive constant. We do not let this value decay as is specified in fictitious play, and therefore lose the guarantee of converging to a Nash equilibrium. However, because we are using neural networks to approximate the best response and average strategies, and we are using a version of generalised fictitious self-play, we do not have this guarantee to begin with.

If we set α_s to be very small the reinforcement learning algorithm should have enough time to converge to an approximate best response, and allow the average strategy to incorporate a wider range of historic strategies. As stated in Section 3.4.1, at the point where the learner (best response calculator) and the Polyak averaged opponent (average strategy) converge on the same set of parameters we can conclude that we have arrived at an approx-

imate Nash equilibrium. However, because we are using an iterative algorithm we might arrive at a local maximum that is not a Nash equilibrium. We need to set α_s to be low enough so that a large range of strategies are averaged over, but not too large that the opponents learn too slowly and training stagnates.

To further combat strategy collapse the weights of the slow-moving average opponent team are frozen if the learning team has an average game outcome of less than 0.55. Thus the slowing-moving average strategy waits for the learner if the learner is struggling to beat it consistently. This prevents the Polyak averaged opponent from updating too fast, and stabilises training.

5.2.7 League training

As an alternative to Polyak averaging we also experiment with directly training against a league of opponents, as discussed in Section 3.4.5. In our work we save the policy parameters every 10 iterations to form part of this league. We do not use the same sample update rule as Berner et al. [11] and defined in equation 3.83. Instead we decided to design our own update rule that is a bit simpler in nature. We want the sampling rule to sample stronger opponents with higher probabilities, and weaker ones with lower probabilities. We also want the probability of sampling a very weak opponent to approach 0 as the win rate of that opponent approaches 0. With this in mind we define the probability of sampling an opponent team to be

$$p_i = \frac{(w_i)^\psi}{\sum_{j=1}^{N}(w_j)^\psi}, \tag{5.2}$$

where w_i is the win rate of the opponent over the current policy. If $w_i = 1$ the opponent always defeats the current policy, and if $w_i = 0$ the policy always defeats the opponent. ψ is a user defined non-negative real value. If $\psi = 0$ all previous opponents will be sampled at equal probability, similar to the classic fictitious play algorithm. If ψ is set to be a large value only the opponent with the highest win rate will be sampled. This is approximately the same as learning against only the current best policy. We set $\psi = 2$ which prioritises opponents with higher win rates, but still samples opponents that have slightly lower win rates. Opponents with win rates close to zero are hardly ever sampled. To calculate the win rate for each opponent we average the past 1000 games played between the policy and the opponent. If less than 100 games have been played we keep the win rate at 0.5. This prevents a new opponent from never being sampled again if it loses an initial game.

5.2.8 Shared weights

We use shared policy and critic networks for all the agents in a team. This means that all the generated experiences can be used to train one policy and one critic network, which drastically reduces the number of parameters to be learned, compared to individual networks

per agent. A player's starting location can still be incorporated into the observations, to enable the learning of different behaviours depending on positions.

5.2.9 Policy batching

To decrease the time it takes to generate actions in the environment, we process the policy forward pass of each agent in a team in parallel through batching (parallel matrix operations). Therefore each team performs only one batched policy calculation instead of 11 individual forward passes. According to our experiments, this simple change can make execution up to 3× faster.

5.2.10 Network setup

The critic network architecture we use is shown in Figure 5.1. It takes as input general information such as the ball's location and time, and encodes it using a 2-layer feedforward neural network called `encode_ff`. The encoded information is fed to `query_ff` to generate query embeddings for the multi-headed attention layer. The state information of each of the team's players is fed through `encode_team_ff` to generate team embeddings. The state information of each of the opponent team's players is fed through `encode_opp_ff` to generate opponent embeddings. These player embeddings are all concatenated with the query embeddings (concatenation blocks are not shown to simplify the diagram) and passed through `value_ff` to generate values for the attention layer. The values are concatenated and, with the query embeddings, fed to the multi-headed attention layer. Finally, the weighted output values of the attention layer as well as the ball and time encoded embeddings are sent through `out_ff` to output a scalar value representing an estimate of the expected discounted rewards for the team. This predicted value is subtracted from the real cumulative reward encountered, to generate an advantage estimate which indicates whether the actions agents took were better or worse than expected, to make these actions more or less likely in the future. For the 22 player setting, i and j in Figure 5.1 would both be equal to 11.

Similar to Liu et al. [36], we use an attention based component for the policy and critic networks. It allows for input vectors of different sizes to be fed to the network, depending on how many agents are observed in the case of the policy, or how many are on the field for the critic. Thus agents can, for example, be trained in a 4 player environment and subsequently evaluated in a 22 player environment. Liu et al. [36] fed a pairwise concatenation of agent embeddings to their critic network. We choose not to use pairwise embeddings, as they scale quadratically in computational cost with the number of agents, and instead opt for a simpler embedding per agent approach, which scales linearly with the number of agents and enables resource-efficient training in the full 22 player setting.

The policy network, shown in Figure 5.2, uses much of the same configuration as the critic network, with minor alterations. Due to the environment being partially observable

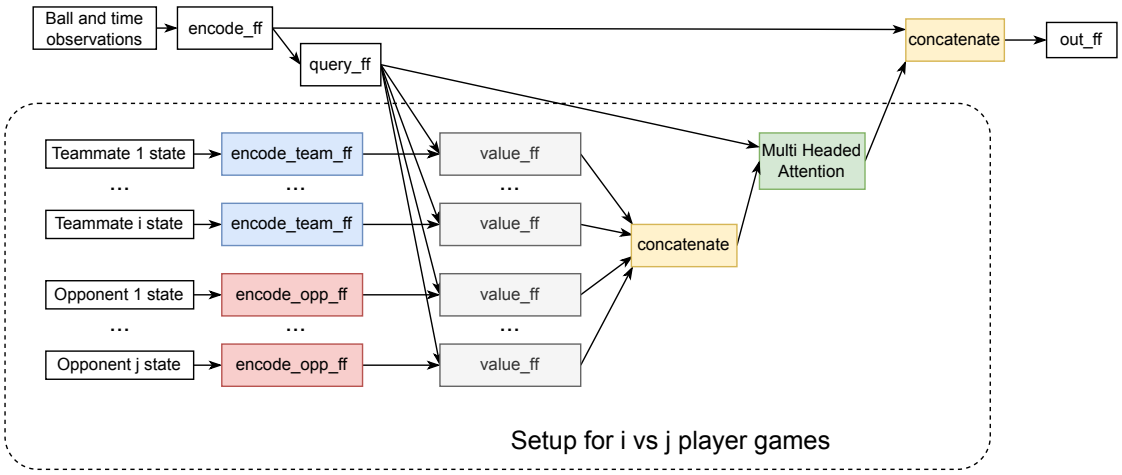


Figure 5.1: The architecture of the critic network in our PPO implementation. This network is designed to be agent order invariant, and produces the same output for any order in which agents are presented as input. It can also take as input a varied number of player information due to the use of a multi-headed attention layer. The `encode_ff`*,* `query_ff`*,* `encode_team_ff`*,* `encode_opp_ff`*,* `value_ff` *and* `out_ff` *blocks all represent 2-layer fully connected feedforward neural networks.*

we use a recurrent policy where `out_r` is represented by a 2-layer LSTM network. This gives the agents an ability to remember previous observations and internal calculations. We also feed the player's location, with the ball and time observations into `encode_ff`. Lastly, we feed in only teammates (i players) and opponents (j players) that the player can see in its 180° vision range, into the attention mechanism. All observations of the ball, teammates and opponents are also converted to the egocentric viewpoint.

5.3 Mava framework

During the course of this study we contributed to the design of an open-source multi-agent computational framework called Mava [47][2]. Mava is a Python library that can be used to easily create multi-agent reinforcement learning systems and run experiments at scale over multiple processors (CPUs and GPUs). Our work on the Mava framework was done in partnership with InstaDeep[3], during a research internship on multi-agent systems. The Mava framework uses TensorFlow for its neural network and training procedures.

Mava extends a popular single-agent reinforcement learning library called Acme [29] to the multi-agent setting. Its design is closely aligned with the distributed A2C architecture, discussed in Section 3.3.4, with special focus on the multi-agent setting.

Figure 5.3 shows the computational setup for Mava. In this figure the trainer and execu-

[2]Mava: https://github.com/instadeepai/Mava
[3]InstaDeep: https://www.instadeep.com/

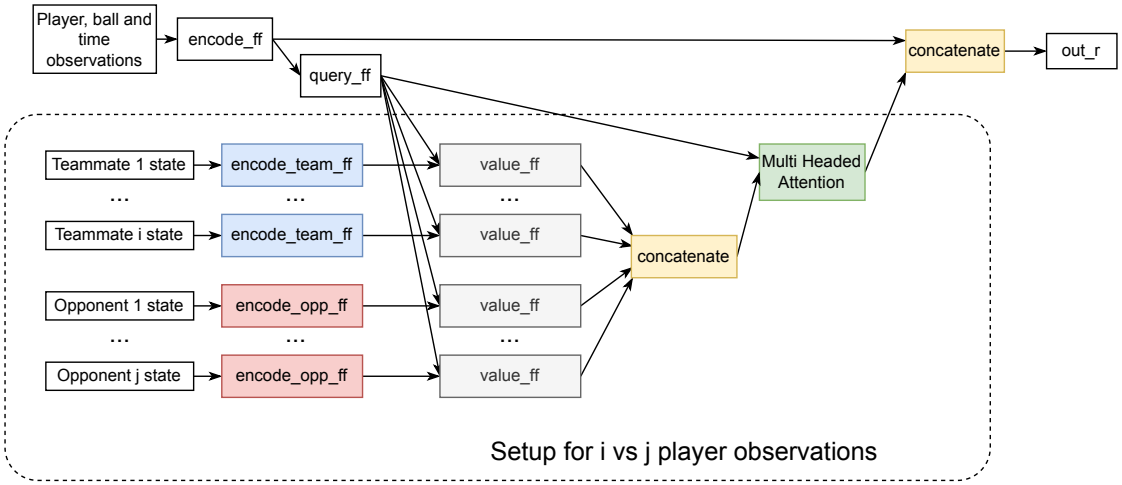


Figure 5.2: The architecture of the policy network in our PPO implementation. Here i and j represent the number of teammates and opponents the player sees at any given time, which dynamically change throughout a game.

tor processes are multi-agent versions of the learner and worker processes presented in the distributed A2C diagram (Figure 3.8). The image on the left side of Figure 5.3 shows the system setup with the executor, dataset and trainer all running in separate processes. The executor is tasked with interacting with the environment by receiving observations and rewards, performing internal actor policy runs (one for each agent), and returning the actions to the environment. The executor then sends this experience to the dataset process which focuses on transporting and storing experience. In Mava we use a Python library called Reverb [15] to create the dataset server. Data can be transferred much faster with a dedicated data server than just sending information from one process to the next. The trainer samples experience from the dataset server to update its internal network parameters. The trainer consist of one or multiple learners, where each learner updates a set of parameters. If all the agents share weights there can be a single learner, or if they have independent networks there can be a learner per agent. The executors periodically request the latest parameters from the trainer to ensure that the experience being generated remains relevant.

The image on the right side of Figure 5.3 shows a distributed Mava system with multiple executors. As stated in Section 3.3.4, the experience generators are usually the bottleneck in training reinforcement learning systems. The trainer performs accelerated matrix operations on a dedicated processor like a GPU or TPU, meaning it might have to wait for experience from the slower executors running on a CPU. In the distributed Mava system one can initiate multiple parallel instances of an executor, all running on separate processes, which can generate much more experience than with only one executor. This drastically increases the number of episodes generated per second and therefore accelerates training. To run all these parallel processes we use a Python library called Launchpad [73].

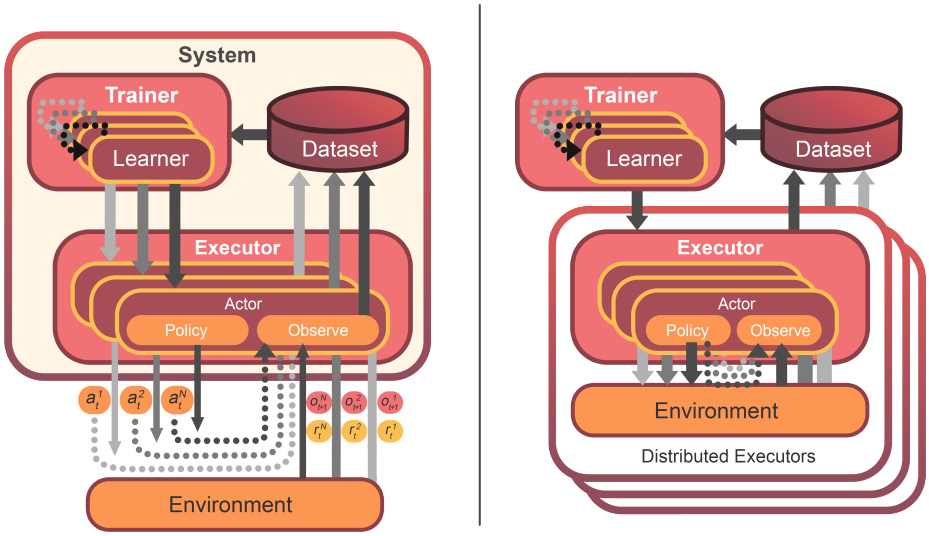


Figure 5.3: The interaction between the environment and different system components inside the Mava framework. ***Left****: The entire Mava system with all its processors, including the executor (worker), dataset and trainer (learner).* ***Right****: The distributed system with multiple executor processes and a trainer process running in parallel. Sources: Pretorius et al. [47].*

Another advantage of using Mava is that it allows for easy experimentation with different multi-agent algorithms and architectures, as indicated in Figure 5.4. For example, we could start by instantiating a multi-agent reinforcement learning system to learn to play soccer in the decentralised setting, where 22 single-agent algorithms all interact with the same environment. An independent learner will then be created for each agent and can update only that agent's parameters. We can also easily experiment with other architectures without needing to completely rewrite the reinforcement learning code. Mava allows us, for example, to switch to the popular centralised training and decentralised execution paradigm, by just changing the executor and trainer arguments. This becomes especially useful in the initial stages of development where the best solution might not be obvious.

