Learning Decentralized Policies with Incremental Reinforcement Learning, Reward Shaping and Self-Play Learning

Jérémie Ngumba Bakambana

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Electrical and Eletronic) in the Faculty of Engineering at Stellenbosch University.

Supervisor: Prof. H. A. Engelbrecht

March 2023

Acknowledgements

For the completion of this modest work, I would like to express my deep gratitude to the following people:

- To my Supervisor Prof. H.A. Engelbrecht for the guidance and assistance throughout the whole project. For the time investigated in weekly meetings to discuss the evolution of the project. And for the opportunity to collaborate in this research.
- To my family who has always been supportive, and motivation brought through the whole time spent far from each other.
- To friends who have been encouraging, and helpful to distress throughout hectic times during the research.
- To anyone who has contributed directly or indirectly to the completion of this project.

Plagiaatverklaring / *Plagiarism Declaration*

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 the use of ideas, material and other intellectual property of another's work and to present is 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 vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
   I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Voorletters en van / *Initials and surname* **J.N. Bakambana**	Datum / *Date* **March 6, 2023**

Abstract

English

Humans have the interesting ability to adapt to complex tasks by leveraging knowledge acquired on simpler tasks. In addition, humans can coordinate behaviors to reach a common objective. The recent progress in the field of Reinforcement Learning (RL) has demonstrated that an agent can acclimate to a complex task after being introduced to a simpler variant of the same task.

In this study, we investigate the ability of RL agents to solve a complex task, while collaborating with another learning agent. The given task is a cooperative volleyball game in 3 dimensions. We used the Proximal Policy Optimization (PPO) algorithm as the training agent because it was successful in solving, in a single-agent scenario, a simple variant of the game, which is the same volleyball game in 2 dimensions. We applied Incremental RL as a training paradigm to address the sparsity due to the large state space of the experimental environment. We first started by investigating the problem in a single-agent scenario. We broke down the main task MDP into a sequence of incremental MDPs, which generated a sequence of different variants of the same task ranging from the simplest to the most complex. Then we trained the agent to solve each task in the sequence starting with the simplest. The investigation demonstrated that: (1) the agent can adapt to an incremental sequence of MDPs; (2) Reaching the optimal level of expertise in a simple variant of a task is not a requirement to adapt to a more complex variant of the same task, the agent can still adapt in a complex task after a partial mastering of a simpler variant; (3) the optimal policy generated by the agent at the final task generalizes over all previous MDPs generated by simpler variants of the final task; (4) A successful incremental learning can be influenced by two parameters: one controlling when the training agent can transit to a more complex variant of the given task, and another controlling how complex the new variant of the task must be.

Based on the experiment result in the single-agent scenario, we investigated the paradigm in cooperative multi-agent scenarios. Toward the investigation, we demonstrated that with appropriate Reward Shaping, decentralized learning can be effective to solve cooperative scenarios without necessarily tuning hyperparameters. We also showed that Incremental Learning is an effective and promising approach to address issues such as the sparsity of

tasks with large state space in the multi-agent scenario. We finally proved in our work the ability of RL agents to adapt to a dynamic environment and maintain collaboration with other agents.

Afrikaans

Mense het die interessante vermoë om by komplekse take aan te pas deur kennis wat op eenvoudiger take opgedoen is, te benut. Daarbenewens kan mense gedrag koördineer om 'n gemeenskaplike doelwit te bereik. Onlangse vooruitgang in versterkingsleer (VL) het bewys dat agente by komplekse taak kan aanpas deur met 'n eenvoudige taak te begin en die kompleksiteit van die taak geleidelik te verhoog.

In hierdie studie ondersoek ons die vermoë van versterkingsleeragente om by 'n komplekse taak aan te pas, terwyl hulle saamwerk met 'n ander agent. Die gegewe taak is 'n vlugbalbalspel in 3 dimensies (3D) waar 'n span moet saamwerk. Ons het die Proksimale Beleidsoptimalisering (PPO) algoritme gebruik om die taak op te los omdat die toestands- en aksieruimte van die taak kontinu is. Ons het ook PPO gebruik omdat dit suksesvol was in die basislynstaak, wat dieselfde vlugbalspel is wat in 'n 2-dimensionele omgewing gespeel word. Ons pas Inkrementele Versterkingsleer toe om die ylheid aan te spreek wat 'n gevolg is van die groot toestandsruimte van die eksperimentele omgewing. Ons het begin deur die probleem as 'n enkelagent-scenario te ondersoek. Ons het die hoofomgewing se Markov Besluitnemingsproses (MBP) opgebreek in 'n reeks inkrementele MBP's en die agent sekwensieël opgelei om elke MBP in die reeks op te los. Die studie het getoon dat: (1) die agent kan aanpas by 'n inkrementele reeks van MBP's; (2) om 'n subtaak perfek te bemeester is nie 'n vereiste vir 'n suksesvolle aanpassing by die taak van 'n volgende inkrement nie, die agent kan aanpas by die volgende inkrement na 'n gedeeltelike bemeestering van die huidige taak; (3) die optimale beleid wat deur die agent by die finale taak genereer word, veralgemeen oor alle vorige MBP's in die reeks.

Ons het ons studie voortgesit deur Inkrementele Leer in 'n koöperatiewe-kompeterende multi-agent scenario te ondersoek. Ons het getoon dat desentraliseerde leer met behoorlike Beloningsvorming die samewerkende scenario's suksesvol kan oplos sonder om spesifieke hiperparameter-instelling te vereis. Ons het ook gewys dat Inkrementele Leer 'n effektiewe en belowende benadering is om kwessies soos die ylheid van take met groot toestandsruimtes in die multi-agent scenario aan te spreek. Ons demonstreer met hierdie studie die vermoë van VL-agente om by 'n dinamiese omgewing aan te pas en om samewerking met ander agente te handhaaf.

Contents

List of Figures

List of Tables

Nomenclature

Variables and functions

$\mathcal{A}$	Action space
a	Action
α	Learning rate
b	bias of a neural network layer
δ	Performance threshold
η	incremental step
$\mathbb{E}$	Expected value
ϕ	Artificial neurone activation function
G_t	Return at time t
γ	Discount factor
h	Artificial neuron output
J	Objective function
$\mathcal{L}$	Huber loss
$\mathcal{L}_{CE}$	Cross-Entropy loss
μ	Deterministic policy
$\mathcal{P}$	State transition probability
∇	Gradient operator
$p(x)$	Probability density function with respect to variable x.
∂	Partial derivative operator
π	Stochastic policy
π^*	Optimal
q	State-action value
q_*	optimal state-action value
Q	Estimate of the state-action value
r	Reward
r'	Shaped reward

$\mathcal{R}$	Reward Function
R^2	R-squared loss
R^2_{adj}	Agjusted R-squared loss
S	State space
s	State
$\mathcal{T}$	Reinforcement Learning Task
θ	Neural network parameter matrix
v	State value
v^*	Optimalstate value
V	Estimate of the state value
w	vector weight of a neural network layer

Acronyms and abbreviations

2D	2 Dimensions
3D	3 Dimensions
Adagrad	Adaptive Gradient Algorithm
Adam	Adaptive moment estimation
ANN	Artificial Neural Networks
CNN	Convolutional Neural Networks
CTDE	Centralized Training and Decentralized Execution
DDPG	Deep Deterministic Policy Gradient
D-DQN	Double Q-network
DISTRAL	DIStill & TRAnsfer Learning
DQN	Deep Q-Network
ELU	Exponential Linear Unit
IMPALA	Importance Weighted Actor-Learner Architecture
IQL	Independent Q-learning
KL	Kullback–Leibler
MADDPG	Mutli-Agent Deep Detrministic Policy Gradient
MADQN	Multi-Agent Deep Q-network
MAE	Mean Absolute Error
MAPPO	Multi-Agent Proximal Policy Optimization
MARL	Multi-Agent Reiforcement Learning
MSE	Mean Squared Error
MDP	Markov Decision Process
MTRL	Multi-Tasks Reinforcement Learning
NN	Neural Networks
POMDP	Partially-Observable Markov Decison Process
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RMSProp	Root Mean Squared Propagation

SELU	Scaled Exponential Linear Unit
SMAC	StarCraft Multi-Agent Challenge
SGD	Stochastic Gradient Descent
TD	Temporal Difference
TPRO	Trust Region Policy Optimization
VDN	Value-Decomposition Networks

Chapter 1

Introduction

Deep RL became the state-of-the-art method to solve RL problems [1, 2]. From games to real-life scenarios, Deep RL methods took precedence over other machine learning methods in solving human-like tasks up to the point of surpassing the human level in many scenarios [3]. However, many challenges in the field need to be addressed. Among these, we can mention the ability of a Deep RL agent to adapt to a difficult task after being introduced to a simpler one [4]. Many solutions have been proposed to address the adaptation issue, but in some situations where agents successfully manage to adapt to new tasks, they end up forgetting skills acquired on previous tasks. Another challenge of Deep RL in Multi-Agent systems is the coordination of behaviors between two or more agents to train together and reach a common goal. Many approaches have been proposed to address the problem. However, most of them investigated value-based algorithms. This project aims to investigate the ability of Deep RL policy gradient methods to learn and maintain coordination of behaviors when the training environment gradually becomes more difficult.

1.1. Motivation

Policy gradient algorithms have the particularity to directly learn the optimal policy, which makes them suitable for tasks with continuous spaces. However, in Multi-Agent systems, value-based methods have been the most used paradigms to solve cooperative tasks [5]. But value-based methods are not suitable for tasks with large spaces. Recent experiments have shown that policy gradient methods also can perform very well in cooperative scenarios if provided the appropriate state representation and hyperparameter tuning [6]. But still, most proposed approaches rely on the idea of centralized training and decentralized execution (CTDE) with parameter sharing. [7] has been able to apply Independent Learning on policy gradient algorithms to obtain effective results in The StarCraft Multi-Agent Challenge (SMAC) [8], which is one of the benchmark multi-agent tasks.

Except for environments of benchmark tasks, MARL suffers from a lack of test-bed

environments to compare algorithms, especially for continuous cases. And among the provided frameworks, implementations usually are not flexible to allow easy customization for research purposes. We developed a 3D environment [9] with a continuous state space on top of the implementation of a 2D in [10]. Motivated by the success of policy gradient algorithms shown in [6, 8] we studied the ability of the Independent PPO algorithms to cooperate in the 3D environment using Self-Play Learning because of the lack of expert models in the environment, and Incremental Learning because the experimental environment has been challenging even in the single agent setting.

1.2. Problem Statement

RL agents have shown impressive results in solving many human-like tasks. However, in some scenarios, the tabularasa[1] perspective fails when agents deal with complicated tasks. Agents struggle to adapt to complicated tasks when starting from scratch. Transfer Learning [11] has been applied in Reinforcement Learning to address this issue in many approaches: Curriculum Learning [12], Meta-Learning [13], Incremental Learning [14, 15], and many others. with the idea of leveraging knowledge acquired in given source domains to help adapt quickly to given target domains. Incremental Learning in particular uses the paradigm of decomposing the main task into a sequence of sub-tasks starting from the simplest to the most difficult. Then an agent will learn in order all tasks in the sequence until it reaches the final task, which is the main task. This approach requires similarities between tasks in the sequence to ensure successful knowledge transfer, otherwise, the sequential learning process can suffer from many issues such as mismatches between tasks [16], negative knowledge transfer [17], catastrophic forgetting [18], etc. that we are discussing more in Section 2.1. One of the ways to ensure similarities between tasks in the sequence is to subdivide the MDP of the main task into smaller MPDs [19] in such a way that the environment MDP will grow incrementally between successive tasks. In addition to addressing issues mentioned previously, this approach has the advantage of skipping the storage in memory of copies of prior policies and task parameters during sequential learning.

Coordination of behaviors to achieve a common goal is one of the challenges of Multi-Agent Reinforcement Learning (MARL). Unlike humans who have the ability to coordinate their behaviors to achieve a common goal, RL agents show difficulties adapting to each other during the training [20]. Nonstationarity, lazy and selfish agents phenomenon, scalability, and partial observability are examples of situations that make multi-agent learning a challenging field in RL. We discuss these issues in Section 2.2. Techniques such

[1] Theory assumes that individuals are born without built-in mental content, and therefore all knowledge comes from experience or perception.

as centralized training and decentralized execution (CTDE) address the nonstationarity issue but still suffer from scalability. Reward Shaping[21, 22, 23, 24] addresses issues such as sparsity of the environment, credit assignment, and the lazy agent phenomena in the multi-agent system.

Despite that, the tabula rasa paradigm works well with humans, in many situations, humans rely on experts' assistance for knowledge acquisition and skill improvement. This situation also applies to AI algorithms, especially in competitive environments. By repetitively facing an expert, RL agents can progressively adapt to the expert's behavior and learn how to tie or defeat the expert. However, the experts' existence is not always guaranteed, especially for handcrafted tasks. In many situations, handcrafted environments lack benchmark experts as baselines to train RL agents. Imitation Learning [25, 26, 27, 28], especially from human experts, has been applied as an alternative to address the lack of real-time interaction with an expert. But still, it requires a lot of data collection from experts' moves, which can be expensive or nonexistent. This approach also does not motivate the discovery of new strategies. Self-Play Learning [29], where the agent uses itself as a training coach, has been introduced as a training paradigm to address the lack of experts issue. The technique has shown impressive results in strategic games such as Go, Shogi, and Chess [30]. Furthermore, the technique has been effective in cooperative multi-agent games such as Dota2 [31]. Self-Play became a promising training method for competitive tasks.

This work investigates the coordination of behaviors of a team of two Independent PPO agents to solve a 3D volleyball game [9] by altering the original reward function provided by the game environment. As the 3D environment is more complex compared to the original 2D environment [10], we use Incremental Learning along with multi-agent training to explore the ability of the team to solve the 3D game. The lack of an expert baseline for this specific task motivated us to use Self-Play Learning along with Incremental Learning during the training.

1.3. Objective

The main objective of this work is to train a team of two independent policy gradient agents to solve a cooperative task in a virtual 3D space. Due to the failure of the single policy gradient agent to solve the 3D environment from scratch in a single agent scenario, we decided to start simply and incrementally progress toward our goal as follows:

1. Break down the environment MDP into a sequence of smaller MDPs and train a single PPO agent to solve tasks generated by each MDP in the sequence;

2. Investigate experimentally on the incremental setting that allows efficient learning and optimizes the training time to solve the full environment;

3. In the case of good results with the single agent, we apply the same paradigm in a cooperative multi-agent scenario to teach a team of two independent PPOs to solve the full 3D environment.

1.4. Contributions

This project shows that RL agents can solve a task in a complex environment after solving the same task in a similar and simpler environment. It shows that the optimal policy of the final task generalizes over all previously solved sub-tasks. It demonstrates that with appropriate reward shaping, independent policy gradient algorithms can learn cooperation without the necessity of tuning hyperparameters [6, 8]. It also shows the ability of policy gradient algorithms to handle Incremental Learning, Self-Play Learning, and Collaboration using the same policy network at the same time.

The project comes with an implementation of a 3D open-source environment [9] which can be used for multiple research purposes for both single and multi-agent scenarios. It also provides an implementation of a decentralized multi-agent PPO (MAPPO) built on top of the OpenAI Stablebaselines [32] PPO2 implementation and which can be used for different scenarios involving decentralized learning.

1.5. Project Outline

In Chapter 2 we discuss topics related to our project. We give an overview of Transfer Learning and subsequent methods as effective approaches to solving complex RL tasks. We discuss the notion of Multi-Agent RL and mention the challenges faced in the field, different propositions to address these challenges, and the limitation of these propositions. We discuss the notion of Reward Shaping as an effective approach to address issues such as sparse reward and cooperation in decentralized learning. We mention challenges faced in the field and discuss different solutions. Finally, we discuss the notion of Self-Play Learning, an effective approach that addresses the lack of experts issue in situations such as competitive scenarios.

In Chapter 3 we give an overview of RL. We discuss topics such as policy which is a probability function that RL agents rely on to perform decisions. We discuss the notion of value functions that help agents to build an optimal policy based on the reward received from the environment. We also discuss the different standard methods used in RL such as Q-leaning which uses an off-policy paradigm to learn an optimal policy. We finally give

an overview of the mathematical formalization of Multi-Agent RL, and different training approaches.

In Chapter 4 we discuss the notion of Deep Learning which is a field that investigates the structure and performance of Deep Neural Networks (DNN). We discuss techniques that NNs use to optimize parameters to generate the optimal approximation of the real data distribution.

In Chapter 5 we discuss the application of Deep Learning in RL. We discuss how Deep RL has been used as a paradigm to empower RL algorithms. We discuss the advantage policy gradient algorithms have on value-based algorithms. We introduced the Proximal Policy Optimization (PPO) algorithm that we use in the experimental chapter of this project.

In Chapter 6 we emphasize the notion of Incremental RL and how the paradigm has been applied in our project. We discuss the ability to solve a large MDP by breaking it down into a sequence of incremental MDPs. The MDP of the first task in the sequence is set to be simple enough to motivate the agent to adapt quickly. We introduced the notion of performance threshold that helps decide when during the training the agent can migrate to a bigger MDP. We also introduced the incremental step, which controls how big the next task MDP should be.

In chapter 7 we present the Slimebot Volleyball game, the environment we developed for the experiments. We start with an overview of the Gym environment, which is one of the standard paradigms to develop RL environments and helps researchers easily collaborate and compare results. We give an overview of the Slimevolleygym which is a gym-like environment to train RL agents to play the Slime Volleyball game in 2D. We then introduce Slimebot Volleyball which is the transposition of the Slimevolleygym environment to 3D. We discuss additional features we added in the 3D environment to make it appropriate for Incremental Learning.

In Chapter 8 we evaluate the experiments we did to solve the Slimebot Volleyball environment in both single and multi-agent scenarios. We start by investigating the single-agent scenario to evaluate the possible confluence the performance threshold and the incremental step have on the incremental training. After that, we investigate the ability to apply Incremental Learning in a decentralized training of two PPO algorithms to solve the 3D environment. We then discuss the experimental results.

Finally, in Chapter 9 we summarize the project. We discuss the success and failure of the experiments and provide directions for further investigation into similar problems with the 3D volleyball game environment.

Chapter 2

Related Work

In this chapter, we present an overview of research done in the fields of Incremental Learning, Reward Shaping, and Self-Play Learning in RL. We discuss generalities about Transfer Learning and its application in RL. After that, we give an overview of subsequent methods derived from Transfer Learning such as Curriculum Learning, Incremental Learning, Meta-Learning, Life-Long Learning, and Multi-Task Learning. We then introduce Reward Shaping, a technique that addresses many issues in RL such as sparse reward, and credit assignment in a multi-agent environment. We finally discuss Self-Play learning, a technique popularized in competitive RL research, where the learning agent uses itself as a training coach.

2.1. Transfer Learning

Transfer learning [11] is motivated by the human ability to adapt quickly to new tasks by leveraging experiences acquired on similar tasks. For example, in some cases, a person can shift to the non-dominant hand to accomplish a given task when the dominant hand is unavailable, although this may require a little bit of adaptation from the non-dominant hand to perform similarly to the dominant one. Since the popularization of transfer learning in the NIPS-95 workshop[1], the paradigm has been applied in RL to accelerate learning and improve the convergence rate [33]. Let π_s represent the knowledge acquired by the RL agent after being trained in a source task $\mathcal{T}_s$, the basic idea of Transfer Learning is to use π_s as an initial knowledge that can help the agent to quickly master a target task $\mathcal{T}_t$ and acquired a new knowledge π_t. However, this raises many fundamental questions [11]: which knowledge to transfer?[2] Where is the suitable place to operate with the transfer?[3] When should the transfer occur? And how to proceed with knowledge transfer?

From the 1995 workshop, Transfer Learning is investigated and generated different fields

[1] $http://socrates.acadiau.ca/courses/comp/dsilver/NIPS95_LTL/transfer.workshop.1995.html$

[2] In RL this can be policies, value functions, etc.

[3] The source domain and target domain might share some common properties to make the knowledge transfer possible

that sometimes bring confusion on how they differ from one to another. In this section, we are aiming to discuss explicitly subsequent fields of Transfer Learning.

2.1.1. Curriculum Learning

We can see Curriculum Learning[12] as an extension of Transfer Learning. It addresses the problem of generalization over a sequence of many tasks by leveraging knowledge acquired from previously solved tasks to adapt quickly to unseen tasks. Given a set of tasks, the agent has to learn all tasks sequentially following a given order and proceed with knowledge transfer from previously solved tasks to the current one. We can consider the first task to learn as the source task and the last task to learn as the target task. The agent has to start with the source task and sequentially go through all intermediate tasks in the sequence until it reaches the target task.

In addition to Transfer Learning's fundamental issues, the sequencing of tasks is another challenge for Curriculum Learning. Most of the time the task sequencing is handcrafted, this means that the programmer decides how to subdivide tasks and how to align them in sequence. While sequencing tasks, the programmer should ensure that they share common structures such as the same state or action space, same objective function, etc. otherwise the Curriculum Learning could be unsuccessful due to the mismatch between domains, which we discuss in section 2.1.

2.1.2. Incremental Learning

Incremental Learning[14, 15] is a particular case of Curriculum Learning, where the main task $\mathcal{T}_t$ is subdivided into a set of sub-tasks. The sequence of sub-tasks is set in a way that the previous task is easier than the current one. During the sequential training, the agent migrates sequentially to difficult tasks until it reaches the final task. Incremental Learning is motivated by the failure of the tabula rasa perspective while aiming to solve complex tasks. The complexity of a task can be noticed when RL agents fail to master the task or adapt slowly to the task. Incremental Learning addresses these issues by decreasing the training time and making trainable tasks that could not be solved from scratch. Despite the successful results of the method, there is still the challenge of how to split the main task into a sequence of sub-tasks and how to proceed with the increment.

2.1.3. Meta Learning

Unlike Curriculum Learning methods that generalize over multiple tasks in a predefined sequence, Meta-Learning[13] uses the paradigm of *training tasks* and *testing tasks* to address the issue of generalization over multiple tasks. The set of tasks is split into two subsets: a training set composed of tasks used to train the agent and a testing set composed

of tasks used to test the performance of the agent. Similar to Curriculum Learning, tasks in the Meta-RL settings are not disjoints. Different tasks can share the same state space, or they can have the same objective function with different state spaces. Adaptability to previously unseen tasks is only possible if these tasks share common structures with experienced tasks [34]. For example, a model trained to walk in a given environment has a high potential to adapt to a running task in the same environment. The meta-training is divided into two steps: meta-training and meta-testing. In the meta-training step, a set of tasks is given to the agent to train and generate a baseline policy, in the meta-test step, another set of tasks is given to evaluate the agent's performance. Both sets are drawn from the same task distribution and may differ one from another. The process is to train the baseline policy on the training set to maximize the expected rewards on the testing set [34].

2.1.4. Life-Long Learning

Learning has always been a continual process for human beings during their life [35]. In addition to their ability to leverage previous skills to acquire new ones, human beings have the remarkable potential to carry learned skills during their all lives as a continual process. However, this perspective is challenging in RL, especially for Deep RL algorithms. In most cases, by leveraging a previously acquired skill to get a new one, Deep RL agents usually suffer from catastrophic forgetting of prior knowledge while trying to adapt to a new task[18]. Life-long learning has been applied in RL to make the sequential learning a lifetime process for Deep RL agents [36, 37, 38], which means the agent will be carrying in memory prior knowledge of previously solved tasks during the continual learning process. Despite the approach working well, it can become memory intensive when the number of tasks grows big. We discuss more in Chapter 6.

2.1.5. Multi-Tasks Reinforcement Learning (MTRL)

Humans can combine different skills to solve a complex task and keep the solution as a new skill that can be used to solve a future complex problem. This motivates the idea behind Multi-task RL [39] which aims to solve multiple tasks at the same. The transfer of knowledge during the training is done by leveraging similarities across tasks to learn efficiently every single task. This requires tasks to have strong similarities in their structures. The RL algorithm A3C (asynchronous advantage actor-critic) [40] is an example of multi-task learning. During the training, different actors are deployed in parallel on different tasks[4] and report their gradient updates to a global network. The global network combines all learned parameters to generate a better set of parameters that all actors might refer to before continuing the training.

[4]This could be the same task duplicated in different environments

Learning across different tasks raises some challenges such as partial observability; scalability: a single task by itself requires long training interaction within the environment. Now with many tasks, the need for interaction grows and the learning can be very slow or impossible because the agent will struggle to transfer knowledge correctly across tasks. In the next section, we give an overview of some challenges faced in the field of Transfer Learning.

2.1.6. Challenges

Transferring knowledge across tasks is a field with many challenges, especially for RL agents using parameterized functions such as Neural Networks. Many algorithms such as DISTRAL (DIStill & TRAnsfer Learning) [41], IMPALA(Importance Weighted Actor-Learner Architecture) [42], PopArt [43], etc. have been proposed to address issues of Transfer Learning methods varying from scalability, distraction dilemma, partial observability, negative knowledge transfer, and catastrophic forgetting [44]. In this section, we discuss some of the challenges of Transfer Learning we mentioned earlier in this chapter.

1. Mismatch between domains

A mismatch of domains between the source task and the target task generates unsuccessful Transfer Learning [16]. This is when the structures of environments are different enough to not allow the proper knowledge transfer. We can see a mismatch between domains in many different ways. Let's consider a model trained using state observation in a multi-agent environment. Knowledge transfer becomes problematic if more agents are added to the environment. The agent observation should include signals (position, velocity, etc.) of the additional agents, this means the state observation becomes wider and cannot fit the input shape of the model received from the source task. For a model using pixels, the mismatch can be seen when the size of the input images from the source task is different from the size of the input images of the target task. Techniques like rescaling images can be an approach to address the problem. However, this can affect the training performance.

2. Negative Transfer

Incorrect state representation and the mismatch between domains can generate Negative Knowledge Transfer. Despite having that same input structure, if the environment structures or the objective functions are not similar, this can negatively affect the transfer learning process. An example could be the Atari 2600 games suits [45], teaching an agent to master the game suits in a Transfer Learning fashion can be problematic as each game differs in its structures, objectives, and visual representations. Transfer learning requires that both sources and target tasks be as similar as possible. An example can be the Meta-World[34] which is a set of 50 distinct robotic manipulation tasks sharing the same

state space and action space. Without appropriate similarities, the knowledge acquired in the source task can be useless in the target task. Hence, the training in the target task will have no difference as if the agent was trained from scratch in the target task.

3. Catastrophic Forgetting

Training agents with the ability to maintain previous skills while acquiring new ones is among the dream achievements in RL. However, differences between source and target domains and Negative Transfer bring another major challenge in the field of Transfer Learning. To master a given environment, Deep RL agents optimize their policy parameters to learn the training environment features in detail [5]. The gradient update will generate a model that optimizes the specific objective function. When asked to solve a new task, the gradient update will aim to optimize the new objective function which might lead to a considerable change in the model parameters. Hence, the new parameters may not be able to perform as well as before if the agent gets evaluated on the previously solved tasks. This issue is referred to as Catastrophic Forgetting [18].

A solution to this problem can be to slow down the learning of important model weights during the gradient update [18]. This can be done by using a small learning rate or reducing the reward signal in the new task. This could lead to a slow adaptation which, in contrast, is an issue that Transfer Learning tries to solve. Another approach to address this problem is to incorporate the idea of Life-Long learning[36, 37, 38], which aims to carry all the prior knowledge during the curriculum learning. At each transfer, a copy of the model representing the acquired skill is saved before starting the training in the target task. However, this approach shows weakness when the number of tasks increases and the memory allocation becomes problematic. In situations of successful Life-Long Learning, the agent should be provided with the ability to recognize the structure of a test task during an evaluation and the ability to associate with the appropriate model stored in memory to solve the test task. This also brings the challenge of task parameterization which can be problematic, especially in real-world environments.

2.2. Multi-Agent Reinforcement Learning (MARL)

MARL is a sub-field of RL that studies the interactions between many agents within the same environment. In recent years RL has shown considerable success in many scenarios such as games, robot manipulation, real-world, etc. However, most of these investigated single-agent systems. Many real-life situations require cooperation with other agents to successfully achieve a common goal or competition where individuals aim to maximize

[5] This depends also on their perception of the state space.

their gains at the expense of others. Hence, the interest in investigating the ability of multiple agents to learn to interact in the same environment [46].

2.2.1. Centralized Training Decentralized Execution

MARL was introduced with Independent Q-learning (IQL) [47] where each agent in the system optimizes a policy function using the Q-Learning algorithm[48] in a single agent fashion regardless of other agent improvements. Hence, the convergence to the optimal behaviors is not guaranteed as with Q-learning. The optimal behavior of an agent depends not only on the environment but also on the policies of other agents present in the system [49]. MADQN [50] is the transposition of IQL in deep learning where each agent optimizes its behavior using the DQN algorithm [51]. Due to the weakness of *decentralized* training, Value Decomposition Networks(VDN) [52] addresses the issue by introducing the notion of centralized training and decentralized execution (CTDE), where agents in the system learn a joint action-value function with *centralized* Q-learning while assuring a decentralized policy learning for each agent. QMix [53] follows the same logic of learning decentralized policies in a centralized fashion by adding a constraint between the joint action-value function and individual action-value functions. Despite the significant results in multi-agent tasks such as StartCraft II[53], IQL, VDN, and QMix are value-based algorithms, which means they are not suitable for environments with continuous state spaces. To address the issue, the notion of CTDE has been extended to policy gradient algorithms. MADDPG [54], and subsequent algorithms, are probably the most successful multi-agent policy-based algorithms published in the literature. MADDPG extends the notion of DDPG [55] in the multi-agent system, with DDPG a particular policy gradient algorithm that directly computes a deterministic policy. MADDPG uses the paradigm of Actor-Critic where different actors share the same critic network which takes into consideration all actors' moves and computes the best individual criticism for respective actors. Independent PPO, also called MAPPO performs similarly to IQL, where both learning and execution are decentralized among agents, this makes it less efficient compared to MADDPG. To bring the high performance of PPO [56] to the multi-agent setting, some tips such as reward shaping, and NN hyperparameter tuning can be applied to make the cooperative training successful [6].

Libraries such as Mava [57], PyMARL [58], RLlib [59] and others provide benchmark implementations of most of the algorithms mentioned above and many others.

2.2.2. Challenges of MARL

Despite the impressive success, single-agent RL has still many open challenges that need to be addressed. MARL does not escape RL issues and brings additional ones. We provide

an overview of some issues investigated in the field of MARL.

1. Nonstationarity

Constant changes in the MDP structure bring instability during training. In dynamic environments we can see scenarios where the state space constantly changes its structure; the action space may get additional actions or be amputated of some actions; the objective function can be altered, changing the reward signal. All those reasons can bring confusion to a learning agent. In the multi-agent setting, each training agent keeps changing behavior and makes the environment looks no stationary to other agents' perspectives [60].

2. Scalability

Deep RL agents refer to a memory allocation called a buffer where they store previous experience and proceed with the experience replay technique to improve the policy parameters. When the number of agents in the environment increases, the state space grows exponentially [61] such that storing experiences becomes memory intensive. An alternative can be sharing the same observation space such that only one buffer can be used to store states. However, this is unrealistic in real-world situations. There is also the problem of joint-action space which requires high computational resources and memory allocation as well [62].

3. Partial Observability

Full observability of the state space is unrealistic for many problems. Agents can only rely on their partial perception of the environment and local information received from other agents [63]. This requires a mechanism of information sharing to allow each agent to be aware of each other's information such as intention, location, action history, etc., and the ability of each agent to process that information for efficient policy improvement. This is one of the reasons that centralized learning of decentralized policies was introduced such that in addition to the observation each agent can receive extra information related to the environment from a supervisor network [62].

4. Lazy and Selfish Agents

Cooperative MARL consists of training a team of agents to optimize a common objective function. The aim is to incentivize all agents in the team to contribute to achieving the given goal. Depending on the scenario and the interaction settings, it can happen that, during the training, one of the agents quickly finds a way to generate good reward signals for the whole team. This has the consequence of demotivating other agents' contributions to the given objective. As the reward signal will be positive for everyone, their gradient update will lead other agents to non-optimal policies. Reward Shaping which we discuss

in the next section is one of the effective ways to address this situation by incentivizing equitable contribution of all agents in the common objective [64].

2.3. Reward Shaping

Reward Shaping [21, 22, 23, 24] is a method that encourages domain knowledge during the training by generating extra reward signals in addition to the original reward signal provided by the environment. The method is effective in addressing issues such as sparse-reward [65], and task sharing in cooperative multi-agent systems [66].

In the simplest form, we can see reward shaping as a signal $r' = r + F$ where r is the original signal and F is the shaped signal. In the situation where r is sparse[6], F will ensure that the agent receives feedback very often on the quality of its actions selection to motivate early domain knowledge. However, using F to decrease the sparsity can have consequences such as deviating the agent from the original objective, which can lead the agent to train and optimize an unexpected policy[67].

2.3.1. Reward Shaping in Single Agent Systems

A sparse reward task refers to an episodic task[7] where the agent receives the reward at the end of the task or rarely during the episode. Thus, the agent might struggle to differentiate between good and bad actions selected during an episode. In a dense reward task, where the agent receives a reward signal in the majority of timesteps, the agent quickly acquires domain knowledge to differentiate good and bad actions selected during the episode [65]. Designing a dense reward function correctly requires a good understanding of the RL problem and flexibility from the programmer. In addition, in some scenarios such as the board game tic-tac-toe, a dense reward function would not make sense at all. The success of training relies on the designed reward function. Algorithms using gradient descent optimization can get stuck in a sub-optimal solution and deviate totally from the expected behavior. Rewards shaping and subsequent methods [68, 69, 70, 71] address the sparse reward issue by modifying the original reward signal such that the agent will receive reward feedback more frequently.

2.3.2. Reward Shaping in Multi-Agent Systems

In a single-agent setting, reward shaping is applied as an alternative method to deal with sparse rewards. However, the paradigm has an additional utility in the multi-agent setting. In a multi-agent system where agents share the same objective function, reward shaping

[6]This means that the environment doesn't provide a non-0 reward signal in most timesteps.
[7]A task in which environment MDP has at least one conditioned terminal state.

is used to assign behavior credit to each agent in the system. When a team of agents receives a negative reward signal, agents will consider their respective last decision as a bad one regardless of the contribution of other agents' decisions. With a shaped reward signal, only the faulty agent receives the negative reward signal.

In a situation of a joint-reward function, redesigning the reward function can be useful to avoid the selfish and lazy agents phenomenon [64]. During training, the original reward function assigns the same credit to all agents regardless of their respective contributions. After many iterations with positive rewards, the agent generating the positive reward will be acting selfishly and the other agents will not progress as the reward function keeps informing them that they are doing well. This discourages cooperation and generates bad cooperative training.

Reward shaping has shown successful results in multi-agent systems such as the OpenAI Five[31] where each agent receives its particular reward as a linear combination of separate signals coming from the environment.

2.4. Self-Play Learning

From the iconic story of Deep Blue [72] the first computer program that defeated the world chess champion in 1996, machines have shown high potential to learn and defeat an expert opponent in competitive scenarios [30, 31, 73]. Competitive scenarios are particular cases of multi-agent learning where agents interact with one another with opposite goals. Competitive scenarios are one of the most successful research applications of RL. However, in adversarial games, an RL agent refers to an expert[8] as a training coach and will aim to generate the policy that reaches or surpasses the expert's level. From repetitive interactions with an expert in a given environment, RL agents can progressively generate the optimal behavior that reaches the expert's level or, in some cases they can surpass the expert's level. This requires the existence of an expert in the domain, a situation not always guaranteed in many scenarios, especially with handcrafted tasks for research purposes. An alternative to the real-time interaction is Supervised Learning, where the agent is fed with trajectory examples from one or more experts and will aim to generate the same behavior by proceeding with Imitation Learning [25, 26, 27, 28]. However, this requires data collection from the experts, if they exist, which can be tedious.

2.4.1. Self-Play Methods

Self-Play Learning [29] has been introduced as a novel approach to address the lack of an expert as a trainer in competitive and non-competitive scenarios. In competitive scenarios,

[8]This can be a human or another computer program

the agent learns from scratch how to defeat a clone of itself. The clone can be the real-time version of the learning agent or the best agent's copy stored in memory. Every time the agent defeats its last best copy, it generates a new best copy that will be the new opponent. By storing the last best copy, the agent will see its opponent getting stronger through the training. This requires the agent to adapt continually to a stronger opponent. Hence, we can see Self-Play Learning as a particular case of Incremental Learning. The paradigm can be applied in different fashions:

1. Asymmetric Self-Play

Asymmetric Self-Play Learning is a case of a non-competitive Self-Play Learning scenario. It is a situation where two sides are not pursuing the same goal. An example could be the non-adversarial scenario where two agents, named Alice and Bob, are interacting in the following way: Alice generates a trajectory by performing a task and then challenges Bob to do or to undo the same task[74]. Another example can be the famous Hide and Seek[75] where the hiders learn how to hide and seekers how to find the hiders.

2. Symmetric Self-Play

Symmetric adversarial games are situations where all sides have the same objective. An example could be Capture the Flag[76] where each team aims to capture the flag of the opposite team while protecting their own flag. In this work, we are applying the paradigm in a Symmetric scenario.

2.4.2. Turn-Based Games

Turn-based games are scenarios where at least two sides play in turn following a given order. Only one side is allowed to make a move in a given timestep, meanwhile, the other side stays still. The situation is reversed in the next timestep or depending on the game rules. Board games such as tic-tac-toe, Chess, Go, etc. are examples of turn-based games.

RL algorithms have shown many successes in Turn-based board games, surpassing expert human level. One of the greatest breakthroughs was in 2015 when the DeepMind RL program *AlphaGo*[77] defeated the Go world champion. The algorithm mixed Imitation Learning from human experts' examples and *Self-Play* Learning. Furthermore, a more interesting breakthrough showed up when AlphaGo was in its turn dethroned in 2017 by its successor *AlphaZero*[30] who learned from scratch, only with Self-Play Learning, how to play games such as Chess, Go and Shogi.

2.4.3. Simultaneous Games

In turn-based games, one side has often more privilege than the other, in tic-tac-toe, for example, the first player has an advantage over the second player. In case both players are experts, the first player tries always to win the game, while the second play will only target a tie. Simultaneous games bring more fairness to the competition. All sides are allowed to act independently and simultaneously as long as they follow the competition rules.

OpenAi Five[31] is probably the biggest breakthrough of Self-Play Learning in simultaneous zero-sum games. The multi-agent AI system has been able to defeat a team of human experts in the challenging game [31]. OpenAI stated that OpenAI Five learned to play Dota2 through Self-Play Learning for an equivalent of 10000 years of gameplay.

2.5. Summary

In this chapter, we have seen an overview of research done in different fields applied to the present project. We started by discussing the notion of Transfer Learning, which is a learning paradigm that focuses on leveraging knowledge among different tasks. We kept our attention on Incremental Learning which is a subsequent method of Transfer Learning that consists of breaking down a given task into a sequence of sub-tasks and where a learning agent will learn incrementally all tasks in the sequence from the easiest to the most difficult, and by leveraging knowledge across successful tasks. We mentioned that learning tasks in sequence bring different issues such as Negative Transfer and Catastrophic Forgetting. Methods such as Life-Long Incremental RL allow addressing these issues but have a disadvantage in terms of memory allocation, task parametrization, etc. In Chapter 6 we will see that including the full MDP of the previously solved task in the next task can be an effective approach to address the memory allocation issue of Life-Long Learning and the Catastrophic Forgetting or prior knowledge while adapting to new tasks. We will also see that optimal behavior in a source task is not a requirement before migrating to a more difficult task as long as the gap of difficulties is reasonable.

We also discussed the notion of MARL, its successes, and challenges. We saw that the notion of CTDE often brings better outcomes compared to Independent Learning in cooperative scenarios. But we mentioned also that most of the benchmark results are centered on value-based algorithms, which are not suitable for continuous environments. We mentioned that some empirical results demonstrated that techniques such as hyperparameter tuning, can bring good results in cooperative MARL using independent policy gradient algorithms. In our empirical experiment we demonstrate that with only proper reward shaping, independent policy gradient algorithms can perform well in cooperative scenarios using

the same hyperparameter initialization as with single-agent scenarios.

We finally discussed the notion of Self-Play learning, which is a technique of training an algorithm against copies of itself. The technique shows its advantage when the experimental environment lacks an expert benchmark to compare other algorithms. We applied the technique in our experimental environment to study the ability of policy gradient algorithms to solve an incremental competitive task in the single agent setting and to maintain coordination of behaviors in an incremental cooperative-competitive task.

The task we use is based on a famous 2D volleyball game played by 2 slimes [10], the game has been successfully solved by a PPO algorithm. We extended the game in 3D [9] and explored the ability of the PPO algorithm to solve the game in both single and multi-agent scenarios with Incremental Learning. Further details will be discussed in Chapter 8.

Chapter 3

Reinforcement Learning

Reinforcement Learning (RL) is a set of methods that aim to train agents deployed in a given environment to solve specific tasks by progressively improving their behaviors using a function called a *policy*. The policy improvement occurs through repetitive sequences of trial and error. After selecting an action, the environment generates feedback indicating to the agent how good or bad was the selected action. Based on the feedback received, the agent will aim to adjust its policy function to avoid negative feedback and maximize positive ones in the future.

RL is a wide topic covering many theories that address different types of problems. In this chapter, we are reviewing general concepts to introduce readers to the framework. For more details in fundamental RL, we recommend the main book from Sutton and Barto [78].

3.1. Markov Decision Processes

An RL problem can be mathematically formalized with the notion of the Markov Decision Process (MDP) [78]. An MDP is a sequence of states in an environment dynamic that assumes the Markov property given in equation 3.1. In other words, the future state depends only on the current state. In RL we can describe an MDP as a four-tuple [79]: $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where:

- $\mathcal{S}$ represents the set of all possible states of the environment;
- $\mathcal{A}$ is the action space or the set of all possible actions;
- $\mathcal{P}$ the state transition probability to migrate from a given state to another;
- $\mathcal{R}$ the reward feedback received from the environment after taking a particular action in a given state.

$$P\left[S_{t+1}|S_t, S_{t-1}, \cdots, S_0\right] = P\left[S_{t+1}|S_t\right] \tag{3.1}$$

At time t the agent is in the state S_t and will select an action A_t with a probability p_t, from which it will receive a reward R_t and will migrate to another state S_{t+1} at time $t+1$. As shown in Figure 3.1, the trajectory of the agent can be described as a sequence of:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \cdots \tag{3.2}$$

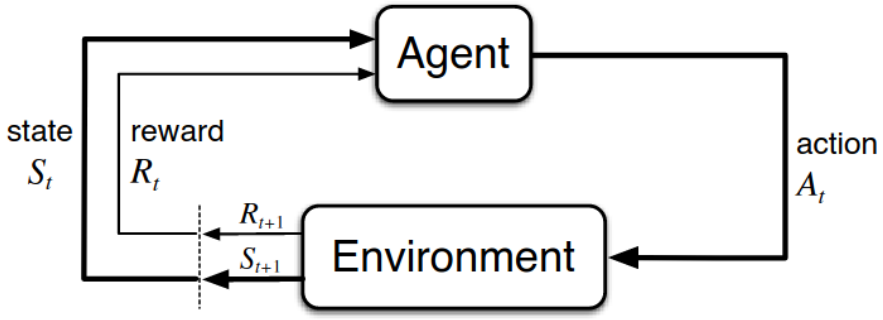


Figure 3.1: agent-environment interaction loop in RL [78]

Hence, if we consider $\mathcal{T}$ as the time horizon, we can describe an RL problem as a family $\{(S_t, A_t, P_t, R_t)\}_{t\in\mathcal{T}}$ where the transition between states is controlled by a probability distribution of random variables R_{t+1} and S_{t+1} constrained by the actual state S_t and the selected action A_t. Sutton and Barto [78] defines the transition probability as:

$$p(s', r|s, a) = P\left[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\right] \tag{3.3}$$

In other words, we are measuring the probability that the agent transits to the state s' and receives the reward r knowing that the agent selected the action a while being in state s.

3.2. Policy

Equation 3.3 tells us that the next state depends on the current state and the selected action. This means at the current state the agent has to decide which action to select among all possible actions. To know the best action to select in different states, the agent relies on its decision-making function called a *Policy*, which is a map from the set of states to the set of probabilities of selecting an action. Let π_t denote the policy at time t, π_t is represented as the conditional probability :

$$\pi_t(a|s) = P[A_t = a|S_t = s] \tag{3.4}$$

π is denoted as a *stochastic* policy because it assigns a probability distribution over the action space. In other words, each action of the action space has a chance to be randomly selected according to a probability distribution using equation 3.4. A *deterministic* policy μ on the other hand computes directly the action a given the current state s, as shown in equation 3.5. This is can be done by assigning different numerical values to actions according to the current state, the n-armed bandits' problem can be an example [78]. It is logical to think that the agent will aim to select the action with the highest value at the current state.

A *greedy* policy is a policy that selects the action with the highest value.

$$\mu(s) = a \tag{3.5}$$

To build the policy function, the agent refers to cumulative numerical values, called rewards, received from the environment as feedback of previously selected actions. Those cumulative rewards are referred to in the literature as *returns* and are the key ingredient for policy improvement in RL.

3.3. Return

The main objective of an RL agent is to generate a policy that maximizes the expected return. The return G_t at time t is defined as the summation of all future rewards from time t:

$$G_t = \sum_{k=1}^{T} R_{t+k} \tag{3.6}$$

With T the final step of the MDP history. The notion of Return is suitable for episodic tasks[1]. However, it is not appropriate for problems with continuous tasks[2]. Because, if the number of timesteps increases ($T \to \infty$), as a summation function, the return will increase as well ($G_t \to \infty$) and will not be appropriate for the policy improvement, also the summation over a long list of future reward can be computationally expensive. To address the problem, the *discounted* return [78] is used in place of the original one. Hence, the equation 3.6 can be written as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.7}$$

where $0 \leq \gamma \leq 1$ is the discount factor that helps to value the future rewards compared to the present ones.

[1] Tasks with at least one conditioned terminal state S_T
[2] Tasks without terminal state or very long episodic tasks

NOTE: By return G_t we will be referring to equation 3.7 in the rest of the report.

G_t can be expressed recursively by using incremental computing of successive returns as follows:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \tag{3.8}$$
$$= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \cdots\right) \tag{3.9}$$
$$= R_{t+1} + \gamma G_{t+1} \tag{3.10}$$

with $G_T = 0$ in a situation of an episodic task.

We can see the return G_t as a measure of the preference to be at state s_t at time t or, in situations where the chosen action is considered, the importance measure of selecting action a_t in state s_t.

3.4. Value Function

We mentioned earlier that deterministic policies rely on values attributed to states or state-action pairs to perform action selections. As said in the previous section, the value of a state s under a policy π is denoted as $v_\pi(s)$, and is the expected return of following π from state s [78]:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{3.11}$$

Algorithms that compute the value of different states of an environment MDP are denoted as *state-value* algorithms. For some cases, the action selected in a given state is considered and a value is attributed to the *state-action* pair. Those algorithms are referred to as *action-value* algorithms. We denotes by $q_\pi(s, a)$ the expected value of the state-action pair (s, a) and it is given as follows :

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{3.12}$$

Sutton and Barto [78] proposed a recursive representation of the state-value function as:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] && (3.13)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_t|S_t = s] && (3.14)\\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']] && (3.15)\\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + v_\pi(s')], \quad \forall s \in S && (3.16)
\end{aligned}
$$

Similarly, with the action-value function, we have:

$$
\begin{aligned}
q_\pi(s, a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] && (3.17)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_t|S_t = s, A_t = a] && (3.18)\\
&= \sum_{s',r} p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']] && (3.19)\\
&= \sum_{s',r} p(s', r|s, a) \left[r + \gamma \sum_{a'} \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s', A_{t+1} = a']\right] && (3.20)\\
&= \sum_{s',r} p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')\right], \quad \forall s \in S && (3.21)
\end{aligned}
$$

Equations 3.16 and 3.21 are the foundation of multiple value-based methods in RL. They are known, respectively, as the *Bellman* equations for the state-value function v_π and for the action-value function q_π under the policy π.

As the real value function is unknown to the agent, the agent relies on an estimate V_π of the sate-value function v_π, or an estimate Q_π the action-value function of q_π. In the following section, we are giving one of the multiple ways an estimate of the real value function can be computed.

3.5. Action Value Estimation

Let q be the real value of action a while in state s. Most of the time q is unknown to the agent. Hence, the agent can rely on an estimate Q_k of q at time t, with $k < t$ the number of times the action a has been selected up to time t.

Given $r_1, \cdots, r_{l-1}$, successive rewards received by selecting action a up to time t. Q_k can be computed by using the *sample-average* method [78] :

$$Q_k = \frac{r_1 + \cdots + r_{l-1}}{k - 1} \tag{3.22}$$

As mentioned in section 3.3, this can be memory expensive when $k \to \infty$ as it requires

the storage of all past rewards, and it can be computationally expensive as a summation should be performed over the list of past rewards.

Using incremental computation equation 3.22 can be written as [78]:

$$Q_{k+1} = Q_k + \frac{1}{k}\left[R_k - Q_k\right] \tag{3.23}$$

Equation 3.23 can be explained as:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}[\text{Target} - \text{OldEstimate}] \tag{3.24}$$

In practice, the StepSize in 3.24 is a default constant value α with $0 < \alpha < 1$. There are many other ways an action value can be estimated.

3.6. Optimality

Let's consider $\Pi = \{\pi | \pi \text{ is a policy } \}$ the set of all policies for a given MDP. Let $\leq$ be a partial order defined in Π as follows:

$$\pi \leq \pi' \iff v_\pi(s) \leq v_{\pi'}(s), \forall s \in S,\ \pi, \pi' \in \Pi \tag{3.25}$$

In the assumption that Π is finite, $\exists\, \pi^* \in \Pi, \forall \pi \in \Pi$ s.t. $\pi \leq \pi^*$. Hence, π^* is the optimal policy that maximizes the return. During the training, the agent will proceed with sequences of policy improvements until it reaches π^*. This introduces the notion of optimal state-value function and optimal action-value function for deterministic policies as:

$$v_*(s) = \max_\pi v_\pi(s) \tag{3.26}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{3.27}$$

Sutton and Barto [78] showed that q_* can be expressed with the optimal state-value function v_* as follow:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] \tag{3.28}$$

The Bellman optimal equation 3.16 expresses the optimal value v^* of a given state s under

the optimal policy π^* as the expected return of the best action in s:

$$
\begin{aligned}
v^*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi^*}(s,a) && (3.29)\\
&= \max_a \mathbb{E}_{\pi^*}\left[G_t | S_t = s, A_t = a\right] && (3.30)\\
&= \max_a \mathbb{E}_{\pi^*}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] && (3.31)\\
&= \max_a \mathbb{E}_{\pi^*}\left[R_{t+1} + \gamma b_*(S_{t+1}) | S_t = s, A_t = a\right] && (3.32)\\
&= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] && (3.33)
\end{aligned}
$$

Equations 3.32 and 3.33 are the principal forms of the Bellman optimal equations for the optimal state-value v^*. The Bellman optimal action-value forms are given as follows:

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] && (3.34)\\
&= \sum_{s',r} p(s', r|s, a)[r + \gamma q_*(s', a')] && (3.35)
\end{aligned}
$$

In practice, the agent doesn't usually reach the optimal policy or optimal value function. Hence, the objective is to approximate them the closer as possible. As the estimated values of actions change and get more accurate through the training, the policy function will also keep improving.

We mentioned earlier that, ignoring the true value of an action, the agent relies on estimation based on previous experience. However, the estimation can be wrong. To ensure training efficiency, the Exploration[3] and Exploitation[4] trade-off [80] has been introduced as a paradigm where the agent selects the action with the highest estimated value according to a given probability, otherwise, it will select randomly a non-optimal action. The paradigm is also referred to as the ϵ-greedy action selection strategy, where the agent decides to take a greedy action with probability $1 - \epsilon$ and a non-greedy action with probability ϵ. This approach has the benefit of discovering rewards associated with different actions.

3.7. Methods

RL is a universe full of different methods, and preference among RL methods depends on the problem to solve. In general, algorithms are separated into classes such as meta-heuristics, policy gradient, value-based, and model-based methods [81].

[3]When the agent is always choosing the greedy action.
[4]When the agent decides to select a non-greedy action.