Mava also provides various popular multi-agent algorithms out of the box, including MA-PPO (used in this work), MA-DQN, VDN [63], QMIX [49], DIAL [20], MA-DDPG [37] and MAD4PG [10]. All these algorithms use the same environmental interface. After we adapt our custom soccer environment to conform with this interface it becomes easy to swap reinforcement learning algorithms and therefore experiment to find which one works best. For the purposes of our work we focused on implementing the MA-PPO, MA-DDPG and MA-D4PG systems in the Mava library. However, MA-PPO was the only algorithm found to be stable enough in the 22 player setting.

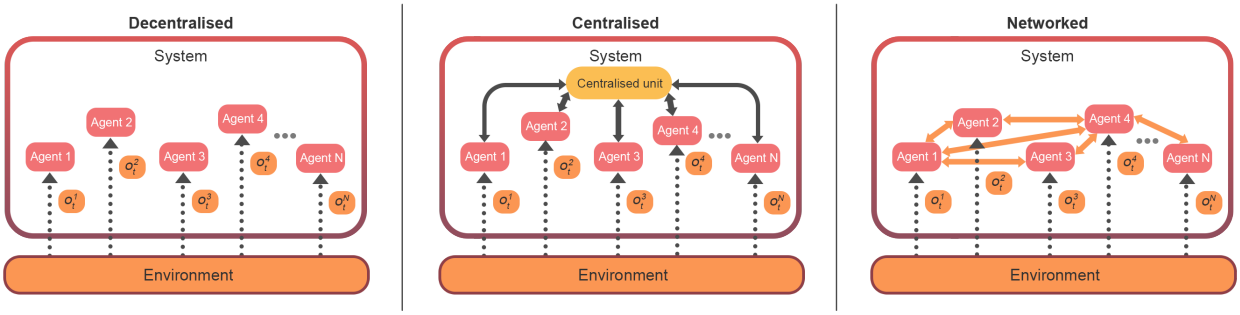


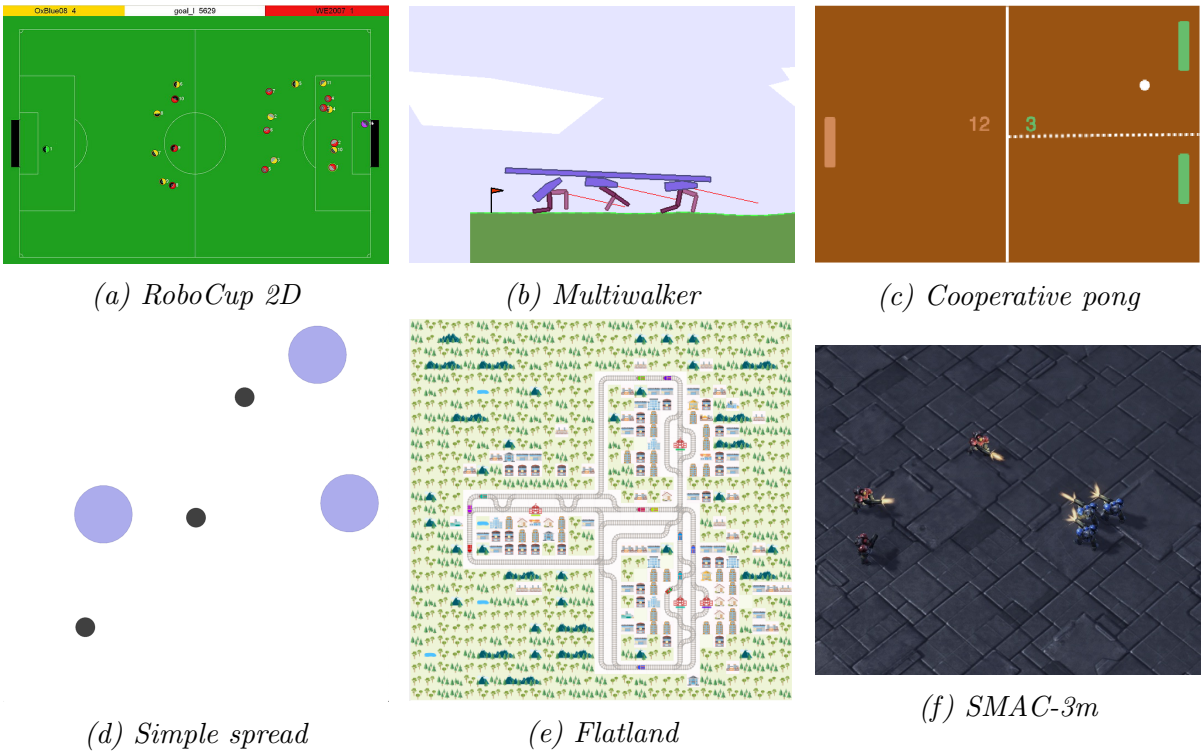
Figure 5.4: Different training architectures available in Mava. ***Left****: The decentralised execution and decentralised training architecture (also known as independent reinforcement learning algorithms).* ***Middle****: The popular decentralised execution and centralised training architecture.* ***Right****: A hybrid networked approach where custom communication setups between agents can be specified. Source: Pretorius et al. [47].*

When an algorithm fails to learn in a new (custom) environment, it might be difficult to determine whether something is broken in the environment or in the learning algorithm. It might be helpful if one can easily experiment with multiple environments and, as seen in Figure 5.5, Mava provides support for a host of different environments out of the box including our 2D RoboCup simulation environment wrapper. During experimentation one can easily swap both the algorithm and environment, as all the algorithms in Mava use a common environment API. The only requirement is that the chosen algorithm works with the action space provided by the environment (e.g. discrete or continuous).

In our work we regularly experimented with the simple spread environment which acts as a debugging environment in Mava. As the name suggests, it is simple to solve and therefore one can observe convergence results within a minute or two and know whether a particular algorithm is broken or not. If the algorithm worked in this environment, we could move to our custom 2D soccer environment, which is not included in Mava but still conforms to the Mava environment API.

5.4 Abandoned research directions

In this section we describe some research directions that we initially investigated, but later abandoned for other ideas that seemed more promising. While this is not an exhaustive list, we think that these methods are worth mentioning, and we present reasons for why we abandoned them. This can serve as a guide for research that builds on this work.



(a) RoboCup 2D (b) Multiwalker (c) Cooperative pong

(d) Simple spread (e) Flatland (f) SMAC-3m

Figure 5.5: Different environments implemented inside the Mava framework, for fast experimentation. Source: (a) Noakes [44], (b) Terry et al. [66], (c) Diallo et al. [19], (d, f) Schafer [55], (e) Reshef [50].

5.4.1 Ray-traced agent vision

When we first constructed our custom 2D soccer environment, we wanted it to align with not only RoboCup but also with what might be expected from real robots. Therefore we looked at a dense vision system for agent observations. An illustration of this vision system is presented in Figure 5.6.

As can be seen in the figure, the vision system consists of light detectors spread out evenly over a 180° area, in an egocentric manner (from the agent's perspective). In this case there are 91 rays, and the first object each ray hits is registered. For each input ray the agent gets the object identifier (in the form of a one-hot encoded vector) and distance to that object. This allows an agent to estimate where the object is in 2D space. Only the circle component of a player's body is considered a hit, to further simplify the ray tracing. If a ray does not hit any player or the ball, it will either hit one of the goal posts or a field boundary line. The rays are calculated in a manner similar to ray tracing [69], but simplified to drastically reduce simulation time. Instead of tracing each ray's path to determine which object it hits first, we simply consider every object and determine if the agent would see it. If it does, the corresponding rays that hit the object are set, while checking if a closer object has not yet been found for each of those rays. The same calculation is performed for the two

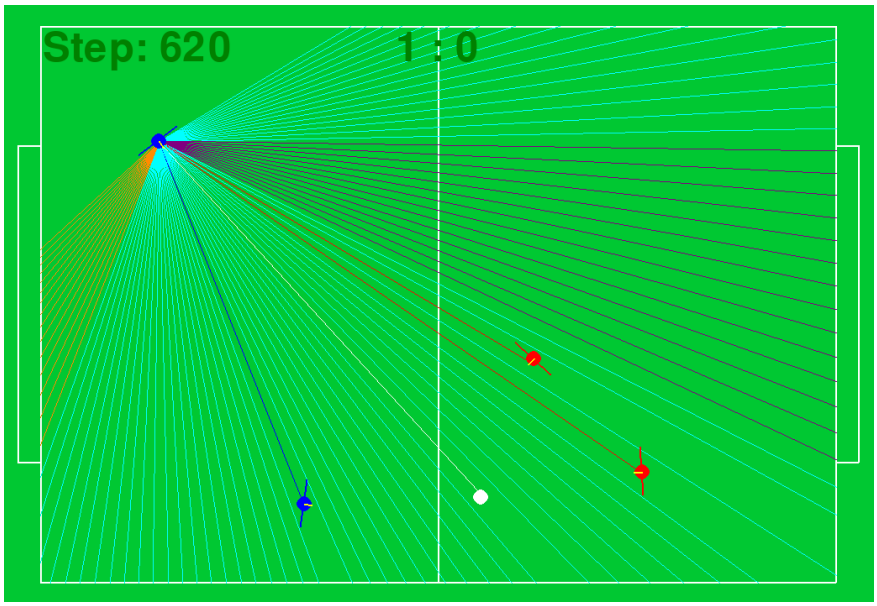


Figure 5.6: A generated view of the simplified ray-traced vision system in our custom 2D soccer environment.

goal posts. The rest of the rays are set to the field outline with the correct distances. This method has an order of magnitude performance increase over standard ray tracing, but only for environments with a small number of objects as is the case in this soccer environment.

This ray based vision system solved many of the initial challenges we had with partial observability in our soccer environment. Because the vision system has a fixed-sized input we could easily process it with a neural network to generate actions. Furthermore, because there are spatial correlations over ray inputs, we could apply a 1D convolutional neural network [6] as a feature extractor for the agent's policies which allowed the agents to learn much faster and was able to outperform a team, with a handcrafted naive strategy, in the 4 player setting entirely through self-play.

However, this vision system does have limitations. The first is that although it is faster than ray tracing, it still requires more computational resources to calculate than just returning the egocentric distances and velocities of the players visible to a particular agent. Secondly, because the rays are discrete, an agent only has access to approximate information about what is in front of it, and if the ball or another player is located between two rays the agent might not see it. Our attention mechanism overcomes both these problems. The official RoboCup environment also provides a list of objects that each agent sees, and therefore using attention would make our algorithm better suited for that environment. The ray based system does however seem more plausible when one wants to train agents in a simulation that could be adapted to the real world, and therefore might still be useful in future work.

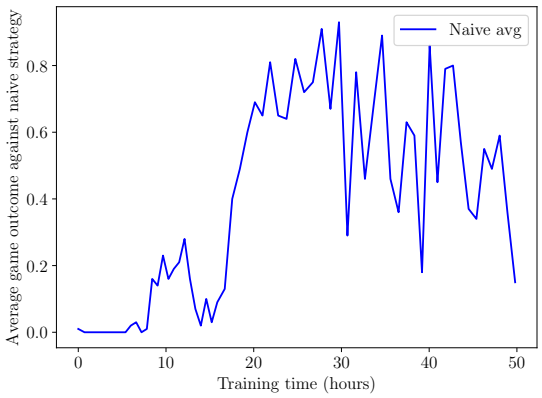
5.4.2 Other reinforcement learning algorithms

We use PPO as our main reinforcement learning algorithm, due to its good track record of stable training in multi-agent environments. However, one shortcoming of PPO is that it uses state information that excludes the actions to estimate the discounted expected reward that a team will receive. Because the critic does not take individual actions into account, all agents get rewarded or punished by the same amount per time step. Even if one agent is performing well it would be punished if another agent happens to perform badly. If actions were part of the input, the critic could estimate how much each agent should be punished or rewarded individually for their actions. We investigated three different algorithms to try and accomplish this. This first is multi-agent deep deterministic policy gradient (MA-DDPG) [37], which is an off-policy algorithm that directly updates a policy's actions by trying to maximise the predicted expected reward output of the critic. Therefore actions that are expected to result in more reward in the future are strengthened, while others weakened. This should allow the critic to individually adjust player actions, which would result in more accurate updates than in PPO. However, it assumes that the critic is an accurate estimator of the expected rewards, which might not be the case. The second algorithm we investigated is the multi-agent distributed distributional deep deterministic policy gradient (MA-D4PG) algorithm [10], which enhances the MA-DDPG algorithm by including a distributional head of the critic in order to converge faster to an accurate estimation of the discounted reward. We also designed our own algorithm that we call hybrid PPO, which is exactly the same as PPO except that it also takes as input all the actions performed by each agent. However, instead of using the PPO advantage estimate presented in equation 3.63, we use the following:

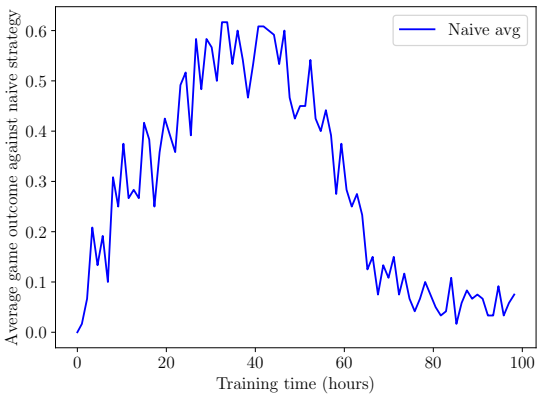
$$\hat{A}_t = v_w(s_t, a_t) - v_w(s_t, \hat{a}), \tag{5.3}$$

where $v_w(s_t, a_t)$ is the value estimated by the critic of the action taken. Here $\hat{a}$ represents the expected action that an agent would take in state s_t. The expected action can easily be calculated for a multivariate Gaussian distribution, as it is the mean. For a bounded action distribution we approximate the mean by sampling 10 actions and averaging them. The intuition behind equation 5.3 is that we are updating the policy based on how much better the agent's action was compared to what the critic expected the agent to do. This stands in contrast to calculating how much better the team's actions where as a whole over the expected reward for that state. In Figure 5.7 we present the performance of each of the algorithms over training time (hours). The vertical axis represents a learning team's average score against a naive strategy which is a handcrafted opponent described in greater detail in Chapter 6.

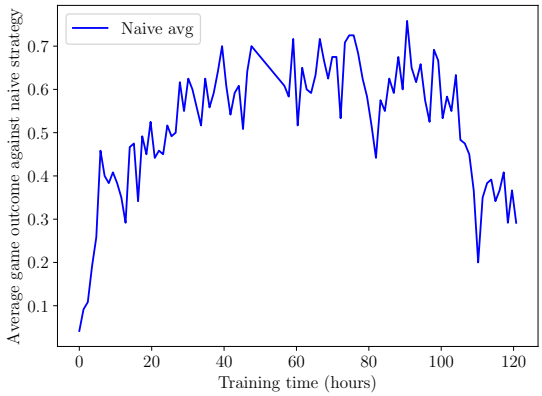
As can be seen in the figure, all the algorithms have a stability problem. They all defeat the naive team at some point in training, but collapse shortly after. MA-DDPG seems to be



(a) MA-DDPG



(b) MA-D4PG



(c) Hybrid PPO

Figure 5.7: Average score against a naive handcrafted team for different algorithms that are training in the 22 player self-play setting. The outcome of a game would be 1 for a win, 0.5 for a draw, and 0 for a loss.

the most unstable (although achieving the highest performance before collapse), while MA-D4PG has slightly more stable training for about 40 hours before collapsing. Our hybrid PPO algorithm seems to exhibit the most stable training, but also collapses after around 100 hours of training. We experimented with various adaptations of these three algorithms, but unfortunately could not get any of them to train stably in the full 22 player setting. However, we still believe that they might have the potential to outperform PPO due to their focus on action-specific updates. Our main guess as to why they did not perform well in our experiments is due to the added complexity for the critic of also needing to model the agent actions. This places additional strain on the critic that already has to deal with 22 players in an ever changing environment due to the self-play component.

5.5 Summary

In this chapter we discussed various design decisions in our development of a system that can produce stable training in the full 22 player setting. In Section 5.1, we presented some of the challenges this setting introduces that make it difficult to apply standard reinforcement learning algorithms. In Section 5.2, we presented our solutions to these challenges. We designed our system to have a team of learning agents compete against a slow-updating opponent strategy. The idea behind this opponent is that it represents an average strategy over the learning team's history, and therefore the learning team needs to derive a strategy that is generally better. As an alternative we also created a league training algorithm that directly plays the learning team against past policies.

A challenge in our soccer environment is that the observation space changes in size throughout an episode, depending on how many other players are seen by an agent. To accommodate for this we decided to use an attention based architecture that is invariant to input size. We also use attention for the critic even though it has a fixed input size, because attention has useful weight sharing inductive biases that reduce the number of parameters to be learned. We add recurrent units to the policy network, which provide the agents with a memory to remember past information.

To improve the speed with which experience is generated we implement various optimisation improvements. The first is the use of parameter sharing across all agents in a team. This not only reduces the number of parameters that need to be learned, but also allows us to batch process policy calculations. Two batched sets of matrix operations, one per team, are much faster than sequentially executing each policy per environment step. We use both TensorFlow's tf.function and JAX's JIT capabilities to compile the worker, learner and environment code for much quicker execution. As described in Section 5.3, we use the Mava framework which allows us to run multiple workers in parallel on the same CPU device, which also greatly improves throughput.

Finally in Section 5.4, we mentioned a few methods that we investigated and later abandoned for better alternatives.

Chapter 6

Experimental results

In this chapter we evaluate the performance of the PPO algorithm with the various improvements proposed in the previous chapter. In Section 6.1, we specify the framework, opponents and hardware used in our experiments. In Section 6.2, we investigate the extent to which various code acceleration methods reduce computation times. In Section 6.3, we compare our team trained in the 22 player setting (with Polyak averaged and league opponents) against various handcrafted strategies, as well as against a team of 11 players trained in the 4 player (2 vs 2) setting. We provide an ablation study in Section 6.4 to indicate how some of the main improvements influence a team's performance. In Section 6.5, we investigate possible reasons for why the best team trained in the 22 player setting might be performing better than a team trained in the 4 player setting. In Section 6.6, we provide a comparison of our work against existing work. In Section 6.7, as a final experiment to demonstrate transferability, we port our algorithm to the wrapped RoboCup environment. We end the chapter with a summary of our results in Section 6.8.

6.1 Experimental design

In this section we provide information on the different components used in our experimental setup.

Multi-agent reinforcement learning framework: As mentioned in Section 5.3, we leverage the open-source multi-agent reinforcement learning framework Mava [47]. Mava can split the training procedure into experience generators (workers) and a network updating process (learner), which can be run in parallel to increase efficiency.

Handcrafted opponents: The aim of this work is to show that stable training is possible in the full 22 player soccer game. To evaluate whether our PPO agents do improve over time, when training only through self-play, we evaluate them against three opponent teams with fixed, handcrafted strategies. We have a team of agents that move randomly on the field, representing our control team, as well as two other teams where the more difficult one is loosely inspired by real soccer strategies. The three teams can be described as follows:

Table 6.1: Hyperparameters used in the 4 and 22 player training settings.

Parameter	Value
Critic loss weighting (α_v)	0.5
Policy loss weighting (α_p)	1.0
Entropy loss weighting (α_e)	0.01
Polyak averaging update rate (α_s)	0.01
PPO clipping value (ϵ_p)	0.1
Opponent update threshold (ϵ_s)	0.55
Reward discount factor (γ)	0.999
epochs_per_step	5
Learning rate in the Adam optimiser [31]	10^{-4}
Experience batch size	660
Mini-batch size	60
Episode length	400
LSTM layer size	32
Feedforward layer size	64

- Random (easy): This team simply outputs random actions and is used as a control.
- Naive (medium): Players search for the ball and move towards the ball if it is found. If a player is close enough to the ball, it kicks it in the direction of the opponent's goal. Therefore, this team disregards cooperation and simply tries to score goals as fast as possible.
- Teamwork (difficult): This team consists of defensive and offensive players. The offensive players try to move towards the ball and kick it towards the opponent's goal (similar to the naive team). The defensive players form a protective semi-circle around their own goal, moving only when the ball is close to their goal and then trying to kick the ball towards the side of the field. This strategy performs better than naively chasing the ball.

Hardware specifications: A key motivation for this work is to reduce the computational requirements as much as possible, to allow researchers with limited resources to reproduce and build upon our results. For every training run a single machine with 68 GB of RAM was used. We have all the workers running on a CPU with 12 cores (Intel® Core™ i7-8700K @ 3.70GHz). One learner is used and runs on a GP104 GPU (NVIDIA GeForce GTX 1080) with 8 GB of RAM.

Hyperparameters: Details on the hyperparameters and network layers we use can be found in Tables 6.1 and 6.2.

Table 6.2: Neural network layers used by the policy and critic.

Name	Type	Description
`encode_ff`	2 layer feedforward neural network	Used to encode general information such as the ball's location and time information.
`query_ff`	2 layer feedforward neural network	Used to generate query embeddings for the multi-headed attention layer.
`encode_team_ff`	2 layer feedforward neural network	Used to generate team embeddings.
`encode_opp_ff`	2 layer feedforward neural network	Used to generate opponent embeddings.
`value_opp_ff`	2 layer feedforward neural network	Used to generate values for the multi-headed attention layer.
`out_ff`	2 layer feedforward neural network	Produces a final scalar output representing the critic's estimate for the cumulative reward remaining in the episode.
`out_r`	2 layer LSTM network	Produces weightings for a probability distribution over the actions an agent can take.

6.2 Code acceleration

To reach competent strategies in the 22 player setting, given our limited computational budget, an emphasis is placed on code efficiency. We implemented various techniques that accelerate different parts of our code execution without changing the underlying computations.

6.2.1 Environment performance

In this section we investigate the relative speed differences of our custom (JAX based) soccer environment[1], DeepMind's simplified soccer environment[2] [35] and our wrapped version of the RoboCup 2D simulation environment[3]. The execution speeds achieved in each of these environments are shown in Figure 6.1, where each player is controlled by a random policy with negligible internal compute. Here we focus only on the environment step speed. As can be seen, the custom environment has the highest steps per second throughput. In the 1 player per team setting, the custom environment runs at 4670 steps per second, which is 7.6× faster than the wrapped RoboCup environment and 13.6× faster than the DeepMind environment. In the 11 players per team setting, the custom environment is 25× and 102× faster than the RoboCup and DeepMind environments, respectively. With 22 players on the field our custom environment runs at 1533 steps per second. This does not imply that the custom environment is better than the other two environments, as each environment

[1]Our custom environment: https://github.com/DriesSmit/MARL2DSoccer
[2]DeepMind's environment: https://github.com/deepmind/dm_control
[3]Wrapped RoboCup 2D environment: https://github.com/instadeepai/Mava

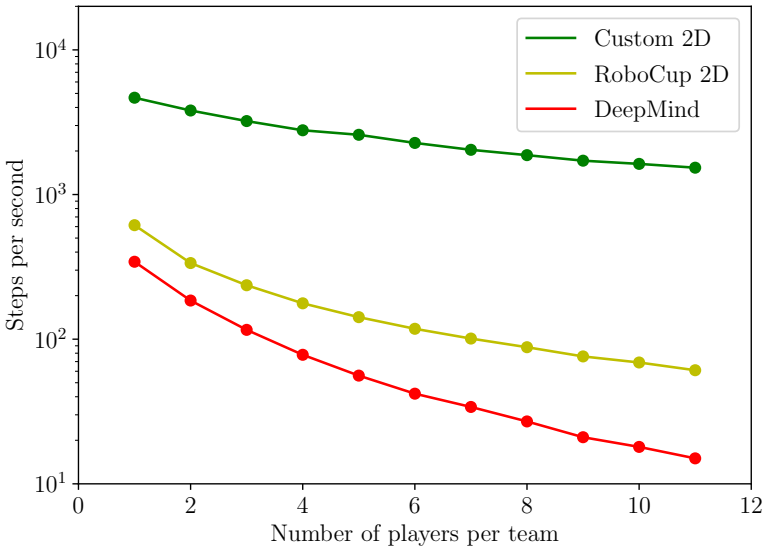


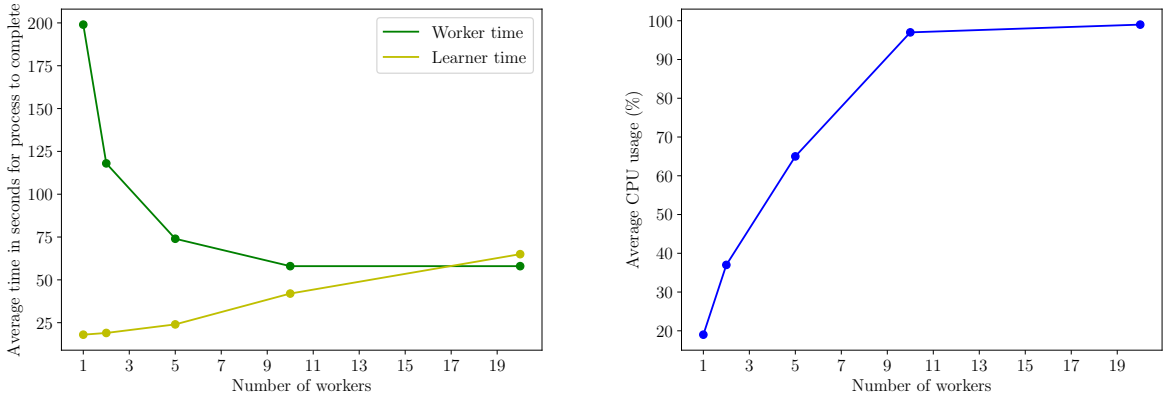
Figure 6.1: The environment steps per second throughput for different 2D soccer environments. The horizontal axis represents the number of players per team, and the vertical axis the steps per second of the environment in log scale.

represents different complexity trade-offs. The custom environment focuses on simplified physics for faster training in the multi-agent setting, where the DeepMind environment has more realistic physics (using the MuJoCo engine [67]). The RoboCup environment is the most realistic soccer engine of the three, as it includes properties such as stamina, fouls, kicking the ball out of bounds, communication and independent head movement. As mentioned, with random policies the custom environment is 25× faster than the RoboCup environment. When using the full 22 player training setup with neural network policies, the custom environment is still 14× faster than the RoboCup environment. It is also easier to learn in the custom environment as it has fewer time consuming dynamics such as throw-ins, fouls and players slowing down due to limited stamina. We can therefore iterate much faster on algorithmic ideas in the custom environment than would otherwise be possible. We therefore focus on the custom environment for the rest of our experiments, and return to the RoboCup environment towards the end of this chapter.