3.7.1. Value-Based Algorithms

An RL agent using a value-based algorithm learns a state-value/action-value function that refers to select actions. We can extract 3 categories of value-based algorithms:

- Dynamic programming: also referred to as model-based methods, they required a complete knowledge of the environment MDP;
- Monte Carlo: Referred to as model-free methods, they do not require full or partial knowledge of the environment dynamic, they use samples of previous experiences to generate an optimal policy;
- Temporal-Difference methods: Methods that make a bridge between the two previous methods, by taking the model-free property of Monte Carlo methods and the incremental property of Dynamic programming.

Depending on the nature of the problem to solve, each of these methods has weaknesses and strengths compared to the others [78].

3.7.2. Dynamic Programming

Dynamic programming (DP) is a set of methods that compute optimal policies given perfect knowledge of the environment structure [78]. To build the optimal policy, DP proceeds with iterative sequences of policy evaluation and policy improvement following the Bellman equations 3.16 and 3.21.

1. Policy Evaluation

Policy evaluation is the process that consists of computing the estimate of the state-value v_π knowing π [82]:

$$v_{k+1}(s) = \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s\right] \tag{3.36}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right] \tag{3.37}$$

2. Policy Improvement

Based on the estimated value functions, policy improvement consists of generating a policy π' better than π ($\pi \leq \pi'$) by acting greedily:

$$q_\pi(s,a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\right] \tag{3.38}$$

$$= \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \tag{3.39}$$

3. Policy Iteration

π has been improved using v_π and generated π', π' helps to compute $v_{\pi'}$ which will generate π''. Hence, we have an iterative sequence of policy evaluation and improvement:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

with $\xrightarrow{E}$ and $\xrightarrow{I}$ representing respectively policy evaluation and policy improvement. And for $i < j$, π_j is guaranteed to be a strict improvement of π_i.

3.7.3. Monte-Carlo

The Monte Carlo (MC) method is based on the idea of learning from previous experiences. MC methods don't require knowledge of the environment structure. The learning process relies on the collection of historical experiences, which are used to compute the mean return as an approximation of the expected return. This implies that MC methods are suitable for episodic tasks (tasks with terminal state s_T) to compute the empirical return.

3.7.4. Temporal-Difference

There are two categories of value-based methods: On-policy methods that consider the existence of an optimal policy and will aim to approximate it by doing iterative sequences of policy evaluation and improvement, TD method SARSA[78] is an example. There are also Off-policy methods where the update of the current value function doesn't rely on a particular policy, the TD algorithm Q-learning[48] is an example.

1. Formalism of TD Learning

As with Monte Carlo methods, prediction of the value function in TD Learning relies on the experience of following a particular policy π. The major difference is that MC methods need a full episode history to compute the return G_t at time $t \in \{1, \cdots, T\}$, then use it as

a target to update the actual value function $V(S_t)$ as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{3.40}$$

with $\alpha \in [0, 1]$ the step size parameter, with the role of controlling how big the update should affect the current value [78].

TD methods on the other hand don't need a collection of a full episode history, the update is applied directly after moving to the next step. The simplest update is known as the $TD(0)$ algorithm and is expressed as:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{3.41}$$

This means that the new estimate old estimate of $V(S_{t+1})$ is used to update the estimate of $V(S_t)$. The quantity $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ measures the difference between the $V(S_t)$ and the best estimate of $R_{t+1} + \gamma V(S_{t+1})$, it represents the difference between successive predictions and is called the *TD error*. This is the reason for the expression *Temporal difference learning*, it represents the error between successive predictions.

3.7.5. Q-learning: Off-Policy TD Control

Q-learning is an off-policy model-free algorithm introduced by [48] in 1989. It is one of the most imported breakthroughs in RL. The algorithm computes the Q function $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, which estimates the action-value of a given state-action pair [79] and stores them in a table called a Q-table. The simplest form of the algorithm is given as a one-step TD update:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{3.42}$$

Where S_t and A_t are respectively the current state and action, S_{t+1} the next state after selecting A_t, α the learning rate, γ the discount factor, and $\max_a Q(S_{t+1}, a) - Q(S_t, A_t)$ is the TD error, or the difference between the maximum Q value that can result from S_{t+1} and the Q value of the current state-action pair (S_t, A_t). [83] demonstrated that Q-learning guarantees convergence to the optimal action-value function Q^*.

The Q value of all state-action pairs is updated following equation 3.42 except for the terminal state S_T in which the Q value is considered as the state-value V and set to be the last reward of being at S_T: $Q(S_T, a) = R_T$. The Q-learning process can be represented by the following algorithm 3.1:

Algorithm 3.1: Q-learning

Initialize $Q(s,a)$, $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}$ arbitrarily and $Q(s_T,a) = 0$
Repeat(for each episode):
    Initiaize S
    Repeat(for each step of episode)
        Choose A from $\mathcal{S}$ using policy derived from Q (e.g.: $\epsilon-$greedy)
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$
    until S is terminal

1. Advantages

- Model-free: The optimal strategy is constructed just based on delayed rewards without complete or even partial knowledge of the environment dynamic.

- Off-policy: The convergence to the optimal policy is independent of the current policy. The agent can learn from experience generated by different policies than the current one.

2. Desavantages

- **Slow convergence**: To converge the algorithm needs to explore the state space sufficiently. In the situation of large state space, the convergence can be very slow due to the number of state-action pairs [84]. Thus, it takes a long to reach the optimal value function.

- **Memory inefficient**: The algorithm stores values of state-action pairs in memory. In situations of large state environments such as the game chess in which the number of states is estimated above 10^120 [85], memory allocation becomes problematic. The reason why the standard Q-learning algorithm is suitable for environments with small state space and action space such as the board game Tic-tac-toe. To address the problem of large state space environments, some variants of Q-learning, such as Deep Q-Network (DQN) [51] which uses Deep Neural Networks, and does not store the Q-values but approximate the Q function using function approximators.

- **Stationarity**: The algorithm assumes that the environment structure is stationary, this means the transition probability between states is fixed. Changes, due to factors such as the influence of other agents, in the structure of the environment during the training can affect the learning and the Q-value. That is the reason why Q-learning works better using the exploitation and exploration trade-off. Also, the problem of non-stationary can be addressed by making adjustments to the Q-learning algorithm [86].

3.8. Multi-Agent RL Framework

As seen in section 2.2, MARL is a set of methods that explore the interactions between 2 or more agents in the same environment. It is a field with many challenges due to the complexity of having many agents interacting with one another in the same environment. In this short section, we discuss the paradigm of MARL.

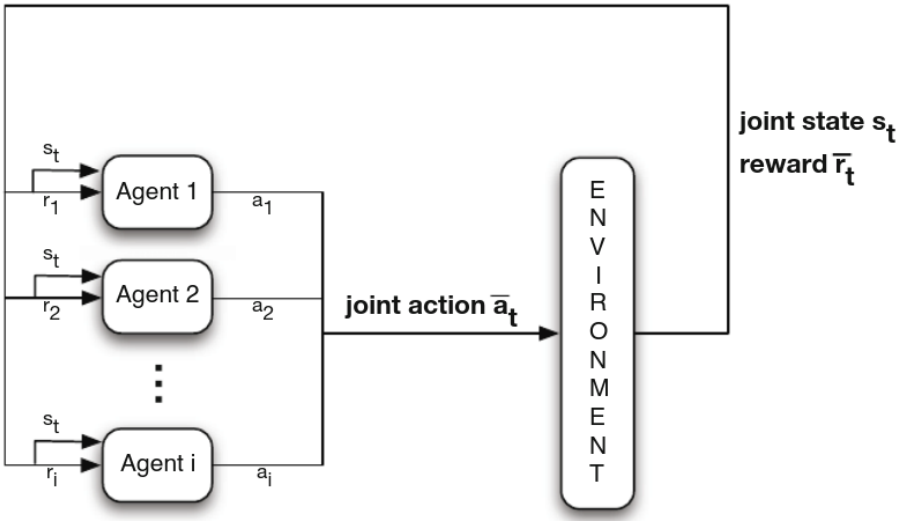


Figure 3.2: multi-agent interaction in a given environment [49]

3.8.1. MARL Formalization

While MDPs are used to formalize single-agent RL tasks, the paradigm is not suitable in multi-agent settings due to the nonstationarity agents bring to the environment structure. Partially-Observable Markov Decision Process (POMDP) brings a generalization of MDP by taking into consideration uncertainty regarding the state of a given MDP and permits state information acquisition [87]. The MDP is seen as a Markov game [88] and can be expressed as a 5-tuple: $\langle N, \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}^i_{i\in[1,N]} \rangle$ where:

- $N \geq 2$ represents the number of agents in the environment;
- $\mathcal{S}$ represents the state space observed by all agents in the system;
- $\mathcal{A} = \prod_{i=1}^{N} \mathcal{A}^i$ represents the joint action space with $\mathcal{A}^i$ the action space of the agent i.
- $\mathcal{P}$ the transition probability to migrate from a given state to another
- $\mathcal{R}$ depending on the scenario this can be a family of reward functions $\{\mathcal{R}^i\}_{i\in[1,N]}$ with $\mathcal{R}^i$ the reward signal of agent i, or a single joint-reward function, especially in cooperative scenarios.

3.8.2. Objective Function in MARL

The objective function in MARL is defined according to the given scenario. In situations of a shared objective function, the goal of each agent will be to maximize the same expected return define in equation 3.6. We have seen in section 2.2 this usually lead to the selfish and lazy phenomena if one of the agents start playing optimally meanwhile other are still weak. In situations of individual objective functions, we can consider 2 cases:

1. Identical Interests

In scenarios where agents receive different reward signals from the environment but have the same goal, the overall reward r can be defined as a summation of all individual rewards:

$$r = \sum_{i=1}^{n} r_i, \qquad \text{With } n \text{ the number of agents} \tag{3.43}$$

Hence, the multi-agent training goal will be to maximize equation 3.6 using r defined in equation 3.43 as reward [89].

2. Individual Interests

Individual interest scenarios are situations where each agent has a specific goal while interacting with other agents. It can be a completely separate goal situation such as Hide and Seek, or a shared reward function scenario where the original reward function has been altered via Reward Shaping [21, 22, 23, 24], and thus generated individual objectives for each agent. In those situations, the objective function will be a family of objective functions: $J(t) = \{J_i(t), i = 1, \cdots, n\}$, where each agent i will aim to maximize the objective function $J_i(t)$ define as:

$$J_i(t) = G_t^i \tag{3.44}$$

with G_t^i the expected return define in equation 3.6, of respective agents.

3.9. Summary

This chapter aimed to introduce readers to foundations that built the field of RL. We discussed the notion of MDPs that allows a mathematical formalization of RL problems which permits applications of a range of solution techniques. We viewed the goal of RL agents is to generate a decision function called a policy that maximizes the expected return while interacting with a given environment. We discussed that stochastic policies attribute a probability distribution over the action space such that each action will have a given chance to be selected. However, value-based algorithms compute a deterministic policy that decides the best action to select in a given state. We also mentioned that mixing

exploration and policy exploitation encourages the discovery of an accurate estimate of the value function.

We gave an overview of the tabular Q-learning algorithm, an off-policy algorithm that uses the temporal difference paradigm to build a value function represented in a table. We said that Q-learning converges to the optimal policy, but requires long interactions with the environment because of its slow adaptation. We also mention its weakness to generate the Q-table in a situation of large state space. And we mentioned that variants such as DQN address the large state space issue but use function approximators instead of a Q-table.

We also introduced a general formalization of MARL. We have seen that MARL is formalized using the notion of Partially-Observable Markov Decision Process (POMDP) instead of MDP because at a given timestep each agent has its private observation of the state. We also viewed that the optimization of a MARL problem depends on the context: shared objective function, identical interest with a separate reward function, and individual interest with a separate reward function.

Chapter 4

Deep Learning

Deep Learning [90, 91] is a class of methods that learn to model complex and large data. Deep Learning techniques outperform classical machine learning techniques. Deep Learning consists of the implementation of deep neural networks, which are artificial neural networks (ANN) with multiple layers. Deep learning has become the most popular breakthrough in Artificial Intelligence due to its ability to learn a larger number of features from raw unstructured data by passing them through several layers from an input layer to an output layer. The term *Deep* stands for the presence of many hidden layers between the input layer and the output layer. The more layers we have, the deeper the neural network. Deep Learning architectures have been applied in various fields such as object recognition [92], speech recognition [93], Natural Language Processing (NLP) [94], and RL [77].

In this chapter, we give a general overview of the field to familiarize readers with the techniques involved in our project. For a deep understanding of the subject, we recommend the book [91] from Ian Goodfellow and Yoshua Bengio, and Aaron Courville.

4.1. Artificial Neural Networks

Based on animal brain models, an Artificial Neural Network (ANN), or just Neural Network (NN), is a collection of connected artificial neurons. Following the biological structure of animal neurons, each connection between neurons can transmit information from one neuron to another. The expression *Feedforward* NN is used when information transcends from an input layer to an output layer. The inputs of a neuron are the dot product of a given feature vector and the weight vector of the neuron's input connections. The output of a neuron is a nonlinear transformation of the summation of all of its inputs.

4.1.1. Mathematical Formulation

Mathematically a neuron is defined as a function:

$$h(x) = \phi\left(\langle w, x\rangle + b\ \right) \tag{4.1}$$

With $x = (x_1, \cdots, x_n)$ the input feature vector, $w = (w_1, \cdots, w_n)$ the weight vector and $b \in \mathbb{R}$ is the bias. The function $\langle , \rangle$ represents the dot product. The function ϕ is the nonlinear activation function. Figure 4.1 shows a representation of an artificial neuron. The simplest form of NN is the *single-layer* perceptron, which consists of a single layer of output nodes.

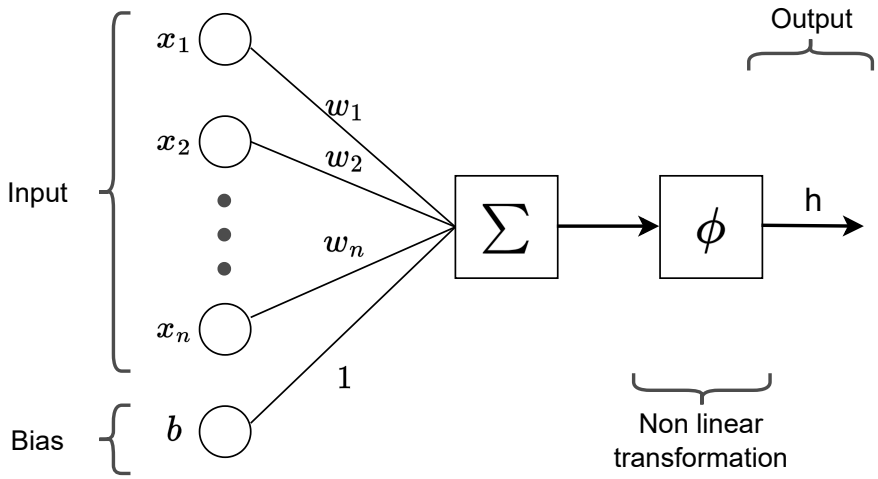


Figure 4.1: Illustration of an artificial neuron

4.1.2. Activation Functions

Different activation functions can be considered depending on their performance and the nature of the problem to solve.

1. **Rectified Linear Unit (ReLU)**: Relu [95] takes a real number as input and outputs the same number if it is positive otherwise it outputs 0;

$$\phi(x) = \max(0, x) \tag{4.2}$$

   It is the most used activation function for many types of NNs because of its simplicity to train NNs and often brings high performance. It generated many variants such as ELU [96], Leaky ReLU [97], and SELU [98], just to mention a few.

2. **Sigmoid**: Also called Logistic or Soft Step, the sigmoid activation is given by the following formula:

$$\phi(x) = \frac{1}{1 + \exp(-x)} \tag{4.3}$$

Sigmoid is effective with binary classification problems. It can be used in the output layer to produce values between 0 and 1, which can be interpreted as probabilities of each class.

3. **Hyperbolic tangent (tanh)** :

$$\tanh(x) = \frac{\exp(x - \exp(-x}{\exp(x) + \exp(-x)} \tag{4.4}$$

tanh outputs values between -1 and 1, which can suitable for problems where the output needs to be bounded in this range. It is also suitable for problems where the input data is averaged at zero.

4. **Binary**:

$$\begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases} \tag{4.5}$$

Suitable for two-class classification problems, the binary activation function produces binary outputs of 0 or 1 for each class.

5. **Identity**: The input layer activation functions are also identities:

$$\phi(x) = x \tag{4.6}$$

Useful when using a pre-trained network for transfer learning, the identity activation function helps preserve the original output range of the pre-trained network.

4.2. Deep Neural Networks

Deep Neural Networks (Deep NN) are extensions of the notion of Artificial Neurons. With Deep NNs multiple neurons are connected sequentially to process different levels of feature extraction of a given input data.

4.2.1. Multi-Layer Perceptron

The fundamental component of a Deep NN is the multi-layer perceptron, which consists of feedforward NNs with at least one hidden layer. The basic idea is that the output of a neuron in a given layer becomes the input of a neuron in the next layer until the output layer is reached. There are particular cases where the output of a neuron can be the input of a neuron in the same layer or a previous layer.

Thus, we can write recursively the output of layer i as :

$$h^{[i]}(x) = \phi^{[i]}\left(\langle w^{[i]}, h^{[i-1]}(x)\rangle + b^{[i]}\right), \quad i \in [K+1] \tag{4.7}$$

where:

- $w_i = \left(w_1^{[i]}, \cdots, w_d^{[i]}\right)$: the connection weights vector of layer i
- $h^{[i]}(x) = \left(h_1^{[i]}(x), \cdots, h_d^{[i]}(x)\right)$: The output vector of nodes of the previous layer, the value of d can be different for different layers and $h^{[0]}(x) = x$
- $\phi^{[i]}$ is the activation function at layer i, different layers can have different activation functions.
- $b^{[i]}$ is a real number, which represents the bias at layer i

4.2.2. Learning

From raw input data, the NN first performs a series of mathematical operations that transform the input into a form that can be useful for the NN to make predictions or perform some other tasks. For prediction problems, the NN is given real examples, also called targets, and will process comparisons between its outputs and the real examples. To improve its prediction, the NN optimizes its weights matrix parameters using some optimization processes. Depending on the nature of the output, which can be a real number or a vector, comparison with reality is performed using some metrics known as *Loss Functions*. NNs refer to Loss Functions to update their weight matrices by following a given optimization algorithm, which minimizes as much as possible the difference between prediction and reality.

4.3. Loss Functions

A loss function also referred to as *Cost Function* or *Error Function* is a differentiable metric function that measures the margin between two sets of variables. Different loss functions are used in Deep Learning depending on the NNs architecture and the problem to solve.

4.3.1. Mean Squared Error (MSE)

MSE, mostly used for regression problems, computes the mean squared difference between two vectors. Let $Y = (Y_1, \cdots, Y_n)$ be the target vector and $\hat{Y} = (\hat{Y}_1, \cdots, \hat{Y}_n)$ be the

predicted vector. The MSE is given by :

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y - \hat{Y}_i\right)^2 \tag{4.8}$$

A perfect prediction leads to an MSE equal to 0. An optimization algorithm aims to gradually decrease the MSE function towards its global minima. In practice, the RMSE (*Root Mean Squared Error*) is more used than the MSE to evaluate the performance of NN. One of the reasons is that RMSE is measured in the same units as variables in the compared dataset. This makes it easier to interpret the results and measure the performance of the model.

4.3.2. Mean Absolute Error (MAE)

As with the MSE, the MAE computes the mean of the absolute difference between two data sets:

$$MAE = \frac{1}{n}\left|Y_i - \hat{Y}_i\right| \tag{4.9}$$

Compared to MSE, MAE is less biased by outliers. As the MAE decreases, the NNs output gets closer to the target. However, it is less used as a cost function than the MSE because it is not differentiable at 0 which makes the gradient not smooth.

4.3.3. R-squared

R^2 is considered the standardized version of MSE. It computes the proportion between the sum squares regression (SSR) and the sum of squares total (SST).

$$R^2 = \frac{SSR}{SST} \tag{4.10}$$

$$= \frac{\sum_{i=1}^{n}\left(\hat{Y}_i - \bar{Y}\right)^2}{\sum_{i=1}^{n}\left(Y_i - \bar{Y}\right)^2} \tag{4.11}$$

Where $\bar{Y}$ is the mean of the target variables. It can also be written in terms of MSE:

$$R^2 = 1 - \frac{MSE}{Var(Y)} \tag{4.12}$$

$$= 1 - \frac{\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2}{\sum_{i=1}^{n}\left(Y_i - \bar{Y}\right)^2} \tag{4.13}$$

4.3.4. Adjusted R-squared

Adjusted R-squared is a slight modification of the R-squared error, it identifies the number of independent variables in a model and it is always less than or equal to the R-squared error.

$$R^2_{adj} = 1 - \left[\frac{(1 - R^2)(n - 1)}{n - k - 1}\right] \tag{4.14}$$

With n as the number of observations in the data and k as the number of independent variables.

4.3.5. Huber Loss

Mostly used in robust regression. Similarly to MAE, Huber loss is less sensitive to outliers than MSE but has the same convergence benefit as MSE.

$$\mathcal{L}(Y, \hat{Y}) = \begin{cases} \frac{1}{2}\left(Y - \hat{Y}\right)^2, & \text{if } |Y - \hat{Y}| < \delta \\ \delta\left(|Y - \hat{Y}| - \frac{1}{2}\delta\right) & \text{otherwise} \end{cases} \tag{4.15}$$

With δ the threshold controlling the point at which the function switches from a quadratic loss to a linear loss.

4.3.6. Cross-Entropy

Based on the idea of entropy, the cross-entropy loss measures the difference between two probability distributions. It is mostly used in Machine Learning for classification tasks, not to be confused with the KL divergence which computes the relative entropy between two probability distributions. An optimization algorithm using the Cross-Entropy loss function will aim to minimize the likelihood of incorrect predictions while increasing the likelihood of correct predictions.

$$\mathcal{L}_{CE}(x, y) = -\sum_{k=1}^{N} y_k \log(x_k) \tag{4.16}$$

4.4. Backpropagation

Three phases of training of NNs can be identified. (1) After receiving the input data, the NN uses forward propagation to extract features from the input data, layer by layer, until it produces an output. (2) From the output, a loss value is computed to measure the difference between the target and predictions. The resulting loss value is then propagated back to the network using a *gradient descent* [99] algorithm to improve the weight matrix

and bias vector.

Recall Equation 4.7, for a given layer weight matrix W and a bias vector b we can write the forward pass as:

$$h^{[1]} = W^{[1]}x + b^{[1]} \tag{4.17}$$
$$a^{[1]} = \phi^{[1]}(h^{[1]}) \tag{4.18}$$
$$h^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \tag{4.19}$$
$$a^{[2]} = \phi^{[2]}(h^{[2]}) \tag{4.20}$$
$$h^{[3]} = W^{[3]}a^{[2]} + b^{[3]} \tag{4.21}$$
$$\vdots \tag{4.22}$$
$$h^{[\ell-1]} = W^{[\ell-1]}a^{[\ell-2]} + b^{[\ell-1]} \tag{4.23}$$
$$a^{[\ell-1]} = \phi^{[\ell-1]}(h^{[\ell-1]}) \tag{4.24}$$
$$h^{[\ell]} = a^{[\ell-1]} \tag{4.25}$$

After computing the loss $\mathcal{L}$, the update at layer k with $1 \leq k < \ell$ is given as [100]:

$$W^{[k]} = W^{[k]} - \alpha\frac{\partial\mathcal{L}}{\partial W^{[k]}} \tag{4.26}$$
$$b^{[k]} = b^{[k]} - \alpha\frac{\partial\mathcal{L}}{\partial b^{[k]}} \tag{4.27}$$

$k < \ell$ because $h^{[\ell]}$ represents the output of the NN and there is no $W^{[\ell]}$. The learning rate α controls how far to move in the direction of the gradient[1]. Given a loss function $\mathcal{L}$, the gradient is an indicator showing in which direction (positive or negative) a parameter needs to be changed.

We use the *chain rule* to compute the gradient:

$$\frac{\partial\mathcal{L}]}{\partial W^{[k]}} = \frac{\partial\mathcal{L}}{\partial h^{[k]}}\frac{\partial h^{[k]}}{\partial W^{[k]}} \tag{4.28}$$

Using the differentiation the second fraction of the right hand in equation 4.28can be written as:

$$\frac{\partial h^{[k]}}{\partial W^{[k]}} = a^{[k-1]} \tag{4.29}$$

Hence we have:

$$\frac{\partial\mathcal{L}}{\partial W^{[k]}} = \frac{\partial L}{\partial h^{[k]}}a^{[k-1]} \tag{4.30}$$

[1]The direction of the steepest increase of the loss function.

We proceed similarly for the bias:

$$\frac{\partial L}{\partial b^{[k]}} = \frac{\partial L}{\partial h^{[k]}} \frac{\partial h^{[k]}}{\partial b^{[k]}} \tag{4.31}$$

$$\frac{\partial h^{[k]}}{\partial b^{[k]}} = 1 \tag{4.32}$$

$$\frac{\partial L}{\partial b^{[k]}} = \frac{\partial L}{\partial h^{[k]}} \tag{4.33}$$

Although easy to compute, the true gradient becomes computationally expensive when dealing with large data sets, this slows down the NNs training, especially with a large number of epochs[2]. An estimate of the gradient is thus used in the backpropagation process. Different methods referred to as *Optimisers* have been developed to estimate the gradient.

4.5. Gradient Descent Methods

There are three types of gradient descent methods.

4.5.1. Batch Gradient Descent

Let $J(\theta) = \mathbb{E}\left[\mathcal{L}(Y_i, \hat{Y}_i, \theta)\right]$ be the *risk function* associate with the parameter θ. $J(\theta)$ cannot be minimized directly as the distribution generating the data is unknown. By taking a finite training set of independent observations, an approximation $\hat{J}(\theta)$, known as the *empirical risk function*, can be computed and used as an alternative in the gradient descent process [101]:

$$J(\theta) \approx \hat{J}_L(\theta) \triangleq \frac{1}{L} \sum_{i=1}^{N} \mathcal{L}(Y_i, \hat{Y}_i, \theta) \tag{4.34}$$

with $\{Y_i\}$, $i \in \{1, \cdots, N\}$ a finite set of independent variables. Given a large training data set [102] showed the minimum of $\hat{J}_L(\theta)$ provides a good estimate of the minimum of $J(\theta)$.