6.2.2 Optimal number of workers

As mentioned in Section 6.1, for all our experiments we use a single CPU with 12 cores and a GPU. To increase the number of episodes we can generate per second we run multiple workers in parallel using the Python library Launchpad [73]. We would like to find the optimal number of workers that would maximise the experience being generated, while still ensuring that the learner can process the experience fast enough. To determine this, we run an experiment where we measure the time it takes to execute 100 episodes in the full 22

player setting. We set up the environment with two sets of policies running for both teams. In the initial reward shaping part of training, one team performs random actions. After the short initial reward shaping, we start with self-play training where we use two neural network policies. We focus on the second setting as our system trains mainly in this setting. We monitor CPU usage to verify whether we are approaching diminishing returns as more workers are added. Each worker runs $100/n$ games, where n is the number of workers. The learner trains on one epoch of experience. If the learner takes more time than the workers, it means the workers are generating experience faster than the learner can learn from it. We set the mini-batch size of the learner to 60 as that is the maximum size we can fit into 8 GB of GPU memory. The results of this experiment are presented in Figure 6.2.



(a) worker and learner times *(b) CPU usage*

Figure 6.2: ***Left****: Average processing time for one worker (green) and learner (yellow), for an increasing number of workers. Notice that at 20 workers the learner takes more time than the workers.* ***Right****: The average CPU usage of the system for an increasing number of workers.*

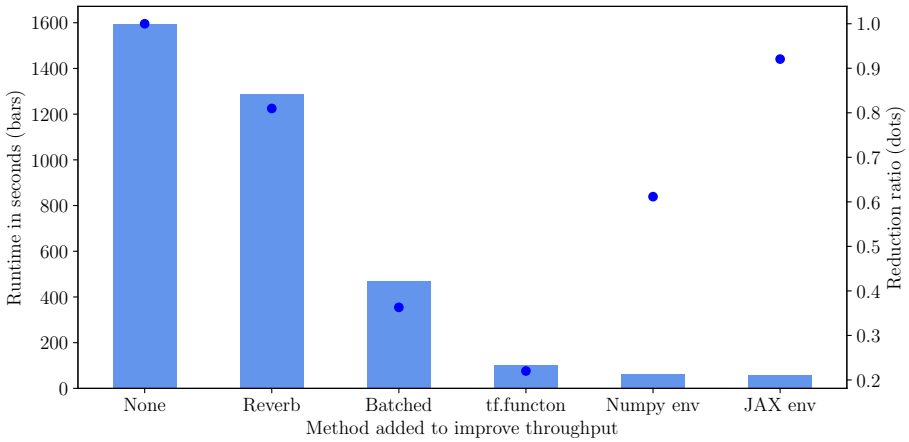
We run the experiment for 1, 2, 5, 10 and 20 workers and interpolate the rest. As can be seen in Figure 6.2, initially the time taken per worker drastically reduces as we increase the number of workers. However, as the number of workers increases from 10 to 20, we do not get any improvement in average worker time (both take about 58 seconds for 100 episodes). This diminishing return in performance can be explained by looking at the CPU usage in Figure 6.2. The 10 workers take only 2% less CPU usage than the 20 workers, as both are close to 100% usage. However, in the case of 20 workers, the learner takes more time to process the episodes than the workers take to generate the episodes. This means that the workers would be generating wasted experience. We conclude that 10 workers seem roughly optimal for our setup.

6.2.3 Other code optimisations

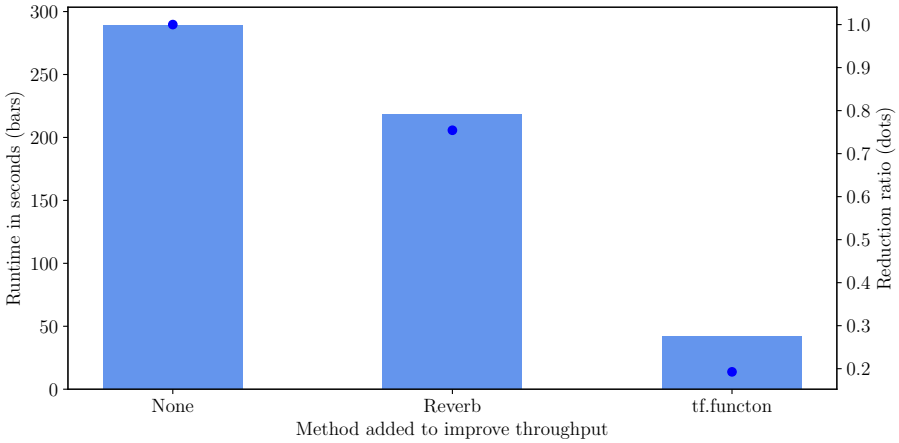
In this section we investigate how various other improvements help to reduce the overall computational times of the workers and learner. In the case of the workers, the main

components that require computational resources are the environment step function, the policy inference function, and transporting the data from the workers to the learner. In the case of the learner, the functions that take the most time are retrieving the data and performing an update to all the networks.

In Figure 6.3 we present the effects of various methods to improve throughput. Based on the conclusion of the previous section, we use 10 workers running in parallel for these experiments. The first method called 'None' in Figure 6.3 is the code running without any improvements. As we move from the left to the right on each bar graph, we keep adding methods and the improvements are therefore cumulative. The methods represented



(a) worker process time



(b) learner process time

Figure 6.3: ***Top****: Average process time for the workers, for different code acceleration methods.* ***Bottom****: Average process time for the learner, for different code acceleration methods. For each bar, all the methods to the left are also included and the blue circles indicate ratios in run time between the current and previous method (lower is better).*

in Figure 6.3 are:

- **Reverb**: adding a Reverb data server to alleviate some of the responsibility of trans-

ferring data to the learner from the worker (Section 5.3);

- **Batched**: increasing the action calculation throughput by batching the policies of the agents in a team (Section 5.2.9);
- **tf.function**: applying the tf.function wrapper to the policy forward pass and learner step function, allowing TensorFlow to compile these functions instead of executing them in Python for each iteration (Section 5.2.5);
- **NumPy env**: converting our Python environment to use parallel NumPy operations over the agents, instead of native Python floating point operations (Section 4.3);
- **JAX env**: converting the NumPy environment to a JAX equivalent environment and then compiling the environment using JIT for increased throughput (Section 4.3).

Upon first inspection of Figure 6.3 it may seem that the batched policies provide the greatest performance improvement in execution time. However, because we are adding methods cumulatively, it is actually the ratio in time saved that is important, as indicated by the blue dots. For example, the batched policies allows the worker to take 36% of the time without it, however the tf.function method is even more effective, taking only 22% of the time without it. The end result of all of these optimisations is that the workers now take only 3.6% of the original time, and the learner only 14.5%.

6.3 Full system training in the custom environment

In this section we train the full system in our 22 player custom environment and evaluate the team's performance against the naive and teamwork handcrafted strategies. We also evaluate the team trained in the 22 player setting against a team trained in the 4 player setting also using our system. The objective of this experiment is to determine whether our system (trained using self-play) can learn to outperform handcrafted strategies and also the 4 player team, without directly training against them. We train our system in both the Polyak averaged opponent and league opponent settings and compare performance.

Polyak averaged opponent: Here we train the system against a Polyak averaged opponent. We train the system in the 4 player setting beforehand for approximately 55 hours over three seeds. The best performing team in this 4 player setting is selected and forms one of our opponents which is used to evaluate our team training in the 22 player setting. We then train a new team in the 22 player setting and regularly evaluate the average score of this team against the other opponents. In Figure 6.4 we plot our training team's average game outcome against the naive team, as progress can be seen earlier than against both the team with the teamwork strategy and the team trained in the 4 player setting. We plot vertical lines where the training team on average first outperforms a specific opponent. As seen our team training in the 22 player setting first defeats the naive strategy on average

after 18 hours of training, the teamwork strategy after 31 hours, and the best team trained in the 4 player environment after 45 hours of training. These results seem to indicate that training in the 22 player setting allows for better group strategy formation that might not emerge from training in the 4 player setting.

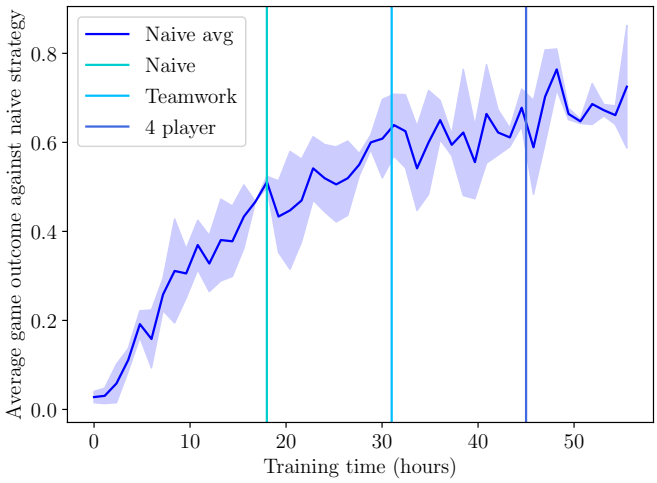


Figure 6.4: The average game outcome of a team of agents trained in the 22 player setting against a Polyak averaged opponent. The performance of the agents are plotted against the naive strategy over time (hours), with one standard deviation over three randomly seeded runs. A single game outcome can be either a loss (0), draw (0.5) or win (1.0) for the team. The three vertical bars represent points where the training team started outperforming a specific opponent team (achieving an average score above 0.5). The 4 player team (last vertical bar) uses the best policy found after training in the 4 player setting for 55 hours. All evaluations are done in the 22 player setting.

League training: Here we train the system against a league of opponents. For the opponent trained in the 4 player setting we decided to use the best team derived when using Polyak averaging. We did this because the team training against a Polyak averaged opponent performed better than the one trained against a league after 55 hours in the 4 player environment. In Figure 6.5 we again plot the average game outcome of a team training in the 22 player setting, against the naive team, and plot vertical lines where the training team on average first outperforms a specific opponent. As can be seen, the team training against a league of opponents could still beat all the fixed opponents presented to it. As mentioned in Section 5.2.7, we add the current training team's policy to the league of opponents every 10 iterations. When comparing the results in Figure 6.5 to those in Figure 6.4, it is immediately apparent that the variance in training runs for the league opponent is higher than for the Polyak averaged opponent. Teams training against the Polyak averaged opponent also seem to learn slightly faster when comparing the win rates against the naive team.

In this section we trained our system in the full 22 player environment using Polyak

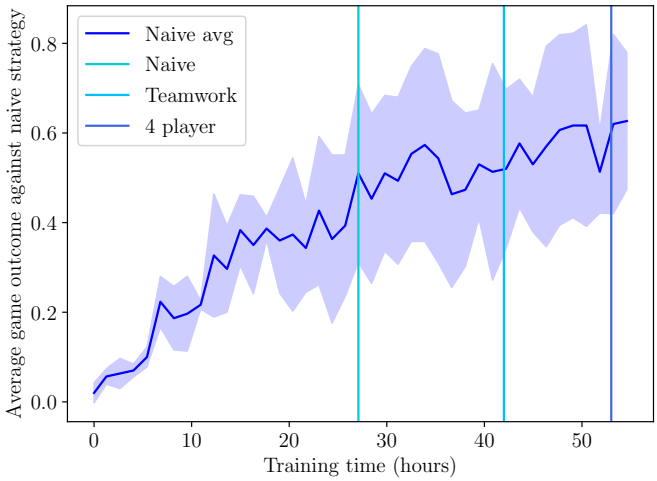


Figure 6.5: The average game outcome of a team of agents trained in the 22 player setting against a league of opponents. All other settings are the same as was the case for Figure 6.4.

averaged opponents and league opponents. Both setups managed to outperform the benchmark opponents in under 55 hours of training and also outperformed the top team trained in the 4 player setting. This seems to indicate that it might be important to train in the 22 player setting for higher-level strategy formation. Furthermore, the Polyak averaged opponent setup seems to lead to less variance than the league opponent, and higher performance at almost every point throughout training.

A team training with Polyak averaged opponents performs well when evaluated against relatively weak higher-level strategies. However, it might not perform as well when higher-level strategies become more important. This is because the training team is learning to defeat only one average strategy. The league opponent strategy might be much less susceptible to this limitation, as it can learn to defeat many opponents with different strategies. Therefore, even though league training led to worse performance in the above experiments, it might give better results as the system improves and opponents become more challenging to beat.

6.4 Ablation study

We now investigate the individual contributions of some of the main improvements on system performance. We do this to determine whether each component is necessary in the final system. We focus on training with a Polyak averaged opponent as it seems to be performing slightly better than the league opponent, and compare the complete system with two modified training systems. (1) We remove the Polyak averaged opponent and replace it with the latest policy. We also remove the freeze functionality for the opponent, such

that the team of training agents is essentially playing against themselves. (2) We replace the attention mechanism in the policy and critic networks with a feedforward layer, while keeping everything else the same. These two modified systems are evaluated because they are the main improvements that can influence the training profile (whether strategy collapse occurs or not). The other improvement methods are related to code acceleration where they execute mostly the same operations, but more efficiently. For the full system we take the training run with the median final score across three seeds.

Figure 6.6 shows results from training the full system, the system without the Polyak opponent and the system without attention mechanisms. We observe that the full system performs better than any of the modified systems. The team with feedforward networks instead of attention mechanisms does not seem to ever improve. The attention mechanism has significantly fewer parameters to be learned, through additional invariance assumptions, which seems to allow for better strategy formation in this multi-agent environment. The system without Polyak averaging does improve over time, but at a relatively slow pace. This may indicate that Polyak averaging and opponent freezing aid the discovery of more general strategies (evaluated against the naive handcrafted strategy which it never trains on), faster than pure self-play can. Without a stable opponent it becomes easier to unlearn a specific skill, as it might not be obvious from the reward signal whether a specific new behaviour is worse than average when both teams exhibit that behaviour.

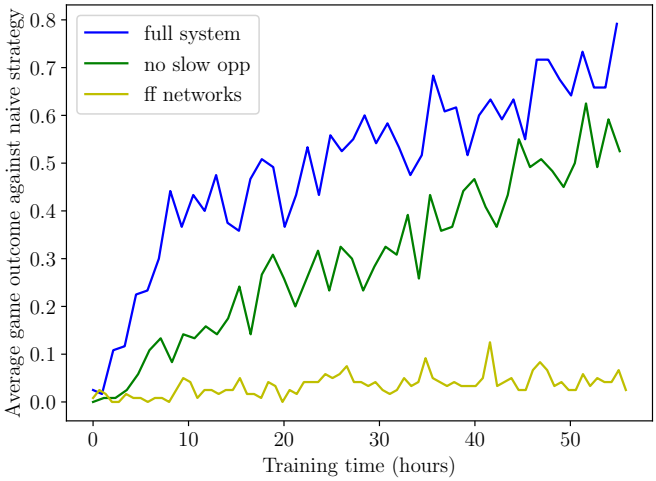


Figure 6.6: The average game outcome against the naive strategy over the course of training, with different system configurations. These training runs are performed in the full 22 player setting. The three graphs represent the complete system (blue), the system without Polyak averaging or opponent freezing (green), and the system with feedforward networks instead of attention mechanisms (yellow). The full system performs better throughout training when compared to the modified systems.

6.5 Trained agent evaluation

In this section we investigate in more detail how agents trained in the 22 player setting compare against agents trained in the 4 player setting and other handcrafted teams. We do so in an attempt to find possible reasons for why the team trained in the 22 player setting performs better than the other teams, and to determine whether training in the 22 player setting has advantages over training in the 4 player setting. We start by providing the average game outcomes of the various teams playing against one another. We then analyse the averaging positioning of our trained teams, and evaluate gameplay strategies that they might employ. At the end of the section, we perform a quantitative evaluation over certain metrics.

6.5.1 Average game results

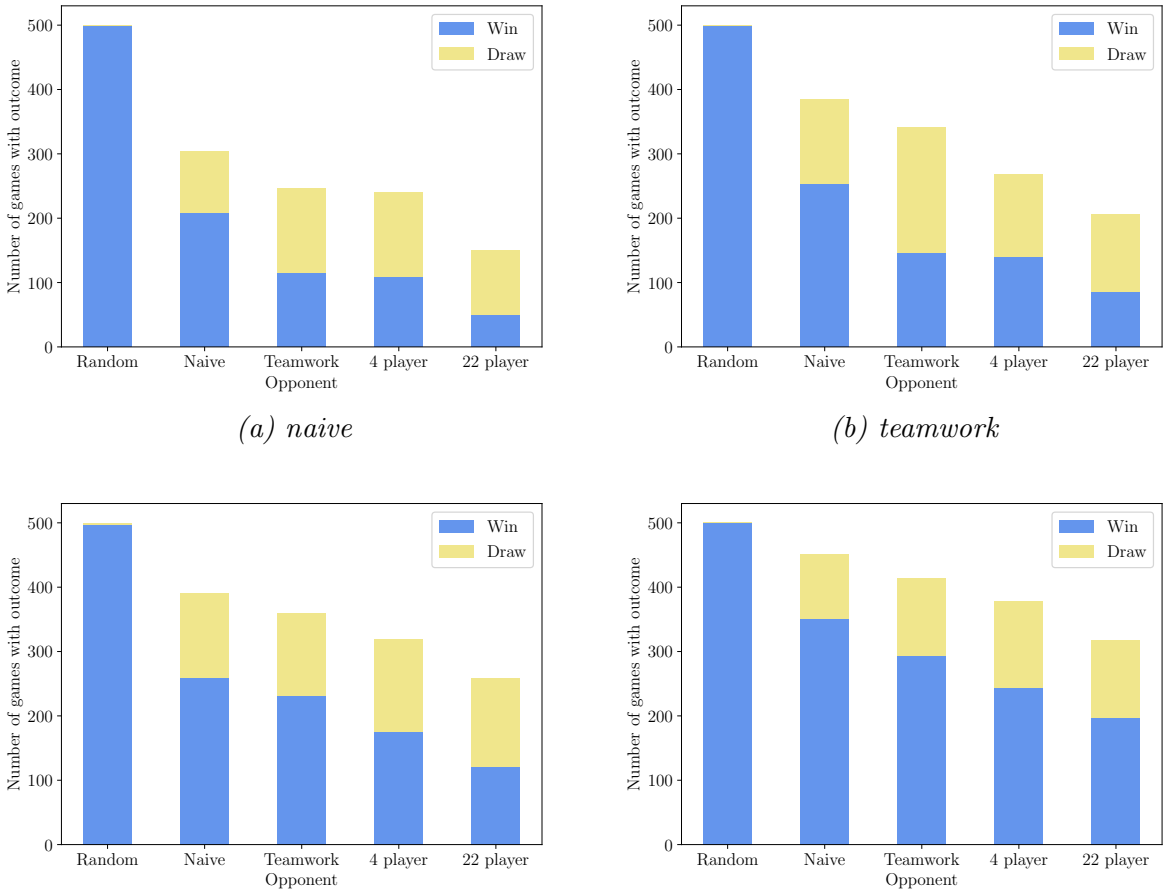
We now investigate the average game outcomes of our teams. We train both the 4 player and 22 player systems for 55 hours. For the team training in the 4 player setting, we spawn the agents at random positions at the start of each 4 player game. This allows them to better transfer to the full game as they have experience over many different positions. Average scores of the different teams after training are given in Table 6.3.

Table 6.3: The average game outcomes between pairs of teams, after 500 games played in the full 22 player setting. Each entry represents the average game outcome (0 for a loss, 0.5 for a draw, 1 for a win) that the row team achieves against the column team, plus-minus the standard deviation. We use the best performing seed for each team. The team trained in the 22 player setting outperforms all other teams.

Team	Random	Naive	Teamwork	4 player	22 player
Random	0.50±0.00	0.00±0.00	0.00±0.00	0.00±0.04	0.00±0.00
Naive	1.00±0.00	0.51±0.45	0.36±0.41	0.35±0.40	0.20±0.33
Teamwork	1.00±0.00	0.64±0.41	0.49±0.39	0.41±0.42	0.29±0.38
4 player	1.00±0.04	0.65±0.40	0.59±0.42	0.50±0.42	0.38±0.41
22 player	1.00±0.00	0.80±0.33	0.71±0.38	0.62±0.41	0.52±0.44

As can be seen in Table 6.3, training in the full 22 player setting seems to yield the best team strategy when using the same computational resources. One obvious reason why agents trained in the 22 player setting might outperform agents trained in the 4 player setting, can be that an agent trained in the 4 player setting has never encountered more than 3 additional agents. Therefore its attention policy might not be adapted to process more agents effectively. In an effort to rectify this for the team trained in the 4 player setting, during evaluation in the 22 player setting we experimented with selecting only three agents to pass to the policy. However, this made little difference and the team still performed worse than the team trained in the full 22 player setting.

We also present a breakdown of the total games won, drawn and lost by each team (except the random strategy) against different opponents. The results can be seen in Figure 6.7.



(a) naive *(b) teamwork*

(c) trained with 4 players *(d) trained with 22 players*

Figure 6.7: The total games won, drawn and lost by each team against different opponents over 500 games. Total games lost would be the difference between the top of each bar and the 500 game mark.

As can be seen, the teamwork strategy has a high draw rate against itself. This seems to indicate that it has a strong defensive strategy, but lacks offensive capabilities, which makes sense as the strikers naively chase the ball. Along with having the highest average score, the team trained in the 22 player setting also has the highest game win rate against every opponents, and fewer draws. This might indicate that the team is more focused on offensive play than defensive play. We will investigate whether this is the case later in the chapter.

6.5.2 Average field positions

We now investigate the average field positions of our two trained teams. Figures 6.8 and 6.9 show the distribution of positions on the field of each trained agent over a number of games in the 22 player setting, for the team trained in the 4 player setting and the team trained

in the 22 player setting, respectively. These positions are from games played against the handcrafted teamwork strategy.

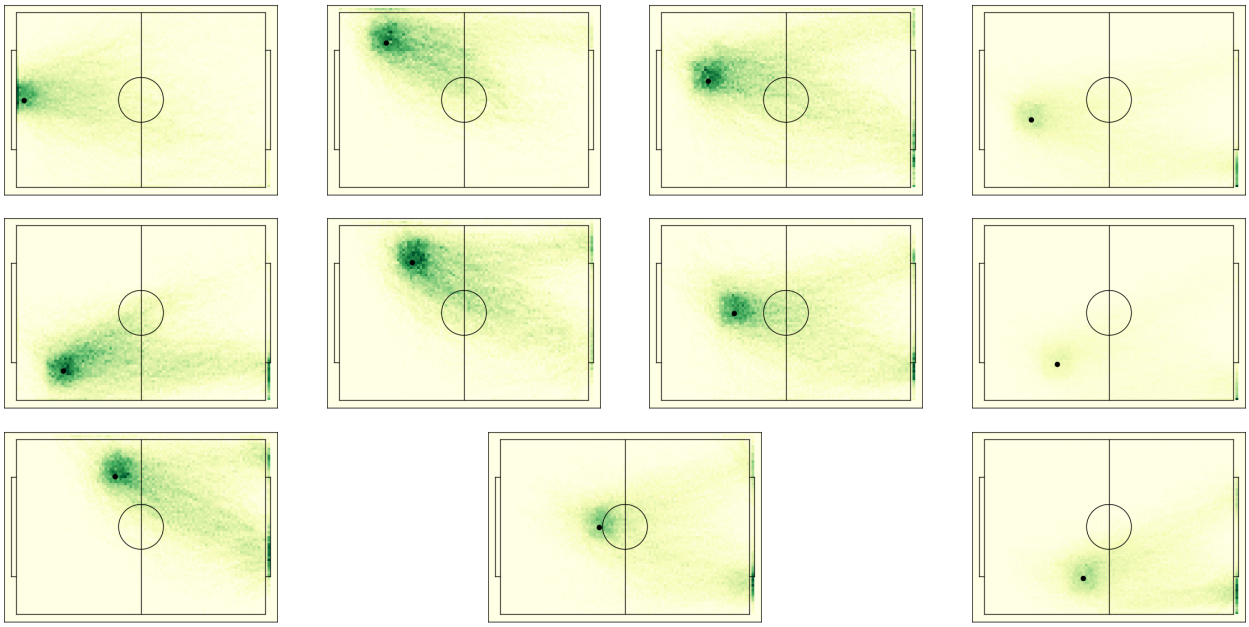


Figure 6.8: Average player positions of agents trained in the 4 player setting, through the course of a 22 player game. Each agent spawns in a specific position, as indicated by the black circle in each image. The green-yellow colour intensities represent the probability of finding a specific player at that position throughout the game. Dark green represents more probable while light yellow is less probable. These agents seem to move away from their starting positions quite quickly. The midfielders and attackers usually charge for the ball and try and score as quickly as possible, even if it means sacrificing defensive play.