To compute the minimum of the empirical risk $\hat{J}(\theta)$ the *batch gradient* algorithm,(also referred as the *vanilla gradient*) proceed by computing successive estimates θ_t of the optimal parameter using the following update rule:

$$\theta_{t+1} = \theta_t - \alpha_t \nabla_\theta \hat{J}_L(\theta_t) \tag{4.35}$$

$$= \theta_t - \alpha_t \frac{1}{L} \sum_{i=1}^{N} \nabla_\theta \mathcal{L}(Y_i, \hat{Y}_i, \theta_t) \tag{4.36}$$

[2]epochs: One-time sequence of forward pass and backpropagation in an NNs training. To reach optimal parameters NNs train in many epochs.

While providing efficiency, it can become computationally expensive as it goes through the entire training data set to compute the average. It requires enough memory allocation to store the data set and can become intractable if the stored data does not fit the memory space.

4.5.2. Stochastic Gradient Descent (SGD)

To address the memory allocation and computation issues of batch gradient descent, SGD, also referred to as *Online Gradient Descent* [101], drops the idea of averaging. At each iteration t a random example Y_t is picked from the training set, and the parameters update is done according to the following rule:

$$\theta_{t+1} = \theta_t - \alpha_t \nabla_\theta \mathcal{L}(Y_t, \hat{Y}_t, \theta_t) \tag{4.37}$$

While reducing the computation, the convergence to the minimum can be noisy resulting in high variance. The noisy convergence helps sometimes to avoid being stuck in a local minimum and finding a global one.

4.5.3. Mini-batch Gradient Descent

It is a mixture of batch gradient descent and SGD. At each training epoch, t the data set is split into mini-batches of size B. Hence, for each mini-batch, a parameters update is performed. This approach has the benefit of combining the advantage of both previous methods.

4.5.4. Challenges with gradient descent methods

Gradient descent methods are among the most effective approaches used to solve optimization problems. However, these are some challenges that need to be addressed:

1. Local Minima and Saddle points

Especially for non-convex problems, gradient descent methods can struggle to find the global minimum. It can instead get stuck in a local minimum or a saddle point.

2. Vanishing gradients

Vanishing gradients is a situation where the gradient gets smaller during backpropagation. This has the effect of making layers far from the output layer learn slower compared to layers close to the output layer.

3. Exploding gradient

A large gradient implies a large update on the NN weights, this leads to unstable training. Techniques such as *dimensionality reduction* can help to address this problem.

4.6. Optimizers

In backpropagation, optimizers are in charge of updating the model parameters (weights and biases) in a way that minimizes the loss function during training. We provide an overview of commonly used optimizers in Deep Learning.

4.6.1. Momentum

Momentum [103] is a method that speeds up SGD in the relevant direction [104]. The update rule is defined as:

$$v_t = \gamma v_{t-1} + \alpha \nabla_\theta J(\theta) \tag{4.38}$$

$$\theta = \theta - v_t \tag{4.39}$$

where θ represents the model parameters, $\nabla_\theta J(\theta)$ is the gradient of the loss function with respect to the parameters, α is the learning rate, γ is the momentum parameter, and v_t is the velocity at time step t. Equation 4.38 computes the velocity as a combination of the previous velocity and the current gradient, and equation 4.39 updates the parameters by subtracting the computed velocity.

4.6.2. Adaptive Gradient Algorithm (Adagrad)

Differently to SGD which needs a supervised adjustment of the learning rate (or step size parameter), Adagrad [105] has the special property of automatically adjusting the learning rate by dynamically incorporating knowledge of the geometry of the observed data in earlier training phases. Frequently updated parameters receive small updates and infrequently updated parameters receive large updates [104]. It computes the SGD gradient update for a particular parameter θ_i as follows:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,ii} + \epsilon}} \nabla_{\theta_t} J(\theta_{t,i}) \tag{4.40}$$

With $G_t \in \mathbb{R}^{d \times d}$ a diagonal matrix where each element (i, i) is the sum of the squares of the gradients w.r.t. θ up to time t and ϵ is a security parameter that avoids the division by zero error. One of the disadvantages of Adagrad is that over time the cumulative gradients

grow large making the learning term small. Hence, the gradient has less effect in the update.

4.6.3. Root mean squared propagation (RMSProp)

The idea behind RSMProp [106] is to address the vanishing learning rates problem of Adagrad. Let $g(t) = \nabla_{\theta_t} J(\theta)$, the update rule is than given as follow:

$$\mathbb{E}[g^2]_t = \alpha \mathbb{E}[g^2]_{t-1} + (1-\alpha) g_t^2 \tag{4.41}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} g_t \tag{4.42}$$

Equation 4.41 computes the exponentially weighted moving average (EWMA) of the squared gradient up to the current time step t, denoted as $\mathbb{E}[g^2]_t$. α is the smoothing factor controlling the rate at which the EWMA is updated, typically set to a value between 0.9 and 0.99. Equation 4.42 updates the weightθ_t at time step t based on the EWMA of the squared gradients up to time step t and the gradient at time step t. ϵ is a small positive constant added for numerical stability.

4..6.4. Adaptive moment estimation (Adam)

Introduced by Kingma and Ba [107], Adam is a combination of previous improvements. Similarly to RMSProp, it keeps in memory an exponentially decaying average of past squared gradients v_t, and similarly to Momentum, it keeps an exponentially decaying average of past gradients m_t [104].

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{4.43}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{4.44}$$

Equation 4.43 computes the first moment m_t which is the EWMA of the gradient up to the time step t. β_1 represents the smoothing factor controlling the rate at which the EWMA is updated, typically set to a value of 0.9. Equation 4.44 computes the second moment v_t which is an EWMA of the squared gradient up to the time step t. β_2 is the smoothing factor controlling the rate at which the EWMA is updated, typically set to a value of 0.999. To avoid biases in estimations of the two moments, the author suggested computations of the bias-corrected version of the first and second moment estimates as

follows:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{4.45}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{4.46}$$

Then the update becomes is given as follow:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{4.47}$$

ϵ is a small positive constant added for numerical stability. In practice, Adam is the most used optimizer among adaptive learning methods. It converges faster and can be less sensitive to hyperparameter choices, such as the learning rate.

4.7. Summary

In this chapter, we discussed Deep Learning, a field that uses the notion of artificial neural networks to solve different machine learning tasks. We have seen that DNNs are architectures that process given input data from an input layer of artificial neurons to an output layer. We mentioned that DNNs use the forward pass technique across sequential layers to proceed with feature extraction of input data and use the backpropagation technique to optimize the parameter weight matrices for better feature extraction in future forward passes.

We saw that backpropagation refers to the computation of the gradient of the output layer. And we mentioned that SGD is the most used gradient-based algorithm in the DNNs parameters optimization process. The SGD has been applied in many forms called Optimizers such as the Adam algorithm.

In the next chapter, we discuss how Deep Learning has been applied as a powerful technique to improve classical RL algorithms.

Chapter 5

Deep Reinforcement Learning

Deep Reinforcement Learning (Deep RL) is a combination of Reinforcement Learning and Deep Learning. Tabular methods require the storage of states and actions to compute the value function. Their weakness arises quickly when dealing with large or continuous action and/or state spaces. This is where Deep RL comes to the rescue. Deep NNs structures are incorporated into RL algorithms to approximate value functions or policies using function approximations. Let θ be the parameter of the weight of the NNs. Given a state s and an action a, let $V(s;\theta)$, $Q(s,a;\theta)$ and $\pi(a|s;\theta)$ be respectively approximations of the state-value $v(s)$, action value $q(a,s)$ and policy $\pi(s,a)$ according to θ. A Deep RL algorithm will perform optimization over θ to generate θ^* that will bring the best estimate of $v^*(s)$, $q^*(s,a)$ or $\theta^*(s,a)$.

From the success of DQN [51], A lot of DRL methods have been developed in the literature, each bringing powerful results in solving complex tasks [108, 109]. Deep RL methods can be split into 2 main categories: value-based methods which aim to approximate the value function; and policy gradient methods which aim to approximate the policy directly. In this chapter, we discuss general concepts of Deep RL to familiarize readers with our project.

5.1. Value Based algorithms

In value-based methods the NNs receive a state s as input, learn features through sequential layers, and output the value function (action-value most of the time) of all possible actions paired with s.

5.1.1. Deep Q-Networks

One of the most important breakthroughs in RL arose when Mnih et al [51] presented a Convolutional Neural Network (CNN) [110] based algorithm that learned to play Atari games [45] at a human level of performance using the Q-learning framework. They referred

to the algorithm as Deep Q Networks (DQN)[1]. DQN is considered the foundation of Value-Based and Off-policy D RL algorithms. It was the first successful application of deep learning to DRL, and it demonstrated the potential of using deep neural networks to approximate Q-functions in complex environments.

The DQN was introduced to address the Q-learning weakness of dealing with large state spaces and still benefits from the feature of being model-free. As it follows the Q-learning framework, DQN trains a model to approximate the optimal action-value function $Q^*(s,a)$ by performing a gradient descent on the objective function defined as:

$$J(\theta_i) = \mathop{\mathbb{E}}_{s,a\sim\rho(.)} \left[(Y_i - Q(s,a;\theta_i))^2\right] \tag{5.1}$$

$$Y_i = \mathop{\mathbb{E}}_{\substack{s,a\sim\rho(.)\\ s'\sim\mathcal{E}}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})\right] \tag{5.2}$$

Where Y_i is the target at iteration i and $\rho(s,a)$ is a probability distribution over states and actions, referred to as *behavior distribution* [51]. Hence, the gradient of the objective function can be written as:

$$\nabla_{\theta_i} J(\theta_i) = \mathop{\mathbb{E}}_{\substack{s,a\sim\rho(.)\\ s'\sim\mathcal{E}}} \left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right) \nabla_{\theta_i} Q(s,a;\theta_i)\right] \tag{5.3}$$

where $\mathcal{E}$[2] is an emulator that serves for experience sampling. We should also point out that the behavior distribution follows the $\epsilon-$greedy strategy as with Q-learning to encourage sufficient exploration.

[111] stated that mixing off-policy, function approximations (NNs), and bootstrapping have some disadvantages. Instability and divergence may occur during the training due to correlations between successive observations; correlations between the Q value and the target; small updates of the value function can deviate from the optimal policy and modify the data distribution. To improve stability during training DQN uses two techniques:

1. Experience Replay

During the episode, a tuple $e_t = (S_t, A_t, R_t, S_{t+1})$ of the current experience is stored in a memory space, referred to as a *replay buffer*. During the policy update, random samples are taken from the buffer to perform parameter updates as an experience replay. As the buffer keeps storing experience through episodes a given sample can be used more than once. Random sampling solves the correlation issue in sequential observations and smooths data distribution changes.

[1] We should emphasize that the CNN used by the authors is not a part of the DQN algorithm.

[2] The emulator serves as a simulator of the environment in which the agent operates, allowing the agent to interact with the environment and learn from its experiences.

2. Off-Policy Training

A DQN uses a separate target network which is a copy of the main network to sample TD targets. Parameters of the target network are kept separately and are updated periodically [3]. This has the benefit to reduce the correlation between the Q value and the target of the TD update. Hence, the Q-learning loss function can be written as:

$$L(\theta) = \left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2 \tag{5.4}$$

with θ_i^- the parameter of the target network at iteration i.

5.1.2. Double Q-learning

Introduced by Van Hasselt et al [112]. The Double deep Q-network(D-DQN) aims to address the issue of over-optimistic estimation of the Q value in both Q-learning and DQN [111] due to the *max* operator in the TD target. Van Hasselt et al [112] proposed to replace the formal TD target in equation 5.2 by:

$$Y_t^{D-DQN} = r + \gamma\, Q\left(s', \arg\max_{a'} Q(s', a'; \theta_t); \theta^-\right) \tag{5.5}$$

That the online network is used to greedily evaluate the policy while the target network is used to evaluate its value function.

5.1.3. Prioritized Experience Replay

The vanilla DQN with experience replay samples random experiences regardless of their importance. In prioritized experience replay [113] important transitions are replayed more frequently, which leads to more efficient learning. Important transitions are those with high expected learning progress measured by the TD error.

5.1.4. Dueling Networks

Figure 5.1 illustrates a Dueling Networks, where two architectures are set up to generate the value $V(s)$ of a given state s and the associated advantage function $A(s, a)$. Where the advantage function measures how much selecting the action a was a good or bad decision in state s. Both architectures are combined to generate the action-value $Q(s, a)$ [114].

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \max_{a'} A(s, a'; \theta, \alpha)\right) \tag{5.6}$$

[3]The target network parameters are kept fixed during the training. They only get updated after every C iteration, with C a constant value.

Where α and β are parameters used to control the behavior of the two fully connected layers[4] that compute the advantage and value function in the dueling network. The author suggested the following update:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}A(s,a';\theta,\alpha\right) \tag{5.7}$$

where the max operator in equation 5.6 is replaced by the average. The author states that the Dueling architecture mixed with prioritized experience replay brought improvements in both DQN and D-DQN.

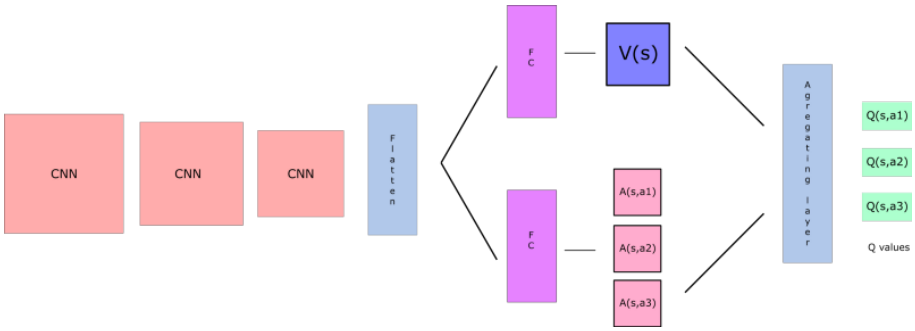


Figure 5.1: Dueling Networks architecture. Illustration of Dueling Networks [115]

5.2. Policy Gradient Algorithms

So far we have focused on algorithms that approximate the action-value functions and perform action selections based on those estimations. In this section, we discuss methods that directly learn the policy and select actions by doing sampling according to the learned policy, regardless of the value function.

Deep RL value-based methods have addressed the memory allocation issue of standard value-based algorithms when dealing with environments with large state space. However, to select the right action, Deep RL value-based algorithms perform a full scan over the value function to pick the action with the maximum value, this becomes computationally expensive while dealing with large action spaces. Policy gradient methods [116], in contrast, are faster by just doing a quick random selection over the action space. Table 5.1 gives a comparison between the two approaches.

5.2.1. Policy Gradient

Let $\pi(a|s;\theta) = \mathbf{P}\left[A_t = a|S_t = s, \theta_t = \theta\right]$ be the policy parameterized by θ [78]. The policy gradient methods optimize θ to maximize the objective function. Different from value-based

[4]In the Dueling Network, the CNN is followed by two streams of FC layers computing the advantage and value functions

Table 5.1: Key differences between Value-Based and Policy-Based RL Methods

RL Methods	
Value-Based	**Policy Based**
Learn the state or state-action value function	Learn the stochastic probability distribution that maps state to action
Select action greedily	Sample an action in the action space
Exploration is not mandatory	Exploration is induced due to the random selection of actions
Slow to compute the greedy action in large action spaces	Fast action selection in large action spaces

methods which aim to minimize the loss via gradient descent, policy gradient methods perform gradient ascent to reach the maximum reward.

Let $J(\theta)$ be the objective function representing the expected return. We have the following expression [117]:

$$J(\theta) = \sum_{s\in\mathcal{S}} d^{\pi}(s)V_{\pi}(s) \tag{5.8}$$

$$= \sum_{s\in\mathcal{S}} d^{\pi}(s) \sum_{a\in\mathcal{A}} \pi(a|s;\theta)Q_{\pi}(s,a) \tag{5.9}$$

With $d^{\pi}(s) = \lim_{t\to\infty} \mathbf{P}\left[s_t = s|s_0, \pi_\theta\right]$ is the probability of being in state s when starting from s_0 and following π_θ for t steps.

According to the *policy gradient* theorem [116], the gradient of $J(\theta)$ can be expressed as:

$$\nabla J(\theta) \propto \sum_{s} d^{\pi}(s) \sum_{a} Q_{\pi}(s,a)\nabla_{\theta}\pi(a|s;\theta) \tag{5.10}$$

Equation 5.10 is the foundation of most policy gradient algorithms. We give an overview of some in the following section.

5.2.2. REINFORCE

Referred as *Monte-Carlo* policy gradient, REINFORCE [118] is the vanilla policy gradient algorithm using function approximation. It uses episodic historical transitions to compute an estimate of the return following the Monte-Carlo technique. Recall the stochastic gradient algorithm 4.5, the gradient of the expected objective function is expressed as

follows [78]:

$$\nabla J(\theta) = \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla_\theta \pi(a|S_t;\theta) \right] \tag{5.11}$$

$$= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t;\theta) q_\pi(S_t, a) \frac{\nabla_\theta \pi(a|S_t;\theta)}{\pi(a|S_t;\theta)} \right] \tag{5.12}$$

$$= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \nabla_\theta \ln \pi(A_t|S_t;\theta) \right] \qquad \text{(replacing } a \text{ by the sample} A_t \sim \pi) \tag{5.13}$$

$$= \mathbb{E}_\pi \left[G_t \, \nabla_\theta \ln \pi(A_t|S_t;\theta) \right], \qquad \text{because } q_\pi(S_t, A_t) = \mathbb{E}\left[G_t | S_t, A_t \right] \tag{5.14}$$

Hence we have the following update as gradient *ascent* algorithm:

$$\theta_{t+1} \doteq \theta_t + \alpha \, G_t \, \nabla_\theta \ln \pi(A_t|S_t;\theta) \tag{5.15}$$

The convergence of the algorithm depends on several factors, including the choice of the learning rate, the variance of the gradient estimates, and the properties of the objective function being optimized. After REINFORCE many variants have been developed in the literature to improve the efficiency of learning.

5.2.3. Actor-Critic

Although standard policy gradient methods refer to the policy for actions selection, they still rely on the value function to learn the policy parameter [78]. In Actor-Critic based algorithms [119] the value function plays a more important role than just being a baseline for the policy parameters update. The Actor-Critic consists of two NN models:

- **Actor:** The actor is the model in charge of performing action selection according to the current policy π and updating the policy parameter θ based on the critic feedback.
- **Critic:** The Critic is in charge of computing the value function $Q(s, a)$ or $V(s)$ and performing the update of the model parameters ω by using bootstrapping to reduce variance and speed up the learning.

In the *One-Step* actor-critic the full return in REINFORCE is replaced by 1 step return [78]:

$$\theta_{t+1} \doteq \theta_t + \alpha \, \delta_t \, \nabla_\theta \ln \pi(A_t|S_t;\theta) \tag{5.16}$$

$$\delta_t = \underbrace{R_{t+1} + \gamma \hat{v}(S_{t+1}, \omega)}_{G_{t:t+1}} - \hat{v}(S_t, \omega) \tag{5.17}$$

The logic behind the actor-critic method is that the actor performs an action and the critic will tell it how good or bad was the selected action, then the actor will aim to optimize his behavior based on the critic's feedback.

Since its introduction, successive policy gradient methods are mostly actor-critic based.

5.2.4. Trust Region Policy Optimization (TRPO)

Introduced in [120], TPRO is a REINFORCE-based algorithm that aims to improve training stability by avoiding updates that move the new policy too far from the previous policy in one iteration. For example, if the learning rate is too large, a small gradient can considerably affect the policy update. To address this problem, the policy update is subjected to a KL divergence [121] constraint at each iteration. Hence, the gradient problem becomes:

$$\begin{aligned} &\max_{\theta} \mathbb{E}_{s\sim\rho_{\theta_{old}}, a\sim q}\left[\frac{\pi_\theta(a|s)}{q(s,a)}Q_{old}(s,a)\right] \\ &s.t. \qquad \mathbb{E}_{s\sim\rho_{\theta_{old}}}\left[KL\left(\pi_{\theta_{old}}(.|s)||\pi_\theta(.|s)\right)\right] \leq \delta \end{aligned} \tag{5.18}$$

where q is denoting the sampling distribution and δ is a parameter defining the bounds of the trust region. In some case $q(a|s)$ is replaced by $\pi_{old}(a|s)$. Despite how clever the approach is, the max operator implies solving a maximization problem at each iteration which can be problematic [122].

5.2.5. Proximal Policy Optimization (PPO)

[56] introduced the PPO algorithm, which can alternate between data sampling and optimization of a surrogate objective function using SGD. PPO is the successor of TRPO, it follows the same logic of avoiding updates that deviate too much from the current policy. However, PPO is more flexible to implement compared to TRPO. To the best of our knowledge, the Actor-Critic PPO is the state-of-the-art policy gradient algorithm at the moment.

[56] proposed changing the constrained optimization problem of TRPO with an unconstrained one, using a penalty in place of the TRPO constraint.

$$\max_{\theta} \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t(s,a) - \beta KL\left[\pi_{\theta_{old}}(.|s_t)||\pi_\theta(.|s_t)\right]\right] \tag{5.19}$$

where $r_t(\theta)$ represents the probability ration:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \tag{5.20}$$

The choice of β can be challenging as the same value might perform differently in different problems. To apply the SGD additional techniques have been proposed:

Clipped Surrogate Objective

The surrogate objective function in TRPO is defined as [56]:

$$J^{CPI} = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t(s,a) \right] \tag{5.21}$$

To avoid update explosion, a penalty is set to move $r_t(\theta)$ away from 1. Hence the objective function is written as follows:

$$J^{CLIP} = \hat{\mathbb{E}}_t \left[min \left(r_t(\theta) \hat{A}_t(s,a), clip(r_t(\theta), 1-\epsilon, 1+\epsilon) \hat{A}_t(s,a) \right) \right] \tag{5.22}$$

Where the function $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)$ make sure that the ratio stays in $[1-\epsilon, 1+\epsilon]$. This removes the motivation of changing considerably the policy for one update.

Adaptive KL Penalty Coefficient

The second technique is the Kullback–Leibler(KL) Penalty where a penalty factorized by an adaptive coefficient is added to the KL divergence.

$$J^{KLPEN}(\theta) = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t(s,a) - \beta KL \left[\pi_{\theta_{old}}(.|s_t),\, \pi_\theta(.|s_t) \right] \right] \tag{5.23}$$

Where the coefficient β is updated as following this procedure:

Algorithm 5.2: KL

Compute $d = \hat{\mathbb{E}}_t \left[KL \left[\pi_{\theta_{old}}(.|s_t),\, \pi_\theta(.|s_t) \right] \right]$
if $d < d_{targ}/1.5$:
    $\beta \leftarrow \beta/2$
if $d > d_{targ} \times 1.5$:
    $\beta \leftarrow \beta \times 2$

According to the PPO, paper [56], the clipped version of PPO performs better than the KL Penalty version.

Despite being the state-of-the-art policy gradient algorithm in tasks involving continuous state space and/or continuous action space, there are still some challenges that need to be addressed [123]:

- Instability on training when the reward vanishes outside bounded support
- Getting stuck in suboptimal policy on discrete action spaces.

- Sensitivity to initialization: Every time a model is initialized for training, it receives a random matrix of parameters. Different randomizations don't necessarily provide the same training performance on the same task.

5.3. Summary

This chapter aimed to introduce readers to the paradigm of Deep RL, a novel approach that uses the power of Deep NNs to solve RL problems. We have seen that Deep RL has been first introduced to address the memory allocation issue of Tabular Methods when dealing with environments with large state space. We discussed that techniques of a random sampling of historical experience are one of the central keys that guarantee convergence.

Furthermore, we discussed the difference between value-based methods and policy gradient methods in Deep RL. Despite the amazing successes in many benchmark tasks, value-based methods show weakness when the action space is large or continuous. Policy gradient methods benefit from the feature of computing directly the policy regardless of the structure of the action space. This makes them suitable for tasks with large action spaces.

We also introduced some policy gradient algorithms, specifically the PPO algorithm, which is an algorithm that addresses the divergence issue, by constraining the gradient update such that it doesn't move the new policy too far from the current policy. However, the approach generates a slow convergence due to the small update at each iteration.

We mentioned that, despite being the state-of-the-art policy gradient algorithm, the PPO has some weaknesses such as the sensitivity to the initialization of the training model. We used the PPO as the benchmark algorithm in our project, because of its high performance on the testbed task used in the project, and in Chapter 8, it will be demonstrated that the algorithm can achieve remarkable performance on a specific task. However, it can also fail to perform well on the same task when utilizing different model initializations.

Chapter 6

Incremental Reinforcement Learning

This chapter aims to discuss in more detail the notion of Incremental RL [14, 15] we introduced earlier in section 2.1. Specifically, the discussion focuses on solving a target task using incremental MDPs. Furthermore, we discuss the notion of initial task, performance threshold, and incremental step, which influence the incremental learning process.

Humans can incrementally master difficult levels of a given task by starting simple. An individual who doesn't train in weight lifting has a higher probability to fail to lift $180\,kg$ compared to another one who can already lift $150\,kg$ with ease. We aimed to apply the same paradigm with the PPO algorithm to incrementally solve an environment that the algorithm failed to solve from scratch in both single and multi-agent scenarios.

6.1. Definition

Let $\{\mathcal{T}_i\}_{2\leq i\leq n}$[1] be a family of tasks where each task $\mathcal{T}_i = \langle \mathcal{S}_i, \mathcal{A}_i, \mathcal{P}_i, \mathcal{R}_i \rangle$ is an MDP representing a classical RL problem with a given objective function. The basic idea of a Transfer Learning [11] method is to train an agent that can learn a family of optimal policies $\{\pi_i^*\}_{2\leq i\leq n}$ for each respective tasks in $\{\mathcal{T}_i\}_{2\leq i\leq n}$ by leveraging knowledge across tasks to learn faster and efficiently.