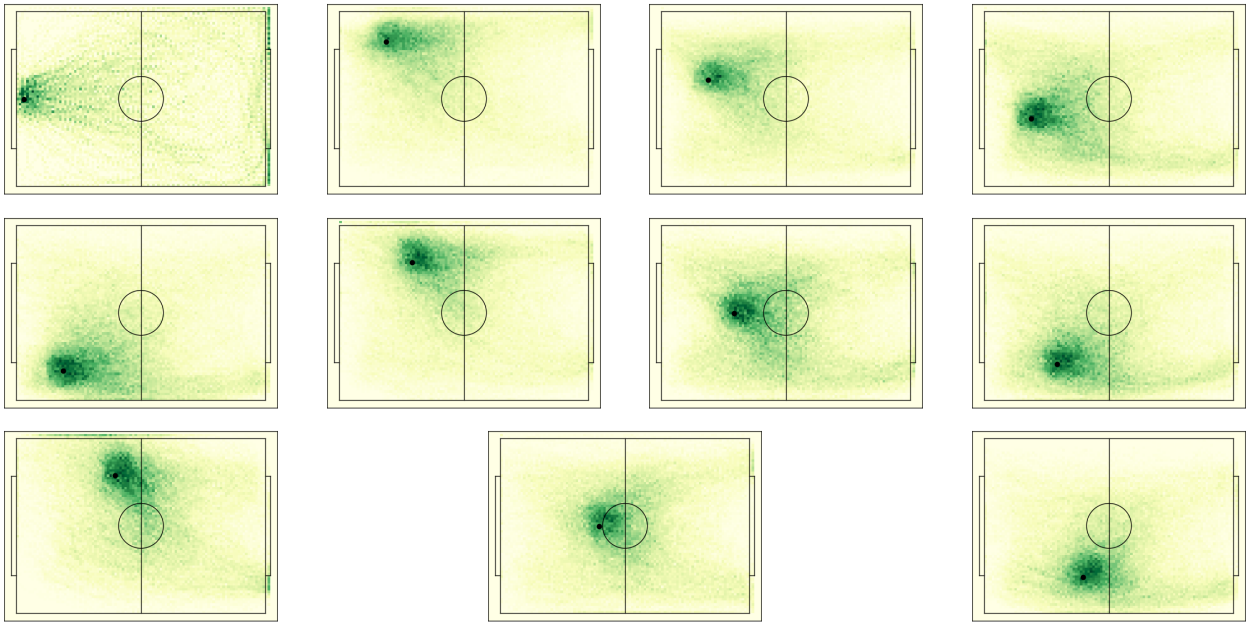


Figure 6.9: Average player positions of agents trained in the 22 player setting. These agents seem to stay closer to their starting positions for a larger proportion of the game, indicating that they may have learned to play more positional soccer.

As can be seen, agents trained in the 22 player setting seem to remain closer to their

starting positions, and spread out more, instead of simply charging towards the ball or the opponent's goal line. Human teams, to an extent, also remain relatively close to their starting positions. Therefore, this may help explain why the team trained in the 22 player setting performs better than the team trained in the 4 player setting.

6.5.3 Gameplay analysis

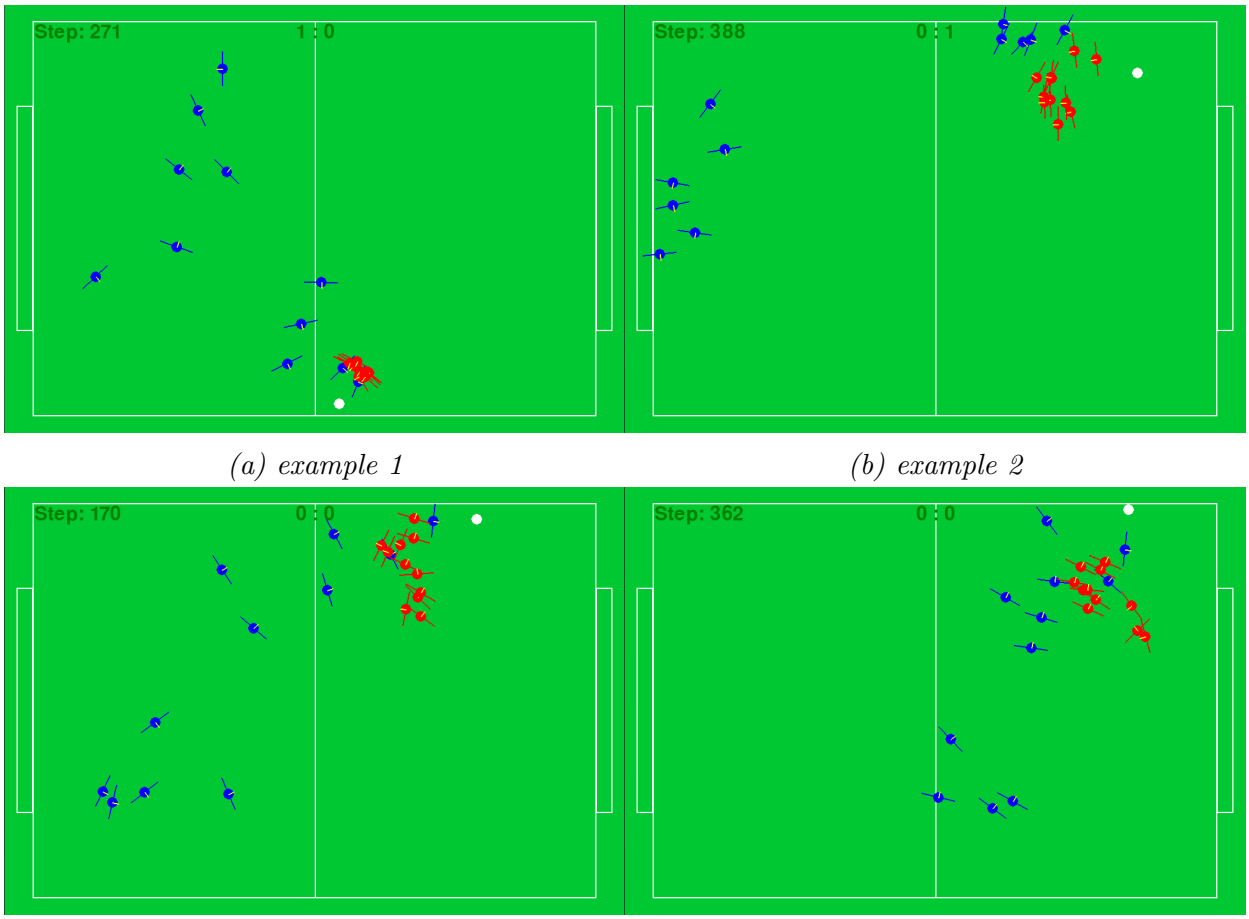
We now perform a more subjective analysis of gameplay footage of the team trained in the 22 player setting. A video detailing various teams playing games against each other, with episode lengths of 400, is available online[4]. The video shows that the team trained in the 22 player setting exhibits less ball chasing behaviour than the team trained in the 4 player setting. The agents from the 22 player setting seem to either chase the ball if they are close to it, or position themselves in locations where the ball might be going to. As a result, this team tends to be much more spread out on the field than the team trained in the 4 player setting. This is seen as a good learned behaviour to have, as real soccer strategies usually involve remaining spread out over the field to allow for better defense, when the defending team does not control the ball, and better offence, as players have more options to where to pass the ball.

As demonstrated in Figure 6.10, this learned positioning play sometimes leads to the team splitting into two groups on the field, akin to strikers and defenders. While the team's positional play is not yet close to what a real soccer team is capable of, it is promising to see that this behaviour can be learned through self-play. It further seems to support the idea that training in the full 22 player setting is important for such strategies to emerge.

6.5.4 Quantitative analysis

Next we consider a more quantitative approach to evaluate certain attributes of the various teams. We let each of the five teams (random, naive, teamwork, 4 player, and 22 player) play against the team trained in the 22 player setting and then track various metrics on their gameplay. We use the same opponent for all teams, as the metrics might change depending on who a team is playing against. The various metrics include the average move action output (in meters per step), the average rotational action output (in radians per step), the average distance of the team's players to the ball (in meters), the average distance of the team's players to each other (in meters), the average distance of the closest player to the ball (in meters), the average distance of the closest player to its own goal (in meters), and the average game outcome against the 22 player team (0 for a loss, 0.5 for a draw, and 1 for a win). These metrics are all measured over 500 games, and the results are presented in Table 6.4. In combination, the metrics provide some indication of a team's positioning throughout gameplay. The handcrafted teamwork players, for example, have good positioning based on

[4]Gameplay video: https://youtu.be/YrQVHeo9ZrA



(a) example 1 *(b) example 2*

(c) example 3 *(d) example 4*

Figure 6.10: Defensive gameplay of the team trained in the 22 player setting (blue). The team is playing against the top team trained in the 4 player setting (red). Notice how the blue team splits into defensive and offensive groups.

these metrics, yet their average game outcome (against the team trained in the 22 player setting) is relatively low. The average move metric is a bit deceiving for this team as the strikers do most of the ball chasing, and bunch up when doing so. The team trained in the 22 player setting on average has players relatively close to both the ball and their own goal. They are also generally more spread out on the field, and do not bunch up that often compared to the other teams, which might explain why they outperform all other teams.

6.6 Comparison to related work

The focus of our work is on applying end-to-end multi-agent reinforcement learning in the full 22 player game of soccer. We created a custom environment that executes more than 25× faster than the 2D RoboCup simulator, even after removing all wait instructions from the latter. MacAlpine and Stone [38] used multi-agent reinforcement learning to let their agents learn individual soccer skills in the 3D RoboCup environment. However, their setting was limited to only two agents. Liu et al. [35] also proposed a 2D environment with relatively

Table 6.4: Metrics of various teams playing against the team trained in the 22 player setting. For the first two rows, a lower value is generally better as it implies that the players do not need to correct their positions that often. For rows 3 and 4, a higher value is generally better as it implies that the players are not bunching up. For rows 5 and 6, a lower value is generally better as it implies that at least one player is close to the ball or their own goal, respectively. For the average game outcome, higher values are better.

Metric	Random	Naive	Teamwork	4 player	22 player
Move	0.50±0.29	0.97±0.02	0.55±0.50	0.91±0.13	0.54±0.25
Rotation	0.20±0.12	0.08±0.11	0.06±0.10	0.12±0.10	0.08±0.07
Ball distance	34.9±18.2	16.6±11.8	30.8±22.36	16.4±11.4	31.9±19.3
Team distance	28.5±12.5	8.39±10.4	30.7±20.8	8.62±9.19	26.9±14.6
Closest to ball	14.3±11.9	9.20±5.51	8.86±4.78	9.05±5.72	11.0±10.1
Closest to goal	5.48±2.72	35.6±14.7	10.3±1.70	31.3±14.6	19.9±12.8
Average outcome	0.00±0.00	0.20±0.33	0.29±0.38	0.38±0.41	0.52±0.44

fast simulation times. However, we cannot fairly compare our agents with theirs as they trained 32 independent learners over multiple GPUs. We limited ourselves to using only one GPU so that researchers with compute constraints can experiment with the implementation ideas presented in this work. Furthermore, Liu et al. [35] presented results only for agents trained in a 4 player setting, while we were able to achieve stable training in the full 22 player setting. Moreover, we relied on a partially observable environment to more closely align with the 2D RoboCup setup, while Liu et al. [35] provided full state information to their agents.

Our main contribution in this work is showing that stable training in the full 22 player setting, in an end-to-end fashion, is possible. Although our teams still have relatively weak strategies, compared to human soccer strategies, we do see benefit in training in the full 22 player setting. We show that training in this setting allows for higher-level strategies to emerge, such as players spreading out on the field, remaining closer to their starting positions, and forming defensive and offensive groups. More importantly, this behaviour is completely learned through self-play and therefore makes this method applicable to other similar environments.

6.7 Full system training in the 2D RoboCup environment

As a final experiment we consider whether it might be possible to execute our learning algorithm in the wrapped 2D RoboCup environment. As mentioned in Section 6.2.1, this environment is 25× slower than our own custom environment, and 14× slower when neural network policies are used, which significantly impacts training times. For our algorithm to

run in the RoboCup environment, we convert the hybrid action space to a simpler continuous action space (as described in Table 4.1). We use the same hyperparameters as those used for the Polyak averaged opponent experiment in Section 6.3. A RoboCup competition typically lasts around 6000 steps, which equates to 10 minutes of game time. For our training setup we train on only 400 steps per episode, to be consistent with our custom environment's training setup. After training the agents can play in games longer than 400 steps.

To enable learning in the brief initial reward shaping part of training, we remove the ball throw-ins, and instead allow the ball to bounce against the sides. This is needed because an opponent with a random policy would be unlikely to kick the ball back into play, and therefore stall the episode. The throw-ins are added back into the game after initial training when the reward shaping is removed.

For evaluation we implement a naive opponent team that has the same strategy as the naive team in the custom environment, i.e. chasing the ball and kicking it to the center of the opponent's goal. The result from training in the full 22 player RoboCup environment is presented in Figure 6.11.

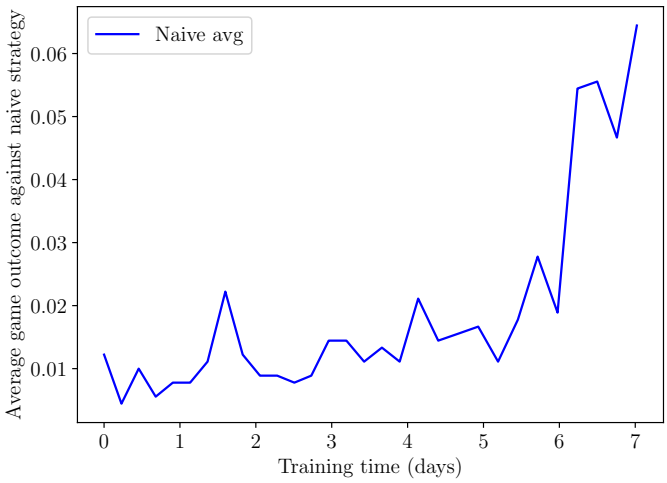


Figure 6.11: Performance of a team training in the 2D RoboCup environment, in the 22 player setting, as evaluated against an opponent with the naive strategy. A game outcome can be either a loss (0), draw (0.5) or win (1.0) for the team.

As can be seen, our algorithm takes significantly longer to train in the 2D RoboCup environment. However, after a few days of training the win rate against the naive opponent does increase. Besides the environment step times, we also found that the actions in this RoboCup environment need to be more precise than in our custom environment. If the actions are too erratic, the team cannot consistently score goals. Therefore, for a large part of training the team does not score many goals, but after the noise (standard deviation) in action outputs lowers the agents start scoring more often.

Importantly, we did not observe any policy collapse through the course of training, which

seems to indicate that the solutions proposed in this work also transfer to the RoboCup environment. However, because the environment is much slower than the custom environment, reaching competent policies takes much longer.

A video showing gameplay of these agents is also available online[5]. The naive team is still better than the team that is training, as the latter has not had enough time to learn to play more positional yet, and seems to also employ only a basic ball chasing strategy.

6.8 Summary

In this chapter we investigated the performance of various aspects of our system. We started by providing some experimental details, and then evaluated various performance improving methods for our code. We showed that we could significantly increase training iterations through compiling aspects of our environment to machine code, running multiple parallel workers, running a dedicated data server to transport data, and batching our agent policies.

Next we trained the full system in the self-play setting. After training we evaluated our agents against three handcrafted opponent teams. We found that after 31 hours of training, the agents started outperforming all these opponents. We then also trained our system in the 4 player setting for 55 hours. Our team trained in the 22 player setting could outperform these agents in just 45 hours of training. This seems to indicate that training in the full 22 player setting is important for the formation of higher-level team strategies.

We also investigated how players trained in both the 4 and 22 player settings move around the field in a game. We saw that the agents trained in the 22 player setting stayed closer to their starting positions, than the agents trained in the 4 player setting. This seems to indicate that the team trained in the 22 player setting has learned to play more positional soccer. Furthermore, we observed that the team trained in the 22 player setting seemed to occasionally form two groups, where one is defensive and the other offensive. This seems to give them a strategic advantage over the other teams, and is also a basic strategy used by real soccer teams.

Lastly, we demonstrated that it is possible to execute our learning algorithm in the wrapped RoboCup environment. To get our agents learning we converted the hybrid action space to a simpler continuous action space. We also removed time consuming events, such as throw-ins, from the initial reward shaping phase of training. While training is slow, the learning team does get better over time which suggests that our algorithm can be transferred to the 2D RoboCup simulation environment.

[5]Gameplay video: https://youtu.be/OYL5fo6yiHY

Chapter 7

Conclusion

In this dissertation we investigated and developed techniques needed to scale end-to-end multi-agent reinforcement learning to a full 22 player soccer simulation. We believe this investigation to be important for the emergence of general learning algorithms that can perform well also in domains other than RoboCup, and therefore aligned with the motivation behind RoboCup as a test bed for algorithms that can one day be used to solve real-world problems. We further limited the computational resources that would be available to train our agents, so that researchers with resource constraints can utilise and build on our work. We also focused on generating soccer strategies entirely through self-play, instead of using some other team strategy to train against or perform supervised learning on. This self-play approach can be useful in domains where good handcrafted or learned policies are difficult to come by.

7.1 Summary of our approach

For our multi-agent environment we needed a learning algorithm that would be fairly stable throughout training in terms of performance increase over time. We decided on the PPO algorithm, due to its on-policy nature and its policy clipping update rules which can prevent strategy collapse. Furthermore, we needed the policy to work with inputs of variable size, according to the number of other players an agent observes, and have an ability to remember information from previous time steps. To achieve this we decided to use an attention mechanism combined with an LSTM network for the policy output. To combat the inherit instability of training in the self-play setting, we considered a slow updating (Polyak averaged) opponent as well as league training. These opponents both represent combinations of previously learned strategies, which can help to stabilise training.

A significant problem with training in the 22 player setting is the amount of computational resources required to achieve good results. To help address this problem we implemented a number of code acceleration improvements. We assigned every agent in a team the same parameters. This helped reduce the number of trainable parameters while also

allowing for batching, where the actions can be calculated for each agent in parallel using a set of matrix multiplications. We used the Mava framework which allows for multiple workers to generate episodes in parallel. We also let the learner run as a separate GPU process which provided more CPU time for the workers. We used TensorFlow's code accelerator function called tf.function, which is a transformation tool that creates Python-independent dataflow graphs for fast execution of the policies and training loops.

For this work we decided to build our own lightweight simulation environment. We built it using a Python library called JAX, which compiles linear algebra Python code to machine code that executes much faster. By using JAX in this manner, our environment runs more than 7× faster than the Python equivalent code.

7.2 Main findings

By adapting the standard PPO algorithm with our proposed improvements we demonstrated that it is possible to train multiple agents directly in the full 22 player soccer setting, through multi-agent reinforcement learning and self-play,, using a limited computational budget. We trained a team of agents and evaluated them against fixed strategies. Our trained agents achieved a higher average score than all other strategies considered, after 45 hours of training in the Polyak averaged opponent setup, and after 53 hours in the league training setup. Furthermore, we performed an ablation study for the Polyak averaged opponent setup and showed that (1) the opponent stabilisation and (2) the use of attention mechanisms assist training in the 22 player setting. We then also showed the benefits of training in the 22 player setting over the 4 player setting. With the same computational budget, the agents trained in the 22 player setting perform better than agents trained in the 4 player setting, when evaluated in the full game. This seems to indicate that it is important to train agents in the full 22 player setting.

We further evaluated the trained agents in various ways. We showed that agents trained in the 22 player setting seem to stay closer to their starting positions than those trained in the 4 player setting, which might indicate better demarcation of individual responsibilities when training is done in the full game. We also investigated some learned behaviours acquired by the team training in the 22 player setting. When agents are further away from the ball they tend to move slower, which naturally results in a defensive group and offensive group of agents. The defensive group often stops the opponents from scoring goals when they have broken past the offensive group. We also investigated quantitative attributes of all our teams, in an attempt to evaluate the positional play of the teams. We found that the handcrafted teamwork team as well as the team trained in the 22 player setting performed well across all metrics, except average game outcome where the trained team excelled, indicating that these teams have some positional play over just ball chasing.

We also showed that our algorithm can directly learn in the full 22 player RoboCup

environment. We evaluated our learning team's performance over time against a naive strategy and could see a gradual improvement over seven days of training.

7.3 Revisiting our objectives

In Chapter 1, we introduced three main objectives. We revisit each of these objectives below.

7.3.1 System design

Our first objective was to design a multi-agent system that can exhibit stable training in the full 22 player setting through self-play. To this end, we proposed various improvements to the PPO algorithm that would help it scale to the 22 player setting. We used an attention mechanism that can process a varying number of players, and scales linearly with the number of players, as appose to the mechanism used by Liu et al. [36], which scales quadratically. The attention mechanism, along with the shared network setup, also reduces the number of parameters that need to be learned. We also implemented various code acceleration improvements such as batched policy forward passes, compiling Python code using TensorFlow's tf.function method, and using parallel running workers and learner. Furthermore, to stabilise training we implemented two types of opponents, namely Polyak averaged opponents and a league of opponents. We showed through various experiments that these proposed improvements do enable stable training in the full 22 player setting.

7.3.2 Soccer environment

A common challenge in multi-agent settings is that training is extremely slow. Our second objective was to design a 2D soccer simulation environment, faster than RoboCup's, that retains the main challenges for multi-agent reinforcement learning. To achieve this objective, we created an environment where players could interact only with the ball and not also each other. Furthermore, the ball bounces from the side walls so no throw-ins are needed. The environment uses matrix multiplication to calculate all player position updates and observation information in parallel. This drastically improves the throughput of the environment. We also compile the environment using JAX's JIT functionality. The end result is an environment that is 25× faster than the 2D RoboCup environment.

7.3.3 Evaluation

Our third objective was to experimentally verify that our solution can learn competent strategies when evaluated against handcrafted opponents. We could see that our system using both the Polyak averaging and league training setups could learn without policy collapse.

We found that our team, trained in the 22 player setting, outperformed all handcrafted opponents and also a team trained in the 4 player setting. This confirmed our belief that training in the 22 player setting can allow for higher-level strategies to form. We further evaluated the behaviour of the team trained in the 22 player setting and found that it exhibits more positional play than the team trained in the 4 player setting. The 22 player trained team tends to spread out more and sometimes forms two groups, one defensive and one offensive. While their strategies are not yet close to that of human teams, it is promising to see these positional play strategies emerge from self-play alone.

7.4 Future work

It seems our agents trained in the custom environment can still improve in terms of team strategy when visually comparing their gameplay to known soccer strategies, in terms of better positional play and more directed ball passing. While training our reinforcement learning team for longer does yield better strategies, the rate of increase in performance diminishes. Below we present possible future research direction which could potentially improve the team's performance against the best handcrafted algorithms.

7.4.1 Scaling computation

In this work, we limited ourselves to using only one CPU and one GPU. It might be interesting to experiment with more computational resources. One way of doing this is by adding more workers running on multiple CPUs and thereby generating more experience. Furthermore, a GPU with more memory can be used to increase the batch size and to process the increased experience throughput. This may possibly allow for much stronger strategies to emerge through self-play.

7.4.2 Communication

A limitation of our current approach might be that the agents need to better assign individual responsibilities within the context of the team, which might also require a communication mechanism to share information dynamically. In future work one could investigate giving the agents communication abilities, which could help them to coordinate better.

7.4.3 JIT compilation

Another improvement to this work can be to use a JIT compiled JAX function for both the worker and learner step functions. We decided to use TensorFlow, as Mava uses TensorFlow and JAX was not yet that well developed at the time. However, in the last year of this study Mava has begun switching to JAX. Initial experiments in Mava seem to indicated that JAX's JIT functionality is much more efficient than TensorFlow's tf.function.

7.4.4 Training in different sized settings

A common problem with training directly in the 22 player setting is that a larger number of episodes need to be generated in order to achieve competent strategies, compared to settings with fewer players. Training is inherently more noisy, as the rewards are generally weakly coupled to the actions of a particular agent. To combat this, we reduced both the computational resources required in the full setting as well as the number of trainable parameters. A research direction could be to investigate training in more than one setting, with different numbers of players, at the same time. One way of doing this (that we attempted) is to first train in the 1 vs 1 player setting until the agents outperform some benchmark, and then incrementally move upwards until the full 22 player setting is reached. This however requires a fixed benchmark to be specified (e.g. a handcrafted opponent) and training might become unstable when the number of players change abruptly. A better approach might be to simultaneously train in settings with different numbers of players. In this way, basic skills can be learned in settings with fewer agents and high-level team strategies in settings with more agents. However, to achieve this training setup, special care needs to be given to the data. The learner would need to sample batches of experience from various settings. Even though attention is used, when operations are batched in the learner they need to be of the same size, i.e. the same number of players. Therefore all experience from settings with fewer than 22 players needs to be zero padded until the observation and state sizes match the 22 player setting. Then, after training, the agents can play in any setting as before.

7.4.5 Using domain randomisation

Our work was inspired in part by the work of Paino et al. [46], who trained a virtual robotic hand to solve a Rubik's Cube in simulation. By using domain randomisation they could guide the reinforcement learning agent to solve cubes under different physical setups and permutations of the environment. If the environment varies enough, the agent can learn to adapt to new environments at inference time. When Paino et al. [46] transferred the learned policy to the real world, the agent was able to solve a maximally scrambled real Rubik's Cube with a 20% success rate; impressive considering the policy never trained in the real world.

To apply domain randomisation to RoboCup, one first needs to address the slow execution times of the RoboCup simulation environments when compared to our custom environment. An idea for future work can therefore be to train teams in many simplified simulations, such that the agents can learn to adapt to new environments, then only deploy them in the RoboCup simulations. Furthermore, training robots to play soccer in the real world is expensive and does not scale well in terms of training time. Hopefully, once good performance is achieved in simple 2D and 3D simulation environments, one could look at applying domain randomisation so that the agents can adapt to the real world. A starting point can be to build a simulation environment that mimics one of the simplest real

RoboCup environments, such as the small size league shown in Figure 7.1. The objective then becomes to get an agent to reliably locate the ball and score goals on a real soccer field, while training only in a simulation environment.