6.2. Previous work

Meta-RL [13, 34] formalises the transfer learning problem as a distribution $p(\mathcal{T})$ over a family of MDPs $\{\mathcal{T}_i\}_{2\leq i\leq n}$ where the objective function is defined as the average of tasks objective functions under the task distribution $p(\mathcal{T})$:

$$J(\theta) = \mathbb{E}_{\mathcal{T}\sim p}\left[J_{\mathcal{T}}(\theta)\right] \tag{6.1}$$

[1] To perform transfer learning, we need at least two tasks

with $J_{\mathcal{T}}$ the objective function of a particular task. The agent in Meta-RL will learn parameters θ^* of the optimal policy π^* that maximize $J(\theta)$. [34] define $\pi(a|s,z)$ as a task conditioned policy, with z an ID denoting the current task. The policy π^* would quickly adapt to new test tasks that were previously unseen.

While Meta-RL learns a distribution policy that aims to maximize the average expected returns over the distribution of tasks, Incremental RL learns a set of sequential optimal policies of a given set of tasks. The environment dynamic $\mathcal{D}$ is split into a set of tasks $\mathcal{D} = [\mathcal{T}_1, \cdots, \mathcal{T}_n]$, where each task $\mathcal{T}_i$ represents a stationary MDP. The idea is to make the agent sequentially learns the optimal policy parameters $\theta_1^*, \cdots, \theta_{i-1}^*$ of respective tasks $\mathcal{T}_1, \cdots, \mathcal{T}_{i-1}$ and leverage them as baselines to facilitate the learning of the optimal policy parameter θ_i^* of the task $\mathcal{T}_i$. Despite the potential success, this is at risk of forgetting knowledge acquired on prior solved tasks during sequential learning. Life-Long Incremental RL [36] addresses the catastrophic forgetting issue [18] by making sequential learning a lifetime learning process. In other words, the agents will generate sequences of optimal parameters $(\theta_1^*, \cdots, \theta_n^*)$ and a library of pairwise parameters $\left(\theta^{(\infty)}, \vartheta^{(\infty)}\right)$ is set to store all pairs composed of the policy θ of the current task and a parameterization ϑ of the current environment.

As in Life-Long Incremental RL, we used a training paradigm that helps agents to maintain prior knowledge as lifetime learning. However, the structure of the task we are aiming to solve doesn't require storage in memory of prior policies or sub-task parametrization. We give further details in the following section.

6.3. Problem Formalization

Let $\mathcal{T}$ represent the MDP of a challenging task to solve from scratch. And let us define $\mathcal{D} = [\mathcal{T}_1, \cdots, \mathcal{T}_n]$ a subset of the power set of $\mathcal{T}$ such that:

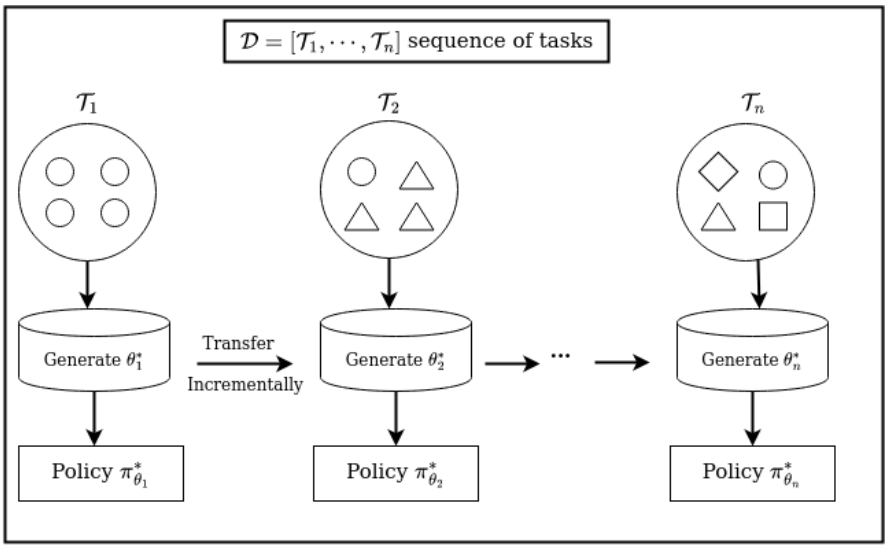
$$\begin{cases} \mathcal{T}_{i-1} \subset \mathcal{T}_i \\ \mathcal{T}_n = \mathcal{T} \end{cases} \tag{6.2}$$

where each $\mathcal{T}_i$ represents a stationary MDP. As shown in Figure 6.1, this simply means that the MDP of a previous task is included in the MDP of the next one. The goal is then to train the agent to generate a sequence $\{\theta_1^*, \cdots, \theta_n^*\}$ of optimal parameters for respective tasks in $\mathcal{D}$. We use the expression *increment* every time the training agent shift from a task $\mathcal{T}_i$ to a task $\mathcal{T}_{i+1}$.

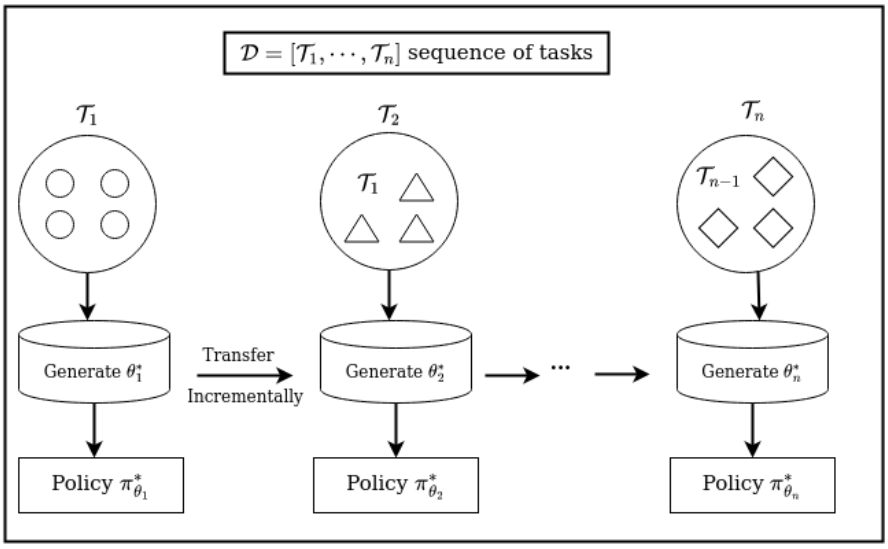
Recall section 3.6 where we discussed optimality. Following equation 3.25, the sequence of optimal policies generated over $\mathcal{D}$ follows the structure $\forall i < j$:

$$\begin{cases} \pi^*_{\theta_i} \approx \pi^*_{\theta_j} \text{ in } \mathcal{T}_i \\ \pi^*_{\theta_i} \leq \pi^*_{\theta_j} \text{ in } \mathcal{T}_j \end{cases} \tag{6.3}$$

This means that a policy that performs optimally in a larger MDP can perform optimally in a smaller MDP of the same task.



(a) Incremental RL



(b) Incremental RL with Incremental MDP

Figure 6.1: (a) Simple Incremental RL diagram. $\mathcal{D}$ represents the subdivision of tasks in a sequential manner. Successive tasks share common similarity structures. (b) Incremental RL with Incremental MDP. Each task different from the initial task contains copies of all previous task MDPs.

The illustration in Figure 6.1 (b) shows that when the agent moves to a new task, it will still experience MDPs of previously solved tasks. This approach has the benefit to allow adaptation on previously unseen transitions while preventing catastrophic forgetting [18]

by securing skills acquired on the previous experience. Differently to [36], there is no need for a library to store previously learned policy parameters and task structures. The new optimal policy generalizes over all previous MDPs as shown in equation 6.3. If the agent gets evaluated in a previously solved sub-task, it will still be able to perform optimally even though the policy parameters have changed.

In simple words, the process consists of training an agent in a small portion of the main task MDP until it generates an initial optimal policy π^*_{init}. Then, make the task more complex by adding another small portion of the main MDP such that the agent will be experiencing new transitions along with previously experienced transitions.

The knowledge to transfer in our project is the *policy* function. However, there is still the need to clarify, in our context, some of the fundamental questions [11] of Transfer Learning that we mentioned earlier in section 2.1 such as: When should the transfer occur, and how to operate with the transfer? The answer to those questions is discussed in the following section, where we introduce the notion of performance threshold to know when to transfer, and incremental step to control the number of transitions we aim to add to the current MDP.

6.4. Threshold Policy

In a situation such as academic training, Humans use incremental learning to gradually master a given domain of interest. But to migrate to an upper level of training is subject to success in the current level of training. The required success is measured by an evaluation score that the learner must reach to be allowed to migrate to an upper level of training. However, Humans don't always reach 100% of the evaluation score. Hence, often in practice, a *threshold* value is set to help decide if a learner is ready to migrate to another training level. For example, in academic training, a threshold score of 50% can be enough to allow a learner to move to an upper level of training. Referring to this, we defined our incremental problem as a threshold learning problem, where perfection on the current task is not a requirement before migrating the learner to a more complex task.

Let δ be a threshold parameter. The process consists of training the agent to generate a policy π^δ_i on the task $\mathcal{T}_{i<n}$, with $\pi^\delta_i \leq \pi^*_i$, from which it can reach a score of δ on average during a test on $\mathcal{T}_{i<n}$. Then, the threshold policy π^δ_i is used as an initialization to generate π^δ_{i+1} on the task $\mathcal{T}_{i+1}$. As shown in the incremental algorithm 6.3, once in the final task $\mathcal{T}_n$ the agent will aim to directly generate the optimal policy π^*_n. In other words, using a threshold parameter δ, the incremental learning process on $\mathcal{D}$ consists of generating a sequence of policies $\left\{\pi^\delta_1, \cdots, \pi^\delta_{n-1}, \pi^*_n\right\}$. As the optimal policy of the final and main task, π^*_n represents an optimal policy for all sub-tasks in $\mathcal{D}$.

Despite using a threshold parameter on the sequence of tasks seems clever, there is still the fact that to select the appropriate parameter to use as the performance indicator. The *return* G could be the appropriate parameter to use as the threshold indicator [2]. By tracking the progress of G during the training, the training environment gets larger every time $G > \delta$. However, in our particular situation with the Self-Play Learning [29], referring to the reward as a performance indicator might not be appropriate. During Self-Play Learning in a zero-sum game, both opponents' expertise can be similar, and the average return for both sides will be fluctuating around 0 as they are supposed to tie most of the time. In episodic task scenarios, the average episode length can be used as an alternative performance indicator if the reward is not an appropriate choice. During the Self-Play training, if the average episode length starts growing, the learning agent has started understanding the task. Hence, it is more appropriate to track the average episode length per episode than the average return to proceed with the increment. This is only possible in scenarios with a conditional terminal state. For example, an episode may end when the agent loses its life, even if it is before the maximum length an episode can have during the task.

Algorithm 6.3: Incremental Learning with Threshold Score

Input $\mathcal{D} = [\mathcal{T}_1, \cdots, \mathcal{T}_n]$, $n \geq 0$: Task sequences
Input Model: The training agent
Input δ : Performance Threshold
Input $t > 0$: Training Horizon
For $i \in [1, \cdots, n-1]$:
    $\mathcal{T} = \mathcal{T}_i$: The current task
    $score = 0$
    While $score \leq \delta$:
        - Train agent on $\mathcal{T}$ for t timesteps
        - Compute the evaluation $score$
    End while
End for
Continue with the formal training on $\mathcal{T}_n$
Save Model

Assuming that we have been able to choose an appropriate performance indicator, we also need to assign the appropriate threshold value. The illustration of the Human learning process, tells us that perfection during training is not always a requirement before advancing to the next level of training. However, it does not indicate how humans decide on the threshold of tolerance because before proceeding with the increment, the algorithm must ensure that the agent has enough experience to deal with the next task. Also, there is

[2] Recall that the main goal of RL agents is to maximize the expected return

a need to control how difficult the new task should be compared to the previous one. This tells us that the success of Incremental Learning relies on 3 major elements: the initial task $\mathcal{T}_{init}$ from where the training is initiated, the threshold score δ that controls when to increment, and the incremental step η which controls how difficult the new training task should be.

6.4.1. The Initial Task

Let $\mathcal{T}_0$ be the smallest possible MDP extracted from the main task MDP from which the agent can initiate the training. $\mathcal{T}_0$ is set such that the agent might receive a positive reward frequently to adapt quickly. A practical example could be to select the portion of the main task MDP where the sparsity of the main task is reduced as much as possible. Even though the ideal would be to start the training on $\mathcal{T}_0$, it is not always mandatory. Let us consider a task $\mathcal{T}_\ell$, with $0 < \ell < n$. The agent can still initiate its training in $\mathcal{T}_\ell$ which includes the $\mathcal{T}_0$ MDP and be able to generate the initial policy for the incremental learning process. This is possible if there is not a big gap between $\mathcal{T}_\ell$ and $\mathcal{T}_0$, otherwise, the agent might not be able to adapt and generate the first initial policy. Starting the training with $\mathcal{T}_\ell$ may improve the training time.

6.4.2. Performance Threshold

We discussed earlier the performance threshold parameter δ which helps control when to increment. After selecting the appropriate initial task and the appropriate performance indicator parameter, a threshold value should be assigned to δ such that the algorithm will know when to increment. We assume that a low value will generate early migrations which might lead to a slow adaptation or a non-adaptation. We also assume that a high threshold value will generate late migration, as the agent will spend too much time in each task to reach the threshold before the increment. We assume this does not optimize the overall training time. We proceed by empirical experiments to determine what could be the right performance threshold in our particular case. We discuss more this in chapter 8.

6.4.3. Incremental Step

The incremental step η is a parameter representing how difficult we would like to make the next task compared to the previous one. A large increment might result in a slow adaptation or non-adaptation as the new task MDP will decrease the opportunity of experiencing solved transitions. This can also lead to catastrophic forgetting [18] of prior skills on previously solved transitions.

Adding η to the increment training raises the question of whether there is an influence between the threshold and the increment step. What could be the effect on the training

when using a high threshold and a low step, or the reverse? How to measure the adaptability rate of the agent using the provided threshold and increment step? We discuss in the next section one of the ways these questions can be answered.

6.5. Training Process

In this project, we proceed with empirical experiments to study the influence between the performance threshold and the incremental step. The approach consists of freezing one of the two parameters, by assigning a constant value, and running different training with different values of the other parameter. Then Repeating the same process with a different other value of the frozen parameter. This helps to know in general how both parameters influence each other.

We measure the performance of a given couple (δ, η) by the average training time it takes to increment up to the final task and start playing optimally on the final task.

Let f be the average training time to training completely the agent in the environment. f can be expressed as:

$$f = \mathbb{E}\left[T = t | \Delta = \delta, H = \eta\right] \tag{6.4}$$

Notice that we refer to the average because the training time is also influenced by the initial randomization. The challenge is to find the combination (δ^*, η^*) that optimizes f in average. Intuitively we may think of the average of both parameters as the appropriate combination. However, We cannot tell until we observe the outcome of experiments for our particular case.

6.6. Summary

In this chapter, we discussed in more detail the notion of Incremental RL [14, 15]. We have seen that Incremental RL is an application of Curriculum Learning [12] in RL where the sequence of tasks are different levels of the same task from the simplest to the most difficult. We focused on the particular case where MDPs of sequential tasks are drawn from the power set of the main task MDP, and where each task MDP contains MDPs of its predecessors. We mentioned that this case has the benefits of skipping the storage of prior policies and task parametrization used in Life-Long Incremental RL [36, 37, 38]. We assumed that the optimal policy of the final task will generalize over all possible sub-task in the sequence.

We discussed the notion of threshold-based learning in an incremental learning process. Instead of training optimally, an agent on a given task before an increment, an appropriate

threshold value can be useful to optimize the training time. We also mentioned that this implies a wise selection of the training parameter to use as a performance indicator and a choice of the threshold score to use as a reference for the increment. We also indicated that identifying the appropriate parameter and value for the threshold is just one step. An appropriate incremental step must be defined to allow proper knowledge transfer and fast adaptation to previously unseen transitions after the increment, otherwise, the training process can be unsuccessful. We pointed out that an initial task must be defined properly to motivate early adaptation on the first task and generate the first threshold policy. This can be done by choosing a portion of the main task MDP where the sparsity is minimized.

In chapter 8 we test these theories by training a PPO [56] algorithm in a competitive multi-agent 3D environment using Self-Play Learning [29] in both single and cooperative scenarios.

Chapter 7

Experimental Environment

In this chapter, we introduce our experimental environment. We first introduce *Gym* which is a benchmark class to generate RL environments. We give an overview of the *slimevolleygym*[10] the Gym-based environment to train RL agents to play the slime volleyball game in 2D. After that, we introduce the *slimebot volleyball*[9] game, a 3D version of the slimevolleygym that we use as the experimental environment. Slimebot volleyball uses 3D graphic robotic software *Webots* for visualization and is more challenging compared to the slimevolleygym due to the additional dimension, and it is more appropriate for cooperative experiments.

7.1. Gym Environment

Gym [124] is a Python library from OpenAI used to create RL environments. It comes with baseline functions and attributes to create RL environments. The goal of Gym is to create a standard paradigm that will allow RL researchers to easily collaborate and share their works. Gym also has a wide community of independent researchers, academics, and professionals who discuss their results, and issues related to RL or the library itself.

7.1.1. Functions

The toolkit is a simple class, referred to as *gym.Env* composed of four basic functions:

1. **reset**: a function that generates the initial state s_0 of the environment. The function is mainly called at the beginning of each episode before agents perform their first actions. It returns initial observations of respective agents present in the environment. An observation can be a Python list of locations, velocities, etc., or any other information received from a sensor such as a camera, radars, etc.

2. **step**: is the transition function of the environment MDP from the current state to the next state. The function takes as arguments actions of different agents in the environment and will perform the environment dynamic based on these actions. It

returns a list of the following elements: *the next* observation for respective agents, the *rewards* for respective agents, a *boolean* variable stating if the episode is over or should continue, and a *dictionary* containing any other relevant information about the environment such as the current state, the current timestep, among others.

3. **render**: This function is in charge of displaying the current state of the environment in pixels. An example could be Atari 2600 games suits [45], which are pixel-based games. To optimize computation speed, it is advisable to avoid using the function during training.

4. **close**: This function closes the current render in a pop-up window. If not called, the current render will only be closed when the device is turned off.

These functions are just the basics for a *gym-like* environment. The freedom is given to customize them and to create additional functions depending on the problem to solve.

7.1.2. Variables

While using *gym.ENV* as a parent class of a customized environment, we need to define the following variables that will allow interactions with the *gym.ENV* class attributes and functions:

1. **observation_space**: An instance of the *gym.spaces* class, this defines the structure of the observation the agent receives from the environments. This can be a Python list in the case of a state-observation problem or a matrix array in the case of pixel observations. It is important to correctly define the observation_space because the success of training relies also on the agent's perception of the environment.

2. **action_space**: Also an instance of the *gym.spaces* class, it defines the structure of all possible actions. There are many ways of structuring it such as binary, discrete, among others.

3. **render**: Not to be confused with the function *env.render()*, this is a Boolean variable that indicates whether the function *env.render()* should be called or not. For an image-based environment, the training loop is faster when *render* is set to *False*.

As stated previously, while customizing a *gym-like* environment, additional attributes of the environment depend on the purpose and the problem to solve.

The Gym library comes with a variety of environment scenarios[1] which use the *gym-loop* paradigm such as *cartepole, lunar landing, mountain car, pendulum*, etc. These are mostly

[1] https://github.com/openai/gym/tree/master/gym/envs

used as baselines to introduce beginners to RL. In the next section, we introduce an advanced environment that requires a certain level of expertise to customize.

7.2. Slimevolleygym

A zero-sum game is an adversarial situation where 2 sides face each other and where profits are anti-symmetric. For instance, if player X made a profit of value b then player Y gets a profit (which will be the loss) of value $-b$, such that the summation of both sides' profits will always be zero. We introduce in this section a zero-sum game that we used as a testbed environment for our experiment.

Slime Volleyball is a simple but interesting volleyball game developed by an unknown author. It is a zero-sum game where two agents (named slimes in the game) face each other and try to win the game by performing two main tasks (1) defending their respective territory by stopping the ball to hit the ground on their respective side; (2) throwing the ball at the opponent's side aiming that it hits the ground. If the ball hits the ground on a player's side, the player loses a life and gets a score of -1. The other player gets a score of 1. If the game timer expires before either of the two players loses all their lives (maximum of 5 lives), the player left with the most lives wins the game, or if both players have equal remaining lives, the game is a tie and both agents get a score of 0 at the final step. The slimevolleygym [10] is a gym environment of the game, suitable for training RL agents and other AI algorithms[2] to solve the game by learning how to defeat the built-in expert player provided with the game, which masters the game almost perfectly. In this section, we provide an overview of the functionalities of the slimevolleygym environment before discussing the 3D version we customized in the following section [10].

7.2.1. Game structure

The game is structured as follows:

1. Observation: The observation can be pixels that can be rendered through a pop-up window. The agent uses a convolutional network to detect relevant features in the image. The observation can also be a state-observation, where the agent receives real-time (x, y) coordinate of both position and the velocity of mobile objects in the environment: *the opponent, the ball* and itself.

2. Actions: There are 3 basic actions: *forward* and *backward* following the $x-$axis, *jump* following the $y-$axis. Those actions can be combined to produce extra actions

[2]The author trained agents using genetic algorithms and the covariance matrix adaptation evolution strategy (CMA-ES).

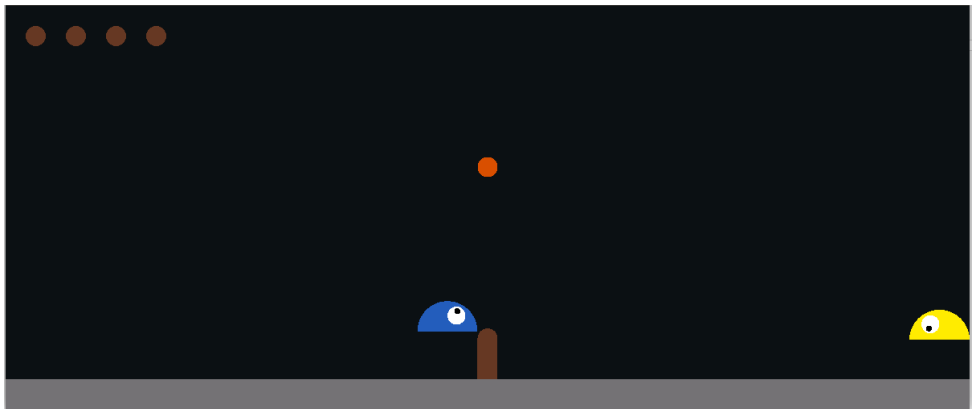


Figure 7.1: An illustration of the slimevolleygym game. The blue agent is the opponent, the game built-in expert. The yellow agent is the learner that aims to solve the game by learning how to defeat the expert. In the above scenario, the Yellow player has lost as there is no remaining life left.

such as *forward-jump*, *backward-jump*, *stay still*[3]. Which gives a total of 6 possible actions for the action space.

3. Timeline: An episode can last a maximum of 3000 timesteps maximum. This means an episode can reach the termination condition before reaching the maximum number of timesteps. It is a terminal state-conditioned game.

4. Reward: Each agent has 5 lives at most. Once an agent loses a life, it receives a reward of -1 while the opponent gets a reward of $+1$. For each agent, the overall reward at the end of an episode is the number of lives lost by the opponent subtracted by the number of lives they lost during the game. If one of the agents loses all of its 5 lives before 3000 timesteps, it loses the game terminates, as illustrated in Figure 7.1.

7.2.2. Pre-trained agents

Except for the built-in expert that comes with the game, the slimevolleygym library comes with some pre-trained agents that learned to play the game at a level close to the built-in expert:

- Agent trained with a covariance matrix adaptation evolution strategy (CMA-ES).
- Genetic Algorithm: The Genetic algorithm is trained with Self-Play Learning.
- PPO: Two pre-trained PPO algorithms come with the game (1) One is trained directly by facing the game expert; (2) The second is trained with Self-Play Learning.

[3]When no action is performed by the agent.

7.3. Webots

Webots[4] is an open-source multi-platform simulator application for mobile robotics manipulation. Webots is well suited for both academic and industrial research. It provides many environments and robot models suitable to simulate control problems such as RL problems.

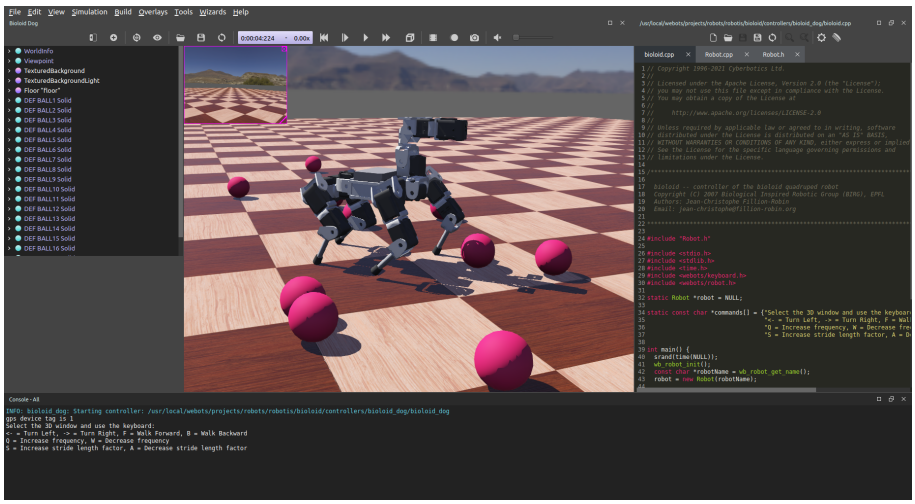


Figure 7.2: *Webots* robotics a real-time graphic simulator for robot manipulation in 3D dimensions

In addition to default models and environments, the software is easy to customize. It is possible to modify existing scenes and objects or to create new ones. The software supports the user to import objects from other graphics software such as *Blender*[5], *Autodesk Maya*, or any other 3D graphics software that can export 3D objects.

To control robots within the software, *Webots* uses the supervisor and controller platforms which help to command robots in the scene and manage all sensors such as distance sensors, cameras, radars, etc. The command platforms are independent of the programming language used. Webots offers APIs for many different programming languages such as C, C++, Java, Python, and MatLab, along with a built-in IDE that allows users to develop controllers directly in the software.

The built-in Python API makes Webots suitable for Machine Learning and RL problems in 3D environments. The Python API allows customization of *gym-like* environments that can use *Webots* as a visualizer. The motivation for using 3D software is to bring a physical representation of the real world in 3D simulation which makes the RL problem closer to reality.

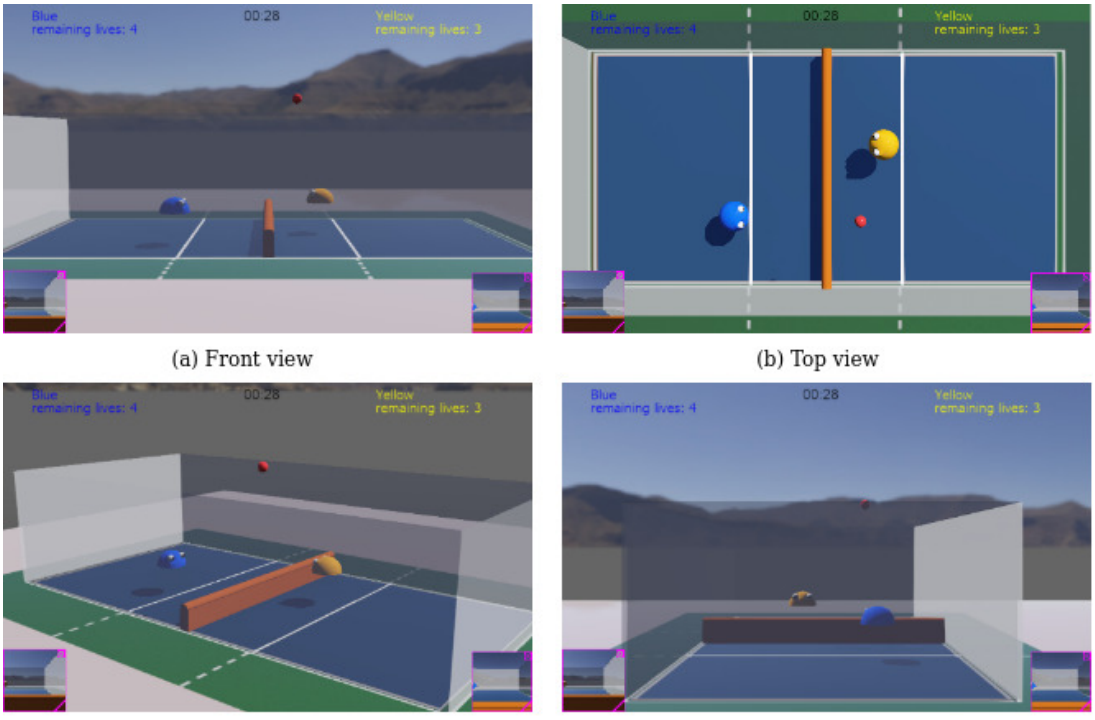
We selected Webots in our project for its flexibility in customization and easy communica-

[4]https://cyberbotics.com/
[5]A complete open source 3D software, https://www.blender.org/

tion with the programming language Python.

7.4. Slimebot Volleyball

Slimebot Volleyball [9] is a 3D version of the slime volleyball game. The game is implemented on top of the slimevolleygym environment and uses the robotic software *Webots* for visualization. Compared to the original game Slimebot Volleyball has an additional axis (z-axis) adding depth to the game area, such that players can move in all directions as in the real physical world. The motivation for creating the 3D environment is the ability to study cooperative scenarios involving two or more teammates per side in a more complex environment. Table 7.1 shows a short comparison between the slimevolleygym and Slimebot Volleyball environments.



(a) Front view (b) Top view

(c) Perspective view (d) Left view

Figure 7.3: Slimebot Volleyball environment with a single agent per side. The same timestep is shown in different views in Webots.

As in the 2D version, the 3D environment is a *gym-like* environment provided with controller instances of the Webots Supervisor and Robot classes to manage the interaction between the game environment and Webots. The physical objects in the game scene were created with *Blender* and imported into Webots. Despite the built-in physical system (collision, gravity, etc.) of Webots, we referred to the handcrafted physical system of the *slimevolleygym* game and brought additional features for compatibility in 3D.

7.4.1. Environment structure

The game has the same logical structures as the slimevolleygym game. Players defend their regions while throwing the ball on the opponent's side hoping that the ball hits the floor. However, the additional axis brings new features to the physical structure of the game.

1. Observation space

Using Webots, there are many ways to represent a player's observations:

1. Camera: Each slimebot has a camera that returns real-time pixels of the camera view. A camera is an object that can be paired with a mobile robot, when the robot moves the camera perspective moves as well. This approach is more realistic as it makes the environment state partially observable for each agent. Another option could be to set the camera unpaired with the robot and keep it in a static position somewhere in the 3D scene. The Camera uses the Emitter Receiver scheme to transfer information to the scene's global supervisor. This means that the camera must be provided with an Emitter device to send data to the global supervisor Receiver device to decode the message received from the Camera. However, this requires a bit more software engineering.

2. State observation: The state observation keeps track of the (x, y, z) coordinates of both position and velocity of all mobile objects present in the $3D$ scene. The length of the list will be $n \times 6$ with n the number of mobile objects in the $3D$ scene. In a single trainer scenario the observation will be a list of size 18 and In a collaborative scenario of 2 agents, each agent's observation will be a list of size 30 because we will have a total of 4 agents and 1 ball in the environment.

3. Distance sensors: The distance sensor detects objects near the sensors given a radius of sensitivity. We did not use it in our experiment, but we believe that it may be a realistic method of observation as real robots can have distance sensors.

In this project, we use **state observation** to train agents. Because of its simplicity to set up and low expense in computation. It also does not require running the training with the simulator. The training can be run with a command line, and it also offers the possibility to run training in parallel.

2. Action space

The additional axis brought two more basic actions in the action space. The game has now 5 basic actions: forward (x-axis), backward (x-axis), left(z-axis), right(z-axis), and

Table 7.1: Comparison between Slimevolleygym and Slimebot Volleyball Environments

Slimevolleygym	**Slimebot Volleyball**
Continuous space	Continuous space
2D	3D
Stationary	Can be stationary or non-stationary
6 actions	18 actions
Single competitive	Multi-agent competitive scenarios
Lasts maximum 3000 timesteps	Lasts maximum 3000 timesteps

jump(y-axis). In this scenario, 3 actions at most can be combined as long as there is no contradiction between actions[6]. We used a one-hot encoding representation to encode each action as a list of 5 elements, where 1 means that the action is active and 0 means that the action is nonactive. We have the following one-hot encoding for the basics actions:

- forward : $[1, 0, 0, 0, 0]$
- backward : $[0, 1, 0, 0, 0]$
- jump : $[0, 0, 1, 0, 0]$
- right : $[0, 0, 0, 1, 0]$
- left: $[0, 0, 0, 0, 1]$

The additional actions can be set by doing an element-wise addition of lists of combined actions. For example:

$$\begin{aligned}\text{forward-jump-right} &= [1, 0, 0, 0, 0] + [0, 0, 1, 0, 0] + [0, 0, 0, 1, 0] \\ &= [1, 0, 1, 1, 0]\end{aligned}$$

The overall number of possible actions is 18, which means the size of the action space is 18.

7.4.2. Environment key attributes

In this section, we give an overview of functionalities of relevant functions and variables of the environment. The environment is by default dynamic along the $z-$axis that we refer to as the *depth* of the game scene. In other words, the depth of the game area can vary values in the range $[0, 24]$ during the training. The length ($x-$axis) and height ($y-$axis)

[6]Combining *left* and *right* doesn't make sense because they will cancel each other out, same for as combining *forward* and *backward*.

are static and have both maximum values of 48. The default incremental learning process in the environment consists of starting the training with a small depth and incrementing it progressively until it reaches the maximum value.

The structure of the environment dynamics is mainly managed by a given class WORLD, whose instance can be accessed with the command *env.world*. This class has special attributes and functions such as:

1. setup: Function that initializes the game environment structure, especially the depth ($z-$axis) at the beginning of the training or an evaluation. It receives a special parameter *init_depth* which is the depth of the environment at the beginning of the training.

2. increment_world: Function that performs the increment of the environment MDP during training. This means the size of the game space will increase according to the $z-$axis. In addition to making the environment wider, it also increases the speed of the ball along the $z-$axis[7].

3. step: Represents the *incremental step* discussed in section 6.4. It is a parameter that decides how many units of the current depth should be incremented. It is a floating point and every time the function *increment_world* is called, the depth of the environment grows for *step* units:

$$world.depth \leftarrow world.depth + world.step \tag{7.1}$$

   as long as the following conditions are respected:

$$world.depth + world.step \leq world.max\_depth \tag{7.2}$$

4. increment: a Boolean variable that indicates if the training will be normal or incremental. It has the value *Fasle* by default, which means the training starts directly with the maximum depth of the environment.

5. stuck: a Boolean variable that freezes the increment option during the training. If set to True, the selected initial depth will stay constant even if the agent surpasses the default performance threshold.

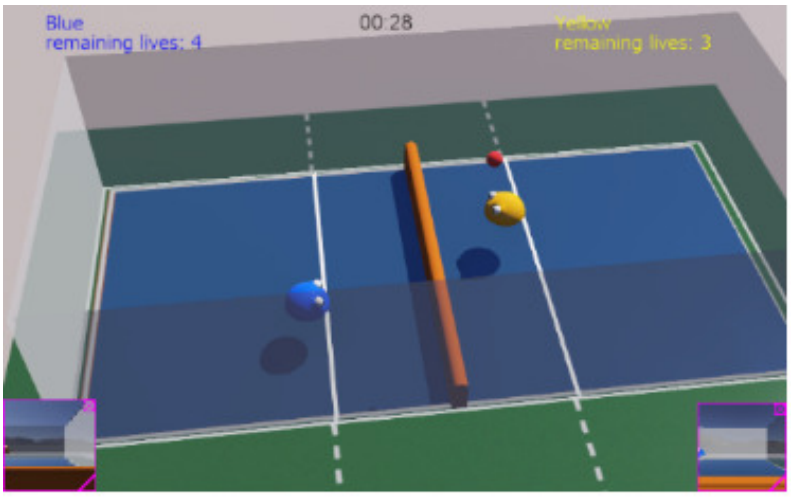
6. depth: Represents the game area along the $z-$axis direction, it is the only dynamic parameter of the environment structure. No mobile object can go beyond the area bounded by the current depth.

[7]The ball speed along the $z-$axis moves proportionally to the current depth.

There are other variables, functions, and classes in the game implementation such as *INCREMENT_THRESHOLD* which indicates the average performance score from which the depth can be incremented. We provide a link in [9], for more details about the implementation.

7.4.3. Scenarios

The game comes with two scenarios:



(a) Single Trainee Scenario

(b) Team Collaboration Scenario

Figure 7.4: Illustration of two different scenarios of the slimebot volleyball game. (a) competitive scenario, where a single agent learns to defend its area; (b) cooperative-competitive scenario, where 2 agents collaborate to defend the same area.

- Single agent: The first scenario is similar to the slimevolleygym scenario, there are just two players in the game: The yellow player represents the training agent that aims to learn the game. The blue player represents the opponent that the yellow

player learns to defeat. In the Self-Play training mode, the blue player can be the real-time version of the Yellow player or the best recent copy of the yellow player.

- Cooperative agents: The second scenario is similar to the first scenario, except for the fact that each side is a team of two players who have to collaborate to defeat the other team. This scenario is more challenging because agents must avoid the ball touching the ground on their side while avoiding bumping into each other at the same time.

The structure of the game area allows the implementation of more than two agents per team. But this would have higher computational requirements.

7.4.4. Pre-trained agents

The game comes with a trained PPO algorithm that solved the game with Incremental Learning. We selected the PPO algorithm because it is suitable for an environment with continuous state or/and action spaces. In addition, it has been successful in the slimevolleygym environment [10]. The game comes also with pretrained MAPPO agents, which is a team of two independent PPO algorithms that are trained to collaborate incrementally to play the game.

7.5. Summary

In this chapter we gave an overview of the environment we used for our experimental evaluation. We discussed the OpenAI *Gym* library, which is a toolkit that brings a standard paradigm to develop RL environments for experiments. We also mentioned that the environment we used is built on top of the slimevolleygym environment [10], which is a single competitive volleyball game in a 2D environment, and we specified that even if the experimental environment allows pixel observation, we used the state observation technique because of implementation flexibility and the possibility of running the environment without the simulator *Webots*.

We also gave an overview of the functionalities of the experimental environment that permit incremental learning. We mentioned that the environment comes with a single agent and cooperative multi-agent scenario. However, unlike the slimevolleygym game, the environment does not have experts that can be used as baselines in both single and cooperative scenarios. We used Self-Play Learning in both situations to train agents. We also associated Reward Shaping in the Multi-Agent scenario. We discuss more of this in Chapter 8.

Chapter 8

Experimental Evaluation

Recall the objective of this work. We attempt to train a team of two independent PPOs to solve a complex task in a volleyball game in a 3D environment. Before diving into it, we first start by addressing the challenge we had to train a single PPO agent to solve the game in a single competitive scenario. We used the PPO2 implementation of OpenAI stable-baselines [32] in our experiment. We tried training the PPO agent in the environment from scratch, but it failed because the environment was too challenging. We tried tuning some hyperparameters such as the learning rate, the discount factor, the experience replay batch size per update, the number of epochs per update, and others. But the agent failed to train. In the first section of this chapter, we discuss how we managed to train a PPO agent in the environment by incrementing gradually the environment MDP. After that, we discuss, in the second section, how we applied the paradigm of Reward Shaping [68, 69, 70, 71] to train the team of 2 independent PPOs to collaborate and incrementally solve the environment.

8.1. Single Agent Evaluation

As mentioned in section 7.4, our experimental environment is a transposition of the slimevolleygym environment from 2D space into 3D space. Experiments in [10] have shown that the PPO agent has successfully solved the slimevolleygym game from scratch by reading a 2D coordinate system in both supervised training[1] and Self-Play [29] training. We can see in Figure 8.1 (a) that the agent solved the game before 3 million timesteps of training horizon[2] in Self-Play training. As shown in Figure 8.1 (b) the average reward per episode was not suitable to measure the training progress. This can be explained by the fact that slimevolleygym is a zero-sum game and with the Self-Play learning both opponents' expertise are almost in equilibrium at every timestep. Hence, the average return will keep fluctuating around 0 during the training.

[1]The slimevolleygym game has a built-in expert that helps to teach other agents to solve the game
[2]The agent was training in a loop of repeating games.

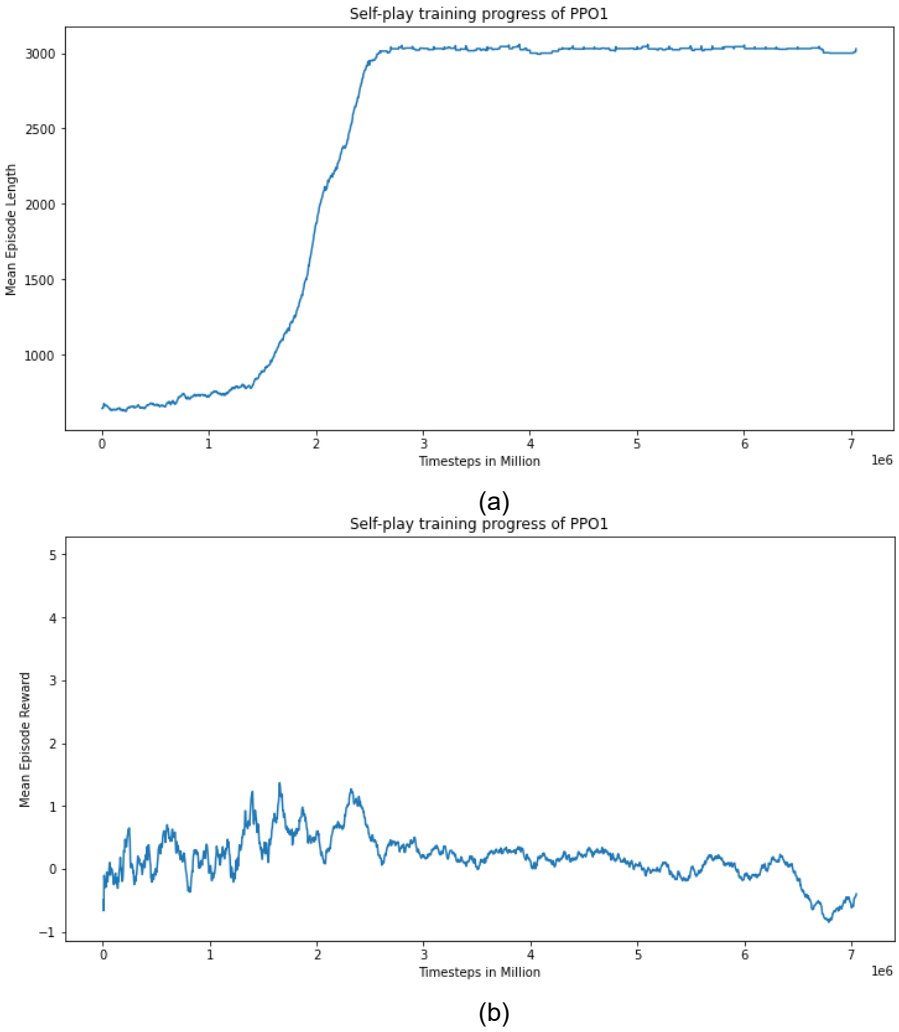


Figure 8.1: Training progress of a PPO algorithm in the 2D environment slimevolleygym using Self-Play training: (a) Training progress of average episode length. (b) Training progress of average episode reward

We used the experiment in Figure 8.1 as a reference to train the PPO agent to solve the 3D version of the game. However, the slimebot volleyball environment was too challenging for the PPO agent. We assumed that the additional dimension increased the state space size, which made the transition between states a more complex problem. In this section, we discuss how we successfully train the PPO agent to solve the environment despite the increasing complexity. Recall the result in Figure 8.1, we use the average episode length as the performance indicator during the training.

8.1.1. Training in a Stationary Environment

In this series of experiments, the depth of the environment stays static during the whole training horizon. We tried different settings: 3D space (the full game environment), 2D space (a mapping of the 3D environment in 2D), shrunk 3D space (The depth of the environment is a positive number less than 24), and trained the agent for a maximum

training horizon of 20 million timesteps.

1. Training in the full Environment

We kept the default environment size and trained the agent for 20 million timesteps. As shown in Figure 8.2, after more than 20 million timesteps the agent was not showing progress. We tried some hyperparameter tuning such as the learning rate, the discount factor, and the replay buffer size. But the agent did not show any particular progress. We assumed that the following reasons were challenging the agent to solve the 3D environment:

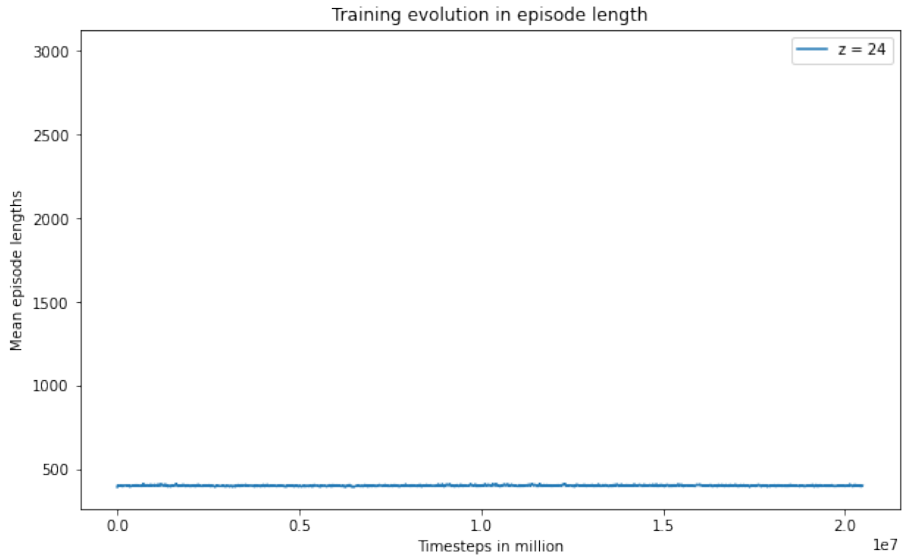


Figure 8.2: Training from scratch of a PPO algorithm in the *Slimebot Volleyball* environment. We can notice that after more than 20 million timesteps the agent is not showing particular progress.

- Sparsity: Both 2D and 3D versions are sparse reward problems. However, the size of the state space in the 2D version gives a higher probability of the agent hitting the ball and generating a positive reward during an episode. In the 3D version, the state space is larger than in the 2D version such that the probability of generating a positive reward is low.

- Randomization: To improve its policy the agent needs to notice that hitting the ball and returning it to the other side is good. This starts when the agent occasionally collides with the ball. However, at each initialization of a match, the ball is launched randomly on the learning agent's side. With the large state space, the probability that the ball hits the agent and gets back to the other side is low.

- The z-axis: In the 2D version when the ball collides with the agent it can only move forward or backward, depending on the collision direction. With the full structure, the ball can also move to the left or right, which brings more complexity to the possible orientation the ball can take after a collision.

We thought maybe this is an unsolvable problem by transposing the same rule and structure from 2D into 3D. Thus, we decided to limit the 3D version to a 2D projection in an attempt to simplify the environment sufficiently. This means that we did a projection of the 3D coordinate onto a 2D coordinate and investigated if the agent could solve the game.

Training in a 2D Projected Space

We did a mapping of the XYZ-coordinates system into the XY-coordinate system by freezing the z-axis. In other words, the agent observes 3 coordinates but the z coordinate is set to 0 at every timestep for both location and speed of all objects in the game scene. As shown in Figure 8.3, mobile objects can only move along x(the length) le and y(the height) axes.

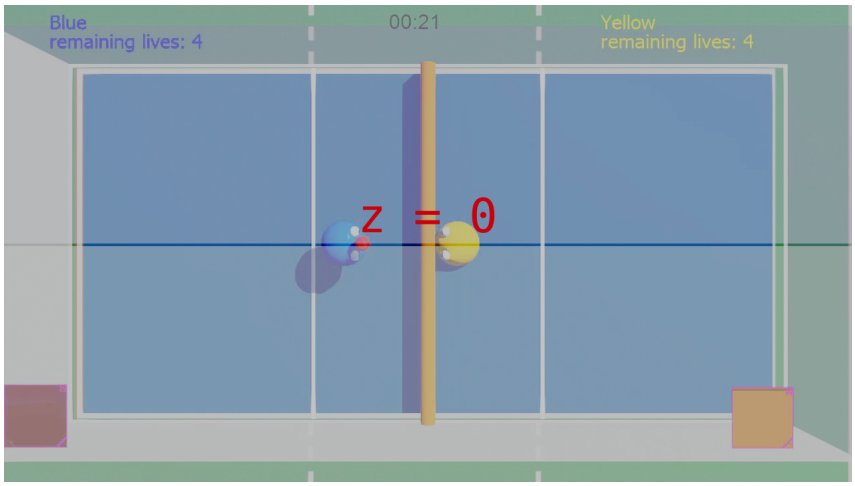


Figure 8.3: Top view illustration of the environment with a depth $z = 0$. Agents can only move inline and cannot access blurred areas in the figure.

The following command is given to the environment before the training starts:

- env.stuck = *True*: To ensure that the depth will stay stationary during the training and evaluations.
- env.world.setup(*,init_depth = 0)[3]: Setting the initial depth to 0, with the previous parameter set to *True* ensures that the third coordinate will stay at 0 at every timestep for all mobile objects in the game scene.

This setup responded as expected. The 3D version with a frozen $z-$axis responded similarly to the 2D version. As shown in Figure 8.4, the PPO agent started playing the game optimally before the 3 million timesteps of the training horizon. This confirms that the full environment space is too large for the agent to adapt.

[3]The * means that there are other parameters passed to the function.

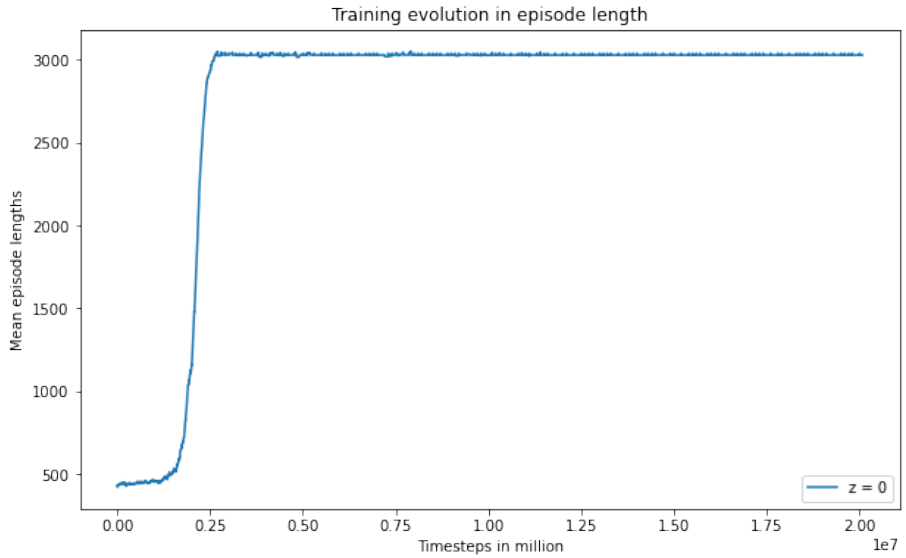


Figure 8.4: Training of a PPO agent in a 2D projection of the slimebot volleyball environment space. The agent reads 3 coordinate systems but in the form of $(x, y, 0)$. We can notice that before 4 million timesteps the agent was already playing optimally the game.

To confirm that the state space size of the 3D version is the problem, we investigated training the agent with a shrunk 3D environment, by making the depth ($z-$axis) of the game area small enough to allow the agent to move in all directions[4]. We discuss the result of this new experiment in the next section.

2. Training in a Shrunk 3D Space

Experiments in previous sections gave us the assumption that the large MDP of the 3D environment is the reason why the agent fails to solve the game. We tried a different experiment by shrinking the depth of the original environment. The agent has a diameter of size 3 units, we made the 3D area small enough to allow the agent to move in all directions and be able to experiment with the full action space. We experimented with a depth of 4 units[5]. This means that the z coordinate for both location and speed can be different from zero and mobile objects are not allowed to go beyond the area bounded by the depth. Figure 8.5 illustrates a shrunken 3D environment with a depth $z = 12$

In this setup, the agent moves along the $z-$axis with the same speed as with the $x-$axis. However, the ball speed along the $z-$axis is constrained to be proportional to the current size of the depth of the game space. The speed of the ball along $z-$axis, noted *ball.vz* in

[4]In 2D the agent can only move forward, backward and up. In 3D the agent can also move left and right.