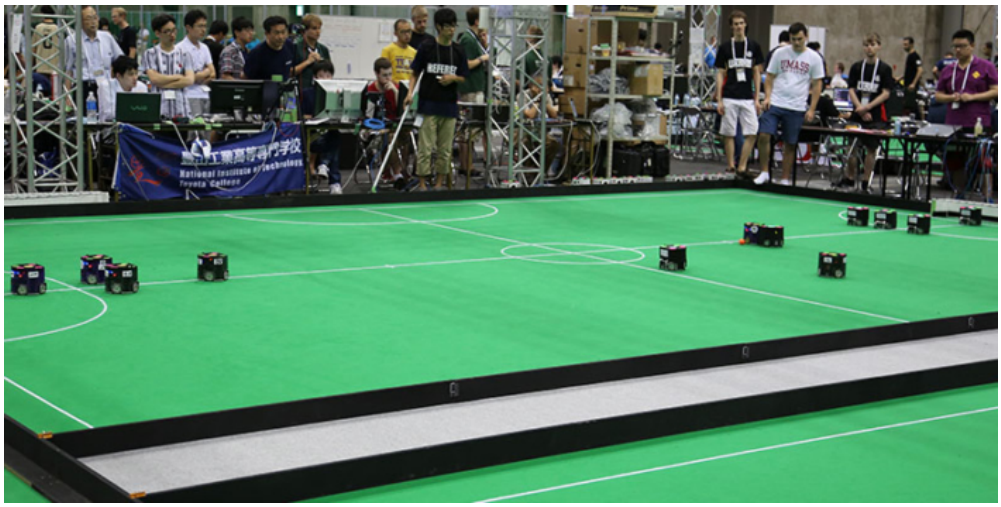


Figure 7.1: Robots in the small size RoboCup soccer league playing a game. Source: RoboCup [51].

Of course, soccer is merely a test bed for these methods, which could hopefully play a small part in advancing robotics to benefit the real world.

Bibliography

[1] Miguel Abreu, Luis P. Reis, and Nuno Lau. Learning to run faster in a humanoid robot soccer environment through reinforcement learning. *Artificial Intelligence and Lecture Notes in Bioinformatics*, pages 3–15, 2019.

[2] Josh Achiam. Extra material, 2018. URL https://spinningup.openai.com/en/latest/spinningup/extra_pg_proof1.html.

[3] Hidehisa Akiyama. RoboCup2019 soccer simulation 2d final, 2020. URL https://www.youtube.com/watch?v=BVMat_hAxss.

[4] Hidehisa Akiyama, Tomoharu Nakashima, Sho Tanaka, and Takuya Fukushima. HELIOS2017: Team description paper. *RoboCup 2017: Robot World Cup XXI*, 2017.

[5] Hidehisa Akiyama, Tomoharu Nakashima, Takuya Fukushima, Yudai Suzuki, and An Ohori. HELIOS2019: Team description paper. *RoboCup 2019: Robot World Cup XXIII*, 2019.

[6] Saad Albawi, Tareq A. Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *International Conference on Engineering and Technology*, 2017.

[7] Simon Alvarez. Elon Musk's OpenAI bots crush veteran Dota 2 players ahead of international tournament, 2018. URL https://www.teslarati.com/openai-bots-defeat-pro-dota-2-players/.

[8] Sashwat Anagolum. Quantum machine learning: Learning on neural networks, 2020. URL https://towardsdatascience.com/quantum-machine-learning-learning-on-neural-networks-fdc03681aed3.

[9] Craig Atkinson, Brendan McCane, Lech Szymanski, and Anthony Robins. Pseudo-rehearsal: Achieving deep reinforcement learning without catastrophic forgetting. *Neurocomputing*, pages 291–307, 2021.

[10] Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, and Timothy P. Lillicrap. Distributed distributional deterministic policy gradients. *ArXiv Preprint*, 2018.

[11] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *ArXiv Preprint*, 2019.

[12] George Brown. Some notes on computation of games solutions. *RAND Corporation Report*, 1951.

[13] Lucian Busoniu, Robert Babuska, and Bart De Schutter. *Multi-agent Reinforcement Learning: An Overview*, pages 183–221. Springer, 2010.

[14] Murray Campbell, A. Joseph Hoane, and Feng-Hsiung Hsu. Deep Blue. *Artificial Intelligence*, pages 57–83, 2002.

[15] Albin Cassirer, Gabriel Barth-Maron, Eugene Brevdo, Sabela Ramos, Toby Boyd, Thibault Sottiaux, and Manuel Kroiss. Reverb: A framework for experience replay. *ArXiv Preprint*, 2021.

[16] Sneha Chaudhari, Varun Mithal, Gungor Polatkan, and Rohan Ramanath. An attentive survey of attention models. *ACM Transactions on Intelligent Systems and Technology*, 2021.

[17] Gang Chen. A new framework for multi-agent reinforcement learning - centralized training and exploration with decentralized execution via policy distillation. *ArXiv Preprint*, 2019.

[18] Mert Cokluk. Kullback-Leibler divergence loss vs (weighted) cross entropy loss, 2019. URL https://medium.com/@mertcokluktttt/kullback-leibler-divergence-loss-vs-weighted-cross-entropy-loss-79126dccc8a2.

[19] Elhadji Amadou Oury Diallo, Ayumi Sugiyama, and Toshiharu Sugawara. Coordinated behavior by deep reinforcement learning in doubles pong game. *Joint Agent Workshops and Symposium*, 2017.

[20] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. *ArXiv Preprint*, 2016.

[21] Kunihiko Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, pages 121–136, 1975.

[22] Geeky.news. DeepMind AI taught virtual characters to play football from scratch, 2021. URL https://geeky.news/deepmind-ai-taught-virtual-characters-to-play-football-from-scratch/.

[23] Dibya Ghosh. KL divergence for machine learning, 2018. URL https://dibyaghosh.com/blog/probability/kldivergence.html.

[24] Gregory Gundersen. Entropy of the Gaussian, 2020. URL http://gregorygundersen.com/blog/2020/09/01/gaussian-entropy/#:~:text=Entropy%20of%20the%20multivariate%20Gaussian.&text=It%20relies%20on%20several%20properties,(%20I%20D%20)%20%3D%20D%20.

[25] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *ArXiv Preprint*, 2018.

[26] Nessrine Hammami and Kim-Khoa Nguyen. On-policy vs. off-policy deep reinforcement learning for resource allocation in open radio access network. In *IEEE Wireless Communications and Networking Conference*, 2022.

[27] Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. In *International Conference on Machine Learning*, 2015.

[28] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, pages 1735–80, 1997.

[29] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alexander Novikov, Sergio G. Colmenarejo, Serkan Cabi, Çaglar Gülçehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. *ArXiv Preprint*, 2020.

[30] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *ArXiv Preprint*, 2017.

[31] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ArXiv Preprint*, 2014.

[32] C. L. Lawson. Transforming triangulations. *Discrete Mathematics*, pages 365–372, 1972.

[33] David S. Leslie and E.J. Collins. Generalised weakened fictitious play. *Games and Economic Behavior*, pages 285–298, 2006.

[34] Sergey Levine. Reinforcement learning and control as probabilistic inference: Tutorial and review. *ArXiv Preprint*, 2018.

[35] Siqi Liu, Guy Lever, Josh Merel, Saran Tunyasuvunakool, Nicolas Heess, and Thore Graepel. Emergent coordination through competition. *International Conference on Learning Representations*, 2019.

[36] Siqi Liu, Guy Lever, Zhe Wang, Josh Merel, S. M. Ali Eslami, Daniel Hennes, Wojciech M. Czarnecki, Yuval Tassa, Shayegan Omidshafiei, Abbas Abdolmaleki, Noah Y. Siegel, Leonard Hasenclever, Luke Marris, Saran Tunyasuvunakool, H. Francis Song, Markus Wulfmeier, Paul Muller, Tuomas Haarnoja, Brendan D. Tracey, Karl Tuyls, Thore Graepel, and Nicolas Heess. From motor control to team play in simulated humanoid football. *ArXiv Preprint*, 2021.

[37] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *ArXiv Preprint*, 2017.

[38] Patrick MacAlpine and Peter Stone. Overlapping layered learning. *Artificial Intelligence*, pages 21–43, 2018.

[39] Daniel McNeela. The REINFORCE algorithm aka Monte-Carlo policy differentiation, 2022. URL https://mcneela.github.io/math/2018/04/18/A-Tutorial-on-the-REINFORCE-Algorithm.html.

[40] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *Proceedings of Workshop at ICLR*, 2013.

[41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *ArXiv Preprint*, 2013.

[42] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, pages 1476–4687, 2019.

[43] Thanh Nguyen, Ngoc D. Nguyen, and Saeid Nahavandi. Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications. *IEEE Transactions on Cybernetics*, pages 1–14, 2020.

[44] Drew Noakes. RoboCup, 2022. URL https://en.wikipedia.org/wiki/RoboCup.

[45] Christopher Olah. Understanding LSTM networks, 2015. URL https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[46] Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, and Nikolas Tezak. Solving Rubik's Cube with a robot hand. *ArXiv Preprint*, 2019.

[47] Arnu Pretorius, Kale-ab Tessera, Andries P. Smit, Claude Formanek, St John Grimbly, Kevin Eloff, Siphelele Danisa, Lawrence Francis, Jonathan Shock, Herman Kamper, Willie Brink, Herman Engelbrecht, Alexandre Laterre, and Karim Beguir. Mava: A research framework for distributed multi-agent reinforcement learning. *ArXiv Preprint*, 2021.

[48] Mikhail Prokopenko and Peter Wang. Guiding self-organisation of intelligent agents (Fractals2019). *RoboCup 2019: Robot World Cup XXIII*, 2019.

[49] Tabish Rashid, Mikayel Samvelyan, Christian S. de Witt, Gregory Farquhar, Jakob N. Foerster, and Shimon Whiteson. QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning. *ArXiv Preprint*, 2018.

[50] Roie Reshef. Multi agent reinforcement learning in Flatland domain, 2019. URL https://crml.eelabs.technion.ac.il/projects/multi-agent-reinforcement-learning-in-flatland-domain/.

[51] RoboCup. RoboCup leagues, 2020. URL https://2020.robocup.org/en/the-leagues/.

[52] RoboCup. 3d tools and support, 2020. URL https://ssim.robocup.org/3d-simulation/3d-tools/.

[53] RoboCup. Objective: Pushing the state-of-the-art, 2020. URL https://www.robocup.org/objective.

[54] RoboCup. A brief history of RoboCup, 2022. URL https://www.robocup.org/a_brief_history_of_robocup.

[55] Lukas Schafer. Blog - multi-agent learning environments, 2021. URL https://agents.inf.ed.ac.uk/blog/multiagent-learning-environments/.

[56] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations*, 2016.

[57] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv Preprint*, 2017.

[58] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, pages 484–489, 2016.

[59] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, pages 354–359, 2017.

[60] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, Shogi, and Go through self-play. *Science*, pages 1140–1144, 2018.

[61] Peter Stone and Manuela Veloso. Layered learning. In *European Conference on Machine Learning*, 2000.

[62] Chenyu Sun, Hangwei Qian, and Chunyan Miao. From psychological curiosity to artificial curiosity: Curiosity-driven learning in artificial intelligence tasks. *ArXiv Preprint*, 2022.

[63] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech M. Czarnecki, Vinícius F. Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. Value-decomposition networks for cooperative multi-agent learning. *ArXiv Preprint*, 2017.

[64] Synced. After mastering Go and StarCraft, DeepMind takes on soccer, 2020. URL https://medium.com/syncedreview/having-notched-impressive-victories-over-human-professionals-in-go-atari-games-and-most-recently-30b88ee363e9.

[65] Yasuto Tamura. Multi-head attention mechanism: "queries", "keys", and "values," over and over again, 2021. URL https://data-science-blog.com/blog/2021/04/07/multi-head-attention-mechanism/.

[66] Justin K. Terry, Benjamin Black, Ananth Hari, Luis S. Santos, Clemens Dieffendahl, Niall L. Williams, Yashas Lokesh, Caroline Horsch, and Praveen Ravi. PettingZoo: Gym for multi-agent reinforcement learning. *ArXiv Preprint*, 2020.

[67] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *International Conference on Intelligent Robots and Systems*, 2012.

[68] Shubhendu Trivedi and Risi Kondor. Papers with code - Polyak averaging explained, 2018. URL https://paperswithcode.com/method/polyak-averaging.

[69] Elena Vasiou, Konstantin Shkurko, Ian Mallett, Erik Brunvand, and Cem Yuksel. A detailed study of ray tracing performance: Render time and energy cost. *The Visual Computer*, 2018.

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[71] Xin Wang, Yudong Chen, and Wenwu Zhu. A comprehensive survey on curriculum learning. *ArXiv Preprint*, 2020.

[72] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, pages 229–256, 1992.

[73] Fan Yang, Gabriel Barth-Maron, Piotr Stanczyk, Matthew W. Hoffman, Siqi Liu, Manuel Kroiss, Aedan Pope, and Alban Rrustemi. Launchpad: A programming model for distributed machine learning research. *ArXiv Preprint*, 2021.

[74] Denis Yarats, Amy Zhang, Ilya Kostrikov, Brandon Amos, Joelle Pineau, and Rob Fergus. Improving sample efficiency in model-free reinforcement learning from images. *ArXiv Preprint*, 2019.

[75] Alastair Young. Worked examples 4: 1-1 multivariate transformations, 2011. URL https://www.ma.imperial.ac.uk/~ayoung/m2s1/Multivariatetransformations.PDF.

[76] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre M. Bayen, and Yi Wu. The surprising effectiveness of MAPPO in cooperative, multi-agent games. *ArXiv Preprint*, 2021.

[77] Yang Yu. Towards sample efficient reinforcement learning. In *International Joint Conference on Artificial Intelligence*, 2018.

[78] Nader Zare, Aref Sayareh, Mahtab Sarvmaili, Omid Amini, and Amilcar Soares. CYRUS soccer simulation 2D team description paper 2021. *RoboCup 2021: Robot World Cup XXV*, 2021.

[79] Zhilu Zhang and Mert R. Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. *ArXiv Preprint*, 2018.