[5]recall that the maximum depth is 24 units.

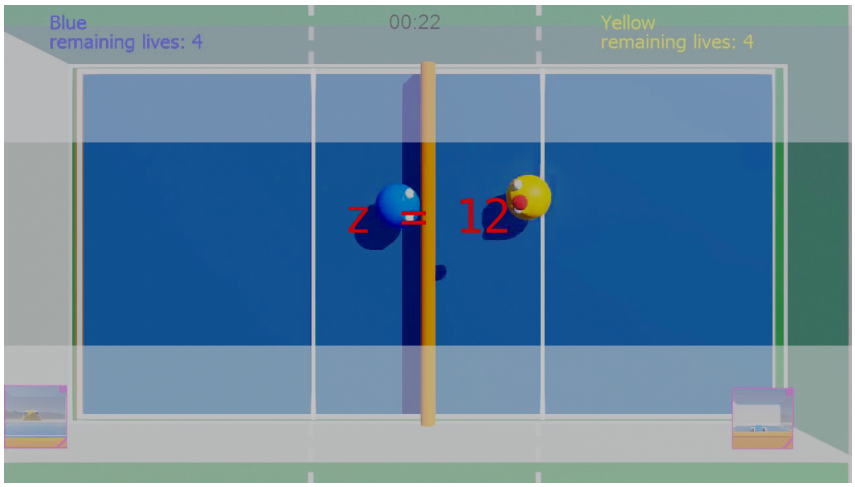


Figure 8.5: Top view illustration of a shrunk environment with a depth $z = 12$. Agents can move in the 3D space but can not access blurred areas in the figure.

the implementation is given shown in equation 8.1.

$$ball.vz \leftarrow ball.vz \times \frac{world.depth}{world.max\_depth} \tag{8.1}$$

In the case of $depth = 4$ $ball.vz$ will be computed as follows:

$$\begin{aligned} ball.vz &= ball.vz \times \frac{4}{24} \\ &= \frac{ball.vz}{6} \end{aligned}$$

This means that every time a vector speed (v_x, v_y, v_z) is computed by the game dynamics, v_z is divided by 6 before updating the ball position. For example, in the full game environment, the ball can move along z with a maximum speed of 22.5 units per timestep, proportionally to an environment of depth 4 the ball can only move along z with a maximum speed of 3.75 units. This trick made the ball move correctly along the $z-$axis according to the given depth.

After experiments, the agent was able to solve the environment with a depth of 4. This confirms that the state space with a depth of 24 is too wide to be solved. We also tried different values of depth such as 6, 8, etc., and noticed that the agent solved those environments. However, Figure 8.6 shows that the larger the depth the longer it takes the agent to solve the game. We can see in **Figure** 8.6 that $z = 4$ took longer to be solved than $z = 0$, and the same for $z = 6$ compared to the two former depths. This shows that solving the game becomes more challenging as we select a wider depth until it could not be solved by the agent. As shown in Figure 8.6, after 20 million timesteps, with $z = 8$ the agent seems to make a very small amount of progress.

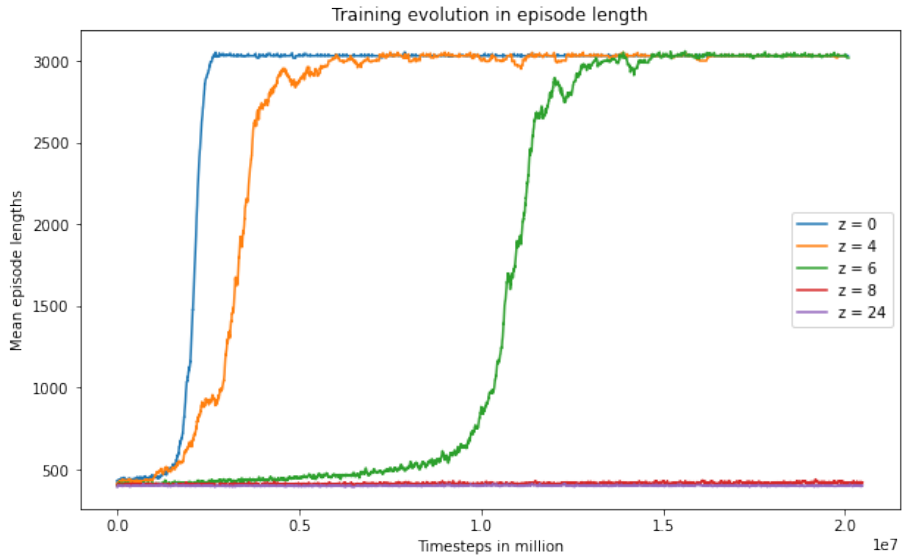


Figure 8.6: Training evolution of a PPO agent with different depths. We can notice that the higher the depth is the longer it takes to solve the task.

3. Experiment Summary

Based on the result of the experiments we concluded that:

- The Slimebot Volleyball is an environment with large state space, and according to the game rule and randomness, the game MDP is hard to solve even if the state is not partially observable[6].
- Reducing the size of the environment dynamic makes the game solvable.
- It is imperative that the reduced environment be sufficiently uncomplicated to optimize the likelihood of adaptation. If this condition is not met, the agent may continue to encounter difficulties in resolving the game.

Now we can confirm that the size of the slimebot volleyball environment is the obstacle blocking successful training, it became necessary to identify an alternative approach to overcome this obstacle. Fortunately, Chapter 6 introduced the concept of Incremental RL which has the perspective of addressing situations where target tasks are difficult to solve from scratch. We broke down the problem into a sequence of sub-problems and instruct the agent to learn progressively how to solve increasingly complex MDPs with the same objective function. We provide details about this experiment in the following section.

[6]The agent has all relevant information of states using state observation.

8.1.2. Training with Incremental MDPs

Experiments in the previous section have shown promising avenues for further investigation. The PPO agent can play the game using 3D coordinates as long as the game space is not too large. In this section we discuss the experiments performed to allow the agent to solve the full environment MDP using Incremental Learning.

As discussed in Chapter 6 we decided to use Incremental Learning to break down the main task into a sequence of sub-tasks where the first task in the sequence is the simpler and the last task is the most difficult. We initialize the training with a shrunken 3D space and set up a performance threshold. Once the agent reaches the threshold score in an evaluation, the depth increments according to a given incremental step. hence, the agent will have a new problem to solve where it will start experiencing unseen transition and will have to control a ball moving faster along the z-axis than previously(as the ball speed is in proportion with the current depth along the $z-$axis). The expectation is to see the agent adapting to the new transitions by leveraging the skill acquired before the increment. We notice the following phenomena:

- If the agent can adapt to the new states that it did not experience initially, we add another portion of the environment and repeat the process until the full environment is mastered or until the agent could not adapt anymore.

- If the agent can not adapt to the additional states, this could mean that the increment was too wide to encourage adaptation and we would decrease the incremental step. Otherwise, it would mean that Incremental Learning is not a solution for this particular problem.

We customized the OpenAI stablebaselines [32] implementation of PPO2 by adding some features to track the agent performance during the training. We followed the same training logic as in the previous section; the agent can move in all axes without any constraint, but the ball speed is limited as in equation 8.1.

In section 6.4 we have seen that the success of the incremental learning relies on three parameters: the initial task $\mathcal{T}_0$, the performance threshold δ and the incremental step η. Before discussing the detail of the experiment, we start by giving an overview of how those parameters are defined in our specific case.

- Initial Task $\mathcal{T}_0$: The previous experiment showed that training can be initiated with a shrunk 3D environment. Even though using the Projected 2D space as the initial task has the likelihood to quickly initiate the knowledge acquisition. In the implementation, $\mathcal{T}_0$ is determined by a parameter called *initial_depth*. The highest

successful initial depth we tested is 6. However, we would not determine if it is the maximum that the agent can start training with. For simplicity, we considered the $z = 0$ as the initial task in the experiment.

- The Incremental Threshold δ: At the beginning of the training the average episode length fluctuates around 450. During the experiment, we observed that once the average episode length reaches 480, the agent starts adapting to the game. Hence, we can choose $\delta \in [480, 3000]$ as a performance threshold value for the increment.

  We used the training-evaluation paradigm, this means that the increment occurs only if the agent beats the threshold during the evaluation.

- Incremental Step η: As we are starting with a depth of 0, the incremental step can take a float value in a range of $(0, 24]$. We used a static incremental step in our experiment. This means at each increment, the same quantity is added to the current depth. In the implementation, the incremental step is computed by the environment once it receives the number of times it should increment.

1. Experiment overview

We chose $\mathcal{T}_0$ to be the simplest possible environment with $z = 0$. As discussed in Section 6.4, we investigated the two other parameters δ and η to see if they influence one another. We fixed the step size and tried 4 different threshold values: $500, 800, 1500, 2500$, to observe the evolution of the incremental training. For each combination of step size and threshold, we tried experiments with 10 different seeds to have an overview of how both parameters influence each other on average.

2. Performance Threshold $\delta = 500$

We started our investigation by using a small incremental step of $\eta = 1$ as an initial experiment. This means that the environment must be incremented 24 times to reach the full environment space.

Figure 8.7 (a) shows that incrementing the environment MDP gradually is a viable approach to solve the entire game. The 24 red diamonds on the graph tell us that the PPO agent was able to adapt at each increment from depth 0 to depth 24 by adding 1 unit to the current depth each increment. However, Figure 8.7 (b) shows us that incremental learning is sensitive to the initialization of the agent's parameters. We notice in (b) that the agent was able to increment only 10 times in 20 million timesteps of training.

In both experiments, we notice that after the fourth increment, the average training length drops down. This can be explained by the fact that the agent has a diameter of 3 so from

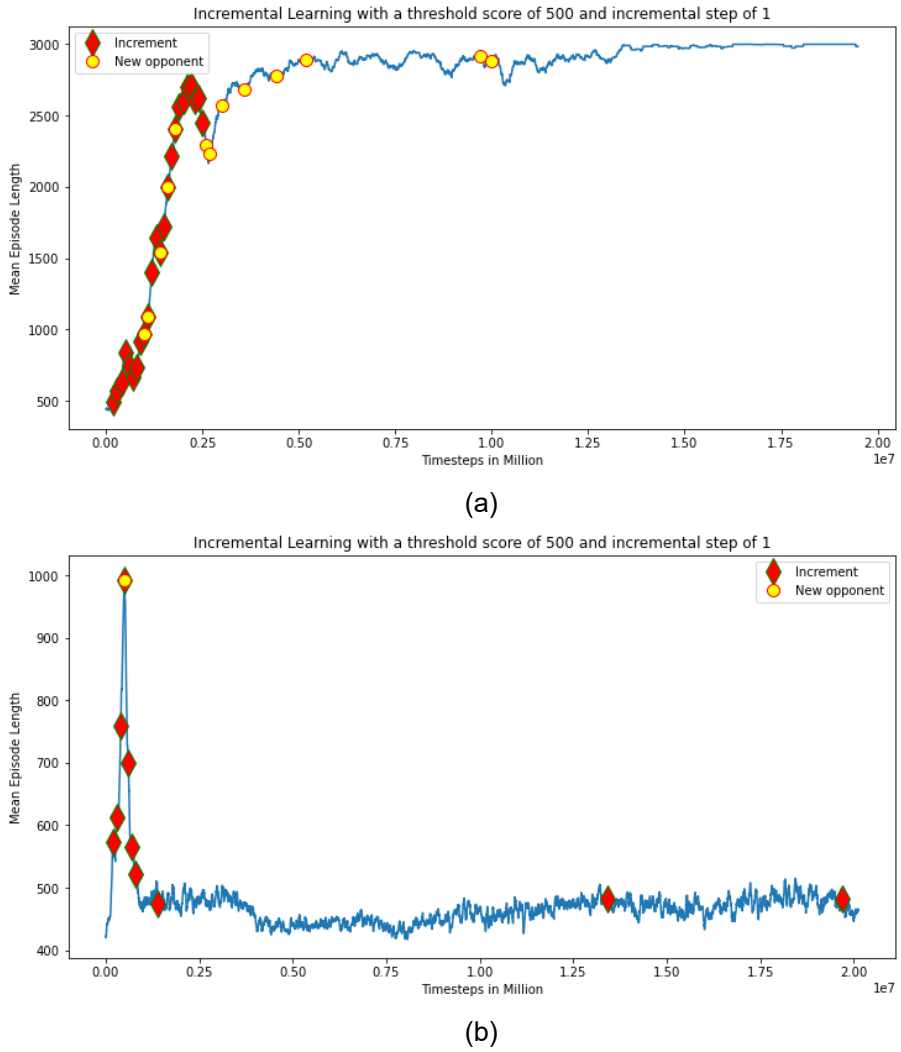


Figure 8.7: Incremental training progress with two different seeds. (a) successful adaptation within 20 million timesteps. (b) slow adaptation within 20 million timesteps.

$z = 0$ to $z = 3$ the agent can only move along a line. The exploration of the full action space starts with depth values greater than the agent's diameter ($z = 4$).

Based on the two results in Figure 8.7 we saw the importance of using random initialization to have an overview of the agent average adaptability with the given threshold and incremental step. We used 10 different seeds and plotted the average performance along with the standard deviation. We repeated the same experiment with bigger incremental steps to see if the agent will still be able to adapt to solve the full environment.

We observe some interesting results in Figure 8.8.

- (a): For step size $\eta = 1$, we notice that on average the agent can adapt incrementally to the full environment. However, the standard deviation is very wide in almost every timestep. This means that for some randomization the agent can increment quickly and solve the environment MDP, and for others, adaptations to the increment are

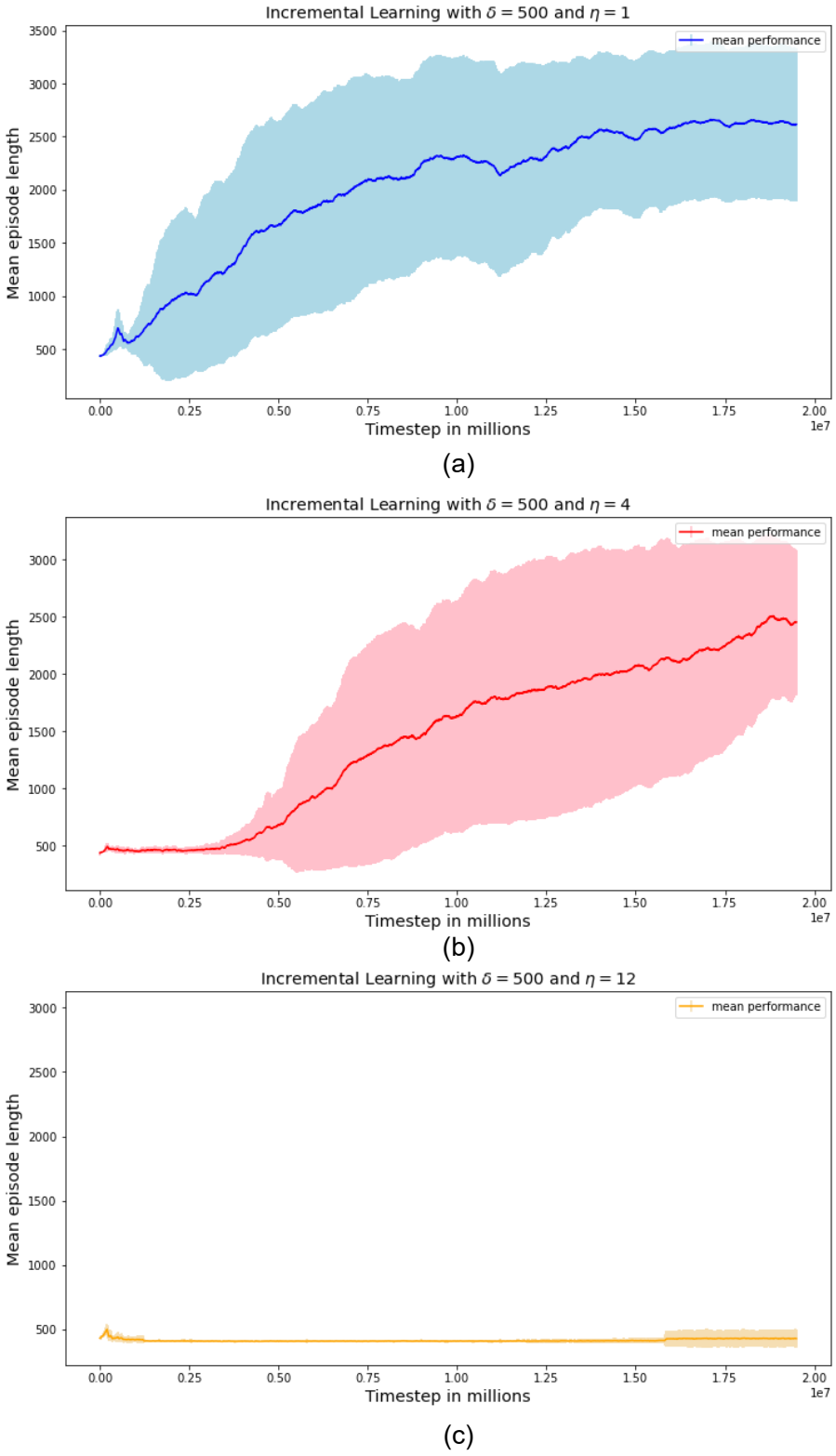


Figure 8.8: Incremental training progression of 10 different seeds with $\delta = 500$. (a) incremental step $\eta = 1$; (b) incremental step $\eta = 4$; (c) incremental step $\eta = 12$

very slow or not effective in 20 million timesteps.

- (b): For step size $\eta = 4$ the environment has to be incremented 6 times. We notice

in the plot that with a bigger step, the beginning of the adaptation occurs very late for all the different initializations. However, the agents still adapt on average. We still notice a bigger standard deviation compared to 8.8 (a). This means adaptation is slower on average for $\eta = 4$ than $\eta = 1$.

- (c): For step $\eta = 12$, the experiment confirmed the assumption made in section 6.4 that a big incremental step can have a negative influence on the incremental learning process. on average, 20 million timesteps were not enough for the agent to adapt after the first increment. However, we point out the small average change after 16 million timesteps, this means that at least one of the trained agents started at a very late stage to adapt after the first increment.

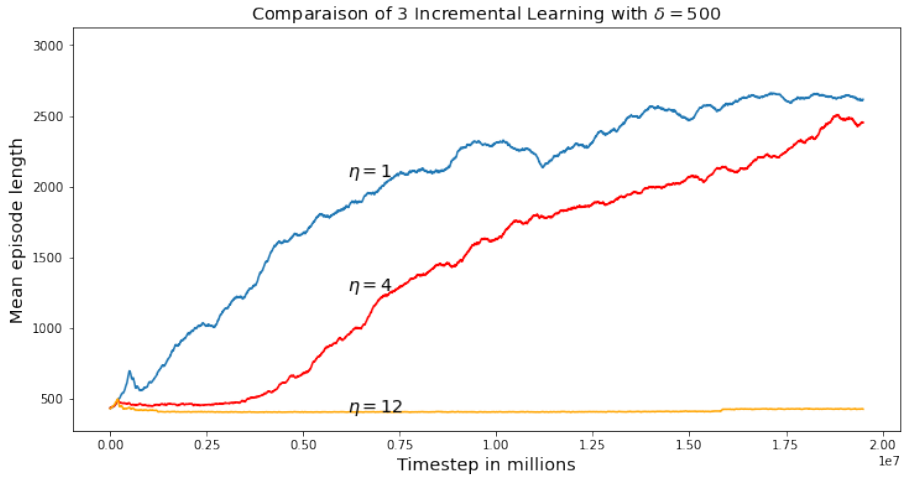


Figure 8.9: Comparison between three different incremental step sizes, and with the same threshold $\delta = 500$.

In Figure 8.9 we plotted the three average performances of Figure 8.8 in the same plot to have a better view of the performance among the 10 different initialization. It is evident from this instance that the incremental step of $\eta = 1$ exhibits a superior rate of adaptation relative to the other two incremental steps.

To investigate more, we repeat the same experiment with three different threshold performance thresholds.

3. Performance Threshold $\delta = 800$

We did another experiment by increasing the performance threshold. We used the same initialization to investigate the average adaptation of a larger threshold with the same step sizes. The results were similar to the previous experiment with $\delta = 500$. We notice in Figure 8.10 that:

- (a): For step $\eta = 1$ the adaptation on average is faster and has a small standard

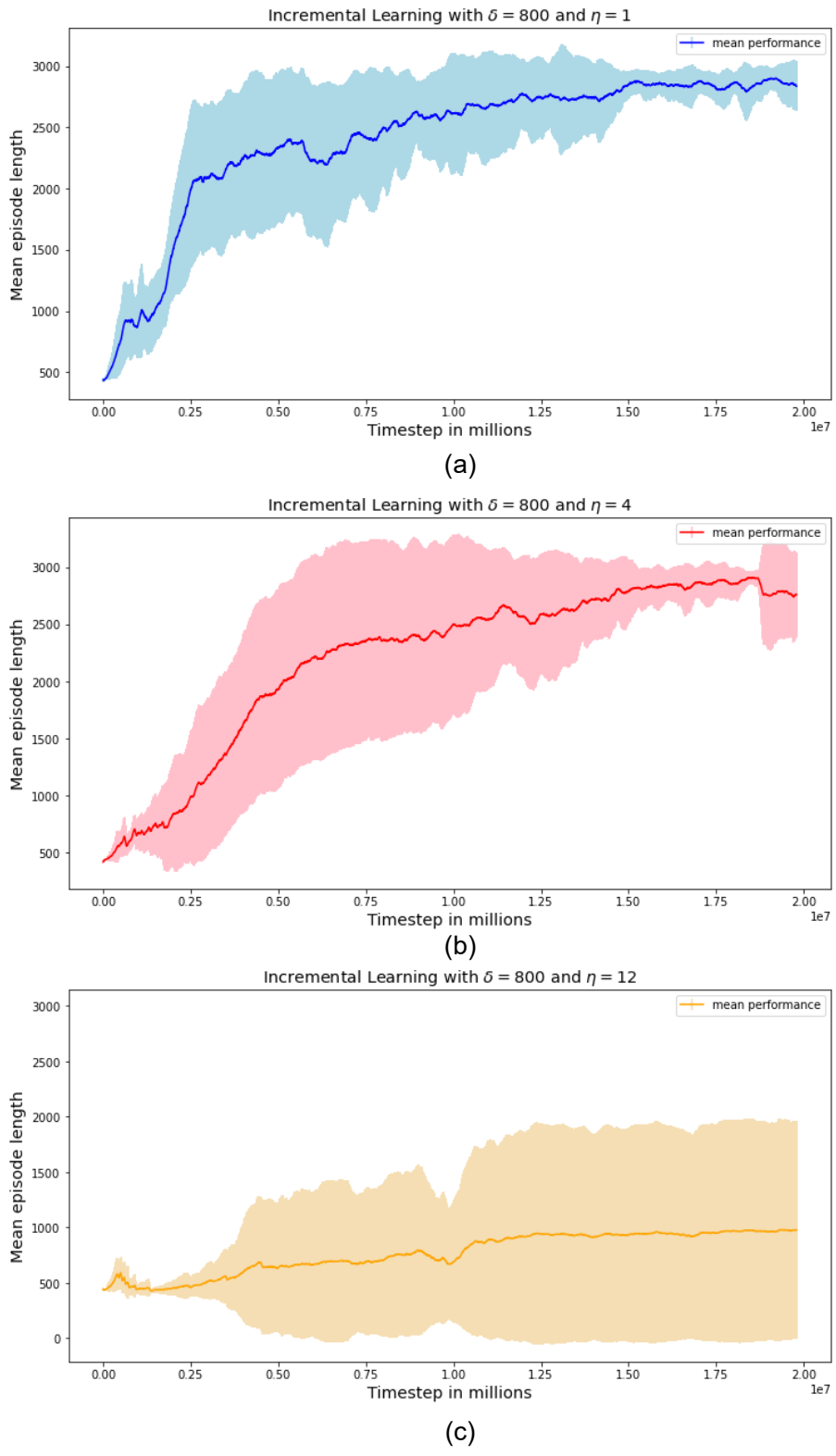


Figure 8.10: Incremental training progression of 10 different seeds with $\delta = 800$. (a) incremental step $\eta = 1$; (b) incremental step $\eta = 4$; (c) incremental step $\eta = 12$.

deviation compared to Figure 8.8 (a). We notice how the magnitude of the standard deviation reduces over an extended duration. this means all initialization has been able to increment 24 times and started performing similarly before 20 million

timesteps.

- (b): For step $\eta = 4$ the progress is almost the same as (a) but the standard deviation is bigger in early training timesteps. This means there was a slow adaptation for some initialization. However, agents performed much better than in Figure 8.8 (b).

- (c): For step $\eta = 12$ the shape of the average curve and standard deviation show that despite a very big increment, some agents can adapt. We show in the appendix that some agents can reach $z = 24$ before 20 million timesteps.

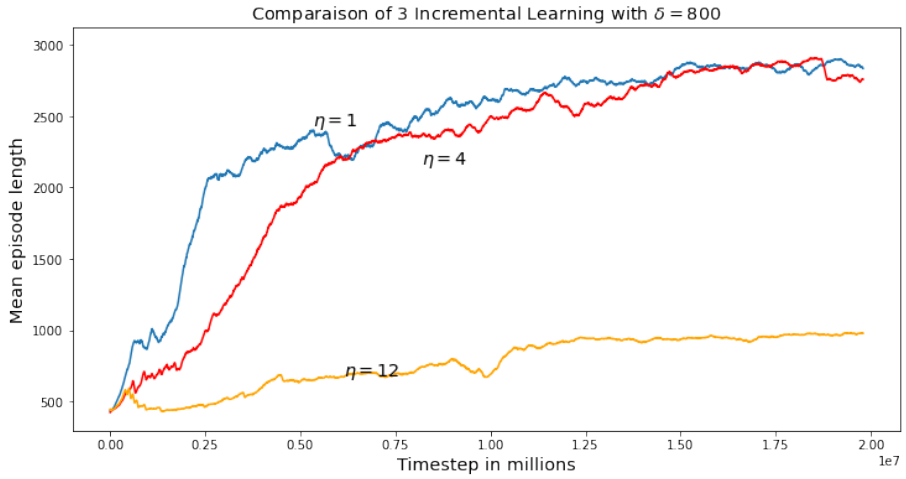


Figure 8.11: Comparison between three different incremental steps and with the same threshold $\delta = 800$.

In Figure 8.11 we also plotted the three averages performance of Figure 8.10 in the same plot. We notice $\eta = 1$ has early adaption on average. However, the performance of $\eta = 4$ starts overlapping with $\eta = 1$ after some training steps.

4. Performance Threshold $\delta = 1500$

In 8.12 the results of the experiment with a threshold of 1500 is shown:

- (a): For $\eta = 1$ the incremental adaptation is fast on average with a small magnitude of standard deviation. There are no major differences with Figure 8.10 (a), except that here the standard deviation stays small from the outset.

- (b): With step $\eta = 4$ the adaptation is faster with small standard deviation comparing to 8.10 (b).

- (c): For $\eta = 12$ we notice that there is better performance on average compared to Figure 8.8 (c) and Figure 8.10 (c). However, the standard deviation is large, which means some initializations struggled.

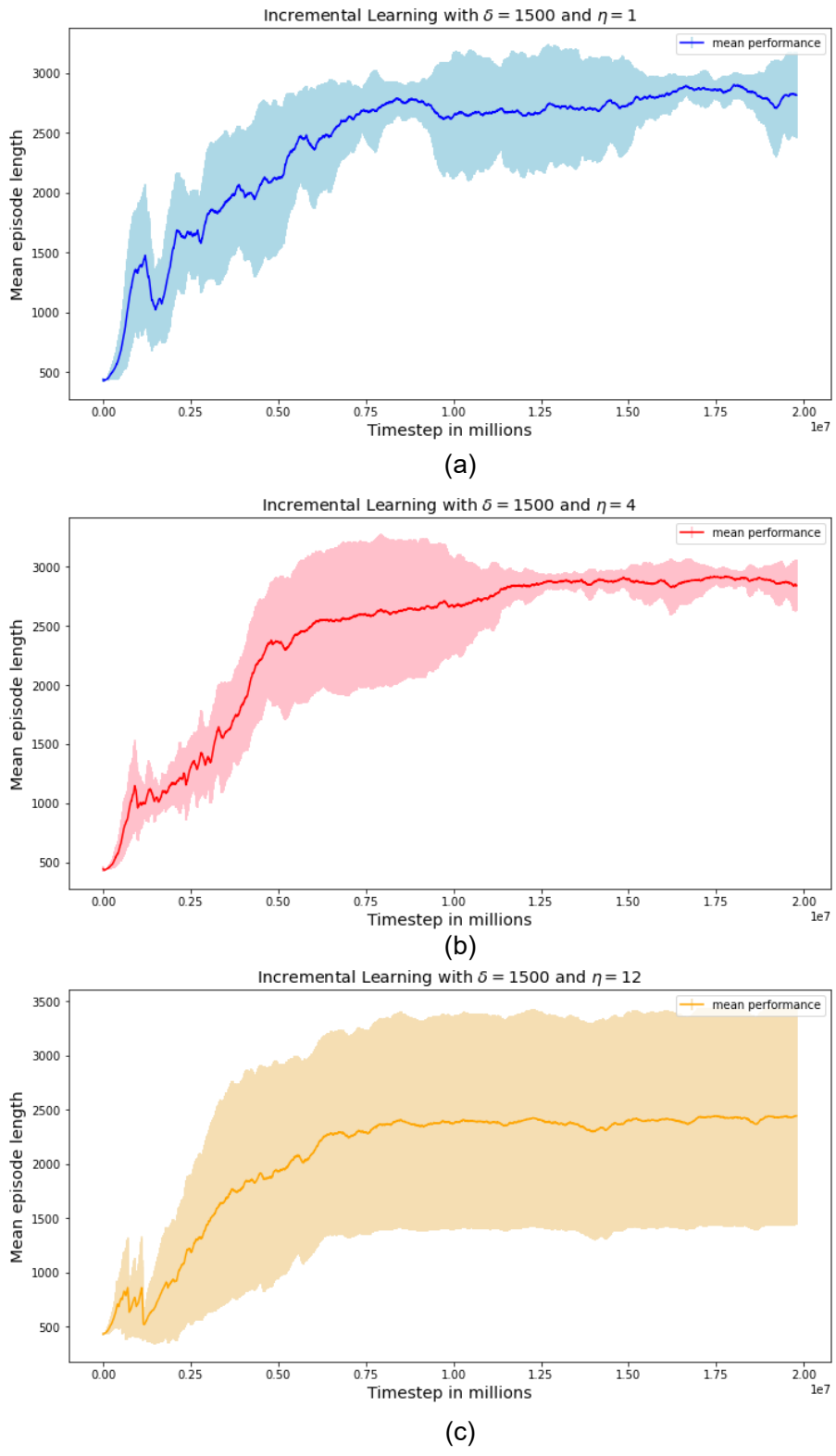


Figure 8.12: Incremental training progression of 10 different seeds with $\delta = 1500$. (a) incremental step $\eta = 1$; (b) incremental step $\eta = 4$; (c) incremental step $\eta = 12$.

For $\delta = 500$ and $\delta = 800$ we see that step $\eta = 1$ performed better than $\eta = 4$ on average. However, in Figure 8.13, they overlap on average. We also see how the average performance with $\eta = 12$ is almost the same as the two other incremental steps.

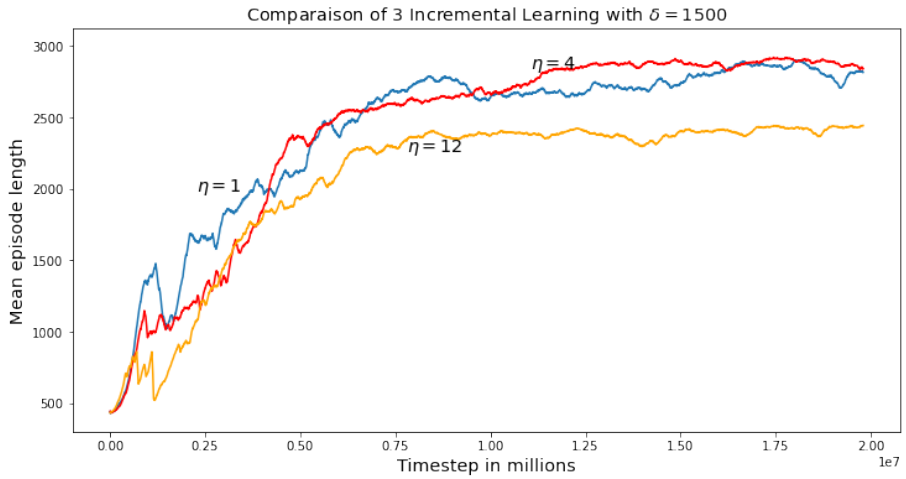


Figure 8.13: Comparison between three different incremental step sizes and with the same threshold $\delta = 1500$.

5. Performance Threshold $\delta = 2500$

2500 is a high threshold for the game, close to the optimal episode length of 3000. Figure 8.14 show the results of experiment:

- (a) For $\delta = 1$ the result is similar to the results shown in Figure 8.12 (a).
- (b) For $\eta = 4$ we notice the way the mean episode length drops down after the first increment. However, the result is similar to the results shown in Figure 8.12 (b)
- (c) For $\eta = 12$ the average performance is also similar to the results in Figure 8.12 (c).

In Figure 8.15 we can see how the three incremental steps overlap especially in early training. Although $\eta = 12$ underperformed a little bit under an extended duration comparing the two others.

There is a large gap between the threshold of 1500 and 2500 but the experiment results are fairly similar on average. In the following section, we summarize the overall incremental experiments.

6. Experiment summary

We observed both expected and unexpected results during the experiments. We observed that Incremental Learning allowed the PPO agent to solve the full environment. Based on the results of the experiments we made the following conclusions:

- The experiment results showed the effectiveness of Incremental Learning as discussed in Chapter 6.3. Environments with large state spaces can be solved by gradually

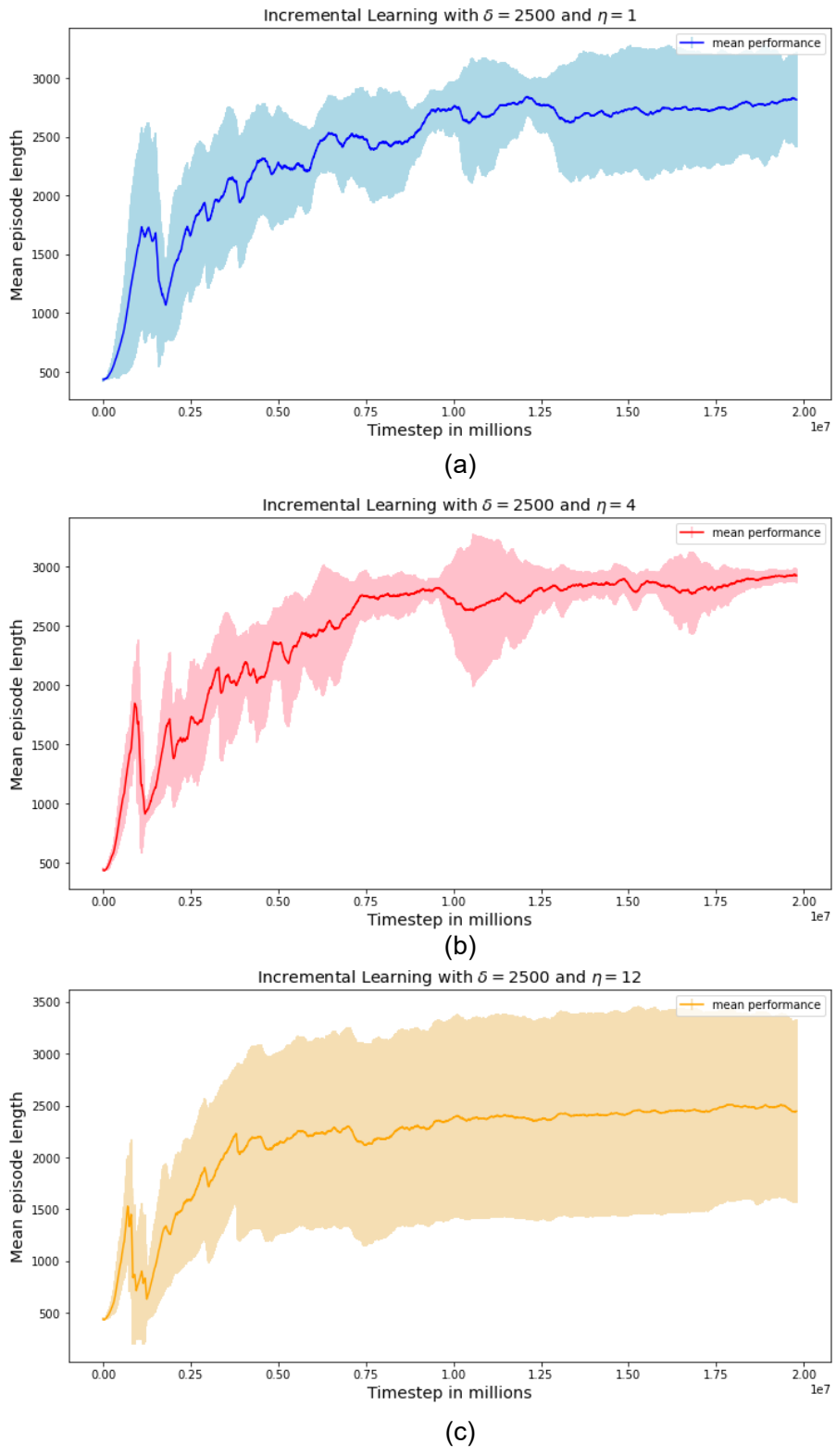


Figure 8.14: Incremental training progression of 10 different seeds with $\delta = 2500$. (a) incremental step $\eta = 1$; (b) incremental step $\eta = 4$; (c) incremental step $\eta = 12$.

increasing the size of the environment space MDP.

- The results shown in Figure 8.7 (a) confirm the discussion in Section 6.4 regarding

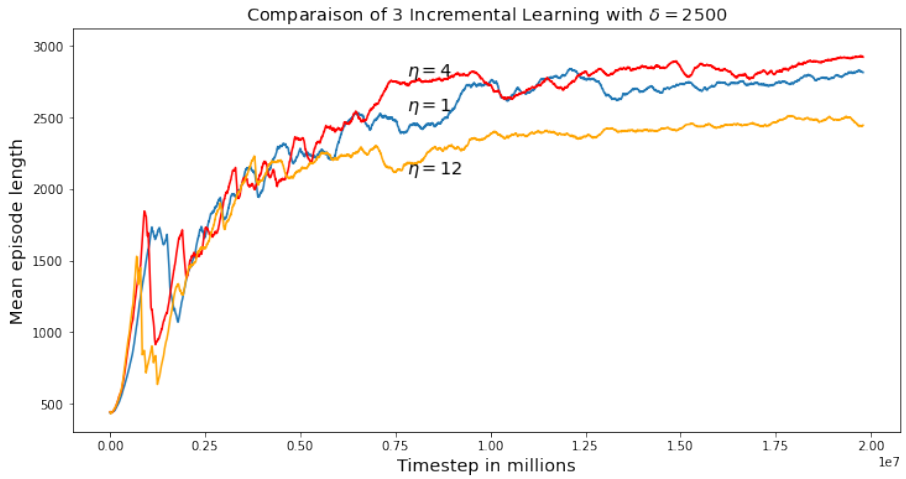


Figure 8.15: Comparison between three different incremental steps and with the same threshold $\delta = 2500$.

the effectiveness of using a threshold score in the incremental process. For a fast learning experience, agents can target a sub-optimal policy to perform an increment.

- Results in Figure 8.8 (c) confirms the discussion in Sections 6.4 and 6.4 regarding the negative transfer effect of using a small threshold and a wide incremental step. However, Figure 8.12 (c) and Figure 8.14 (c) show that with a high-performance threshold, the training is less sensitive to big incremental steps.

- Figure 8.15 shows that with a high-performance threshold, big incremental steps have slow adaptation on average. However, selecting the smallest as possible does not guarantee the best training experience. For example, we can see in the plot that steps 1 with 24 increments and step 4 with just 6 increments have similar average performances such that we can not conclude which of the two is optimal.

We have been able to apply Incremental Learning to our problem. The agent had two incremental problems to solve, the opponent was getting stronger each time a new best model was saved, and the environment was getting more complex each time a performance threshold was reached. The overall training has shown that there is no optimal combination of (δ, η) that results in the best training experience because the model initialization has a role to play in the adaptation ability of the agent. But we can conclude that a high-performance threshold may lead on average to more efficient training regardless of the initialization.

We have shown the success of Incremental RL when training a single agent. However, the main objective is to train a team of Independent PPO agents to solve the 3D environment in a cooperative scenario. In the next section, we present experiments using Incremental Learning and Self-Play in a challenging Multi-Agent setting.

8.2. Multi-Agent Experiment

Recall the objective of the project was to explore the training of two independent agents collaborating as a team in a dynamic environment. Referring to Chapter 2.2, we customized a team of two independent PPO agents to collaborate in the slimebot volleyball game environment. As an initial experiment, we start by training the team in 2D similarly with the single agent in section 8.1.

8.2.1. Training in a 2D Projected Space

Similarly to the experiment Section 8.1, agents can only move inline, and the $z-$axis is frozen. We performed two different experiments; (1) teammates optimize the same objective function; (2) The reward function is altered generating individual reward functions.

1. Common Objective Function

In this scenario, the game keeps its standard structure similarly to the single-agent version. The two agents share the same reward function and the same lives, and they are playing in a Self-Play mode. This means each team member has an equivalent in the adversarial team.

The result in Figure 8.16 (a) showed that In fewer than 20 million timesteps, the PPO team working independently successfully achieved an optimal episode length solution for the 2D game. However, in Figure 8.17 (a) we notice a big difference between the number of times both agents hit the ball on average. The number of times the blue agent collides with the ball decreases significantly while increasing for the orange agent. This means the orange agent is the one generating progress for the team, and the blue agent is probably stuck in a corner of the game scene as illustrated in the section A. Figure 8.17 (b) shows that, how many times on average each agent was leaving its initial corner to visit the teammate's corner during the training. We can notice that the graph of the blue agent vanishes to zero over time. This means the blue agent has stopped visiting the orange agent's corner and stayed stuck in its corner. However, the orange agent was still visiting the blue agent's corner to follow the ball. This means the team has fallen into the situation of selfish and lazy agents that we discussed in Chapter 2.2, where the blue agent is lazy and the orange agent is selfish.

We can not say that this is a failure because in real-life scenarios members of the same team do not always have the same level of expertise. There is often at least one member whose entire team will feel the impact of their absence. However, this does not fulfill our objective of having cooperative training. We introduce in the next section an alternative to address the problem and motivate equity of training within the team.

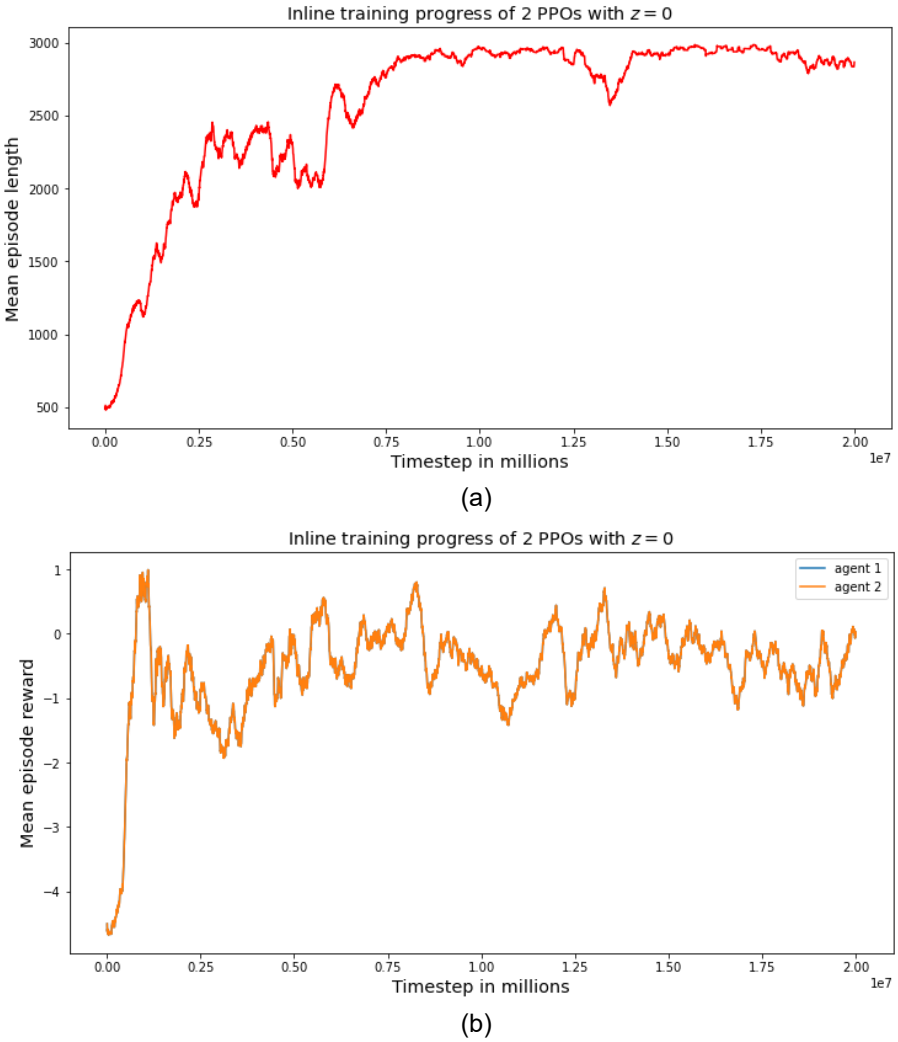


Figure 8.16: Training progress of a team of two Independent PPOs in 2D. (a) The mean episode length. (b) The reward function is the same for both agents as they are sharing the same objective function. Both agents' graphs are overlapping in the plot.

2. Shaped Reward Function

In Section 2.3 we discussed that Reward Shaping is a good approach in a multi-agent system to ensure credit assignment for individual behavior. We used a simple additive version of Reward Shaping that assigned a particular objective function for each team member. Figure 8.18 gives an illustration of how the environment reward can be altered to ensure credit assignment, where the team area in the game scene is split into 2 regions, and each region is the responsibility of one team member.

We tried a weak reward shaping[7] in the experiments. In addition to the main environment reward, each agent receives a bonus of 0.001 if it stays in its region of the game scene, or a penalty of 0.01 if it goes into the teammate's region. Additionally, an agent receives a penalty of 0.05 if they collide with their teammate while being in the teammate's region.

[7]Weak because it barely changes the main environment reward.

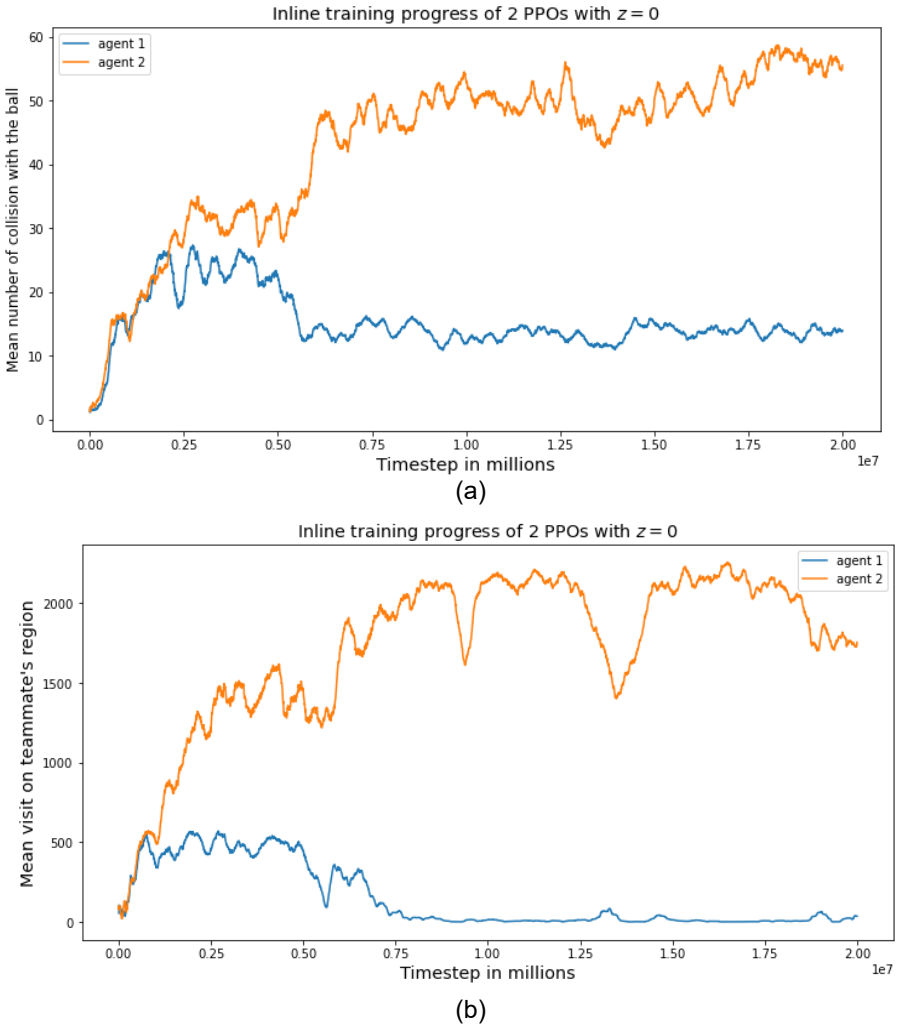


Figure 8.17: Training progress of a team of two Independent PPOs in 2D. (a) The average number of times they collide with the ball during an episode. (b) As agents are placed in line, one in the front and the other one at the back, the plot shows how many times on average they follow the ball up to the teammate's side.

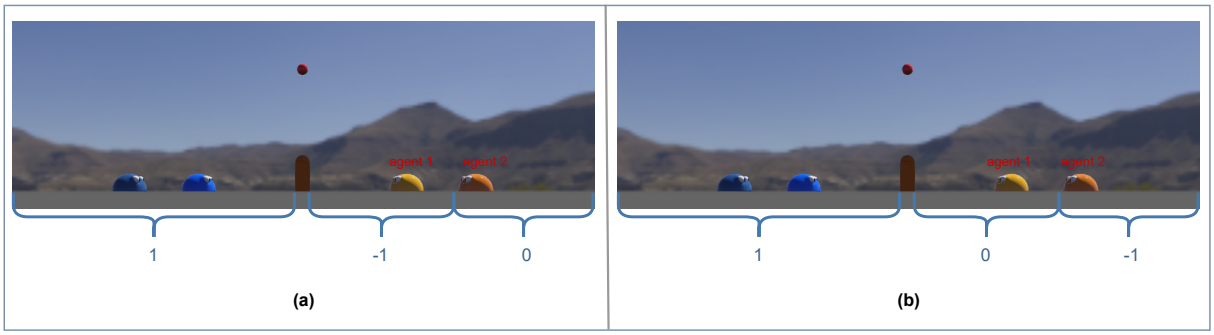


Figure 8.18: Reward shaping illustration, the numbers below the game area indicate the agents' reward when the ball hits the ground:(a) Reward of Agent 1 if the ball hits the ground. (b) The reward of Agent 2 when the ball hits the ground.

The experiment results in Figure 8.19 (a) show that the team has been able to solve the game close to the optimal episode length within 20 million timesteps, and in Figure 8.19

(b) observable dissimilarities in the shapes of the average rewards of the two agents can be attributed to differences in their respective reward functions. We notice the effectiveness of the approach in Figure 8.20 (a) where both agents are equitably hitting the ball, this means that they are equivalently contributing to the team's performance. Furthermore, 8.20 (b) shows that both agents lose interest in spending too much time on the teammate's region. This shows that the altered reward functions helped demotivate the lazy and selfish phenomena.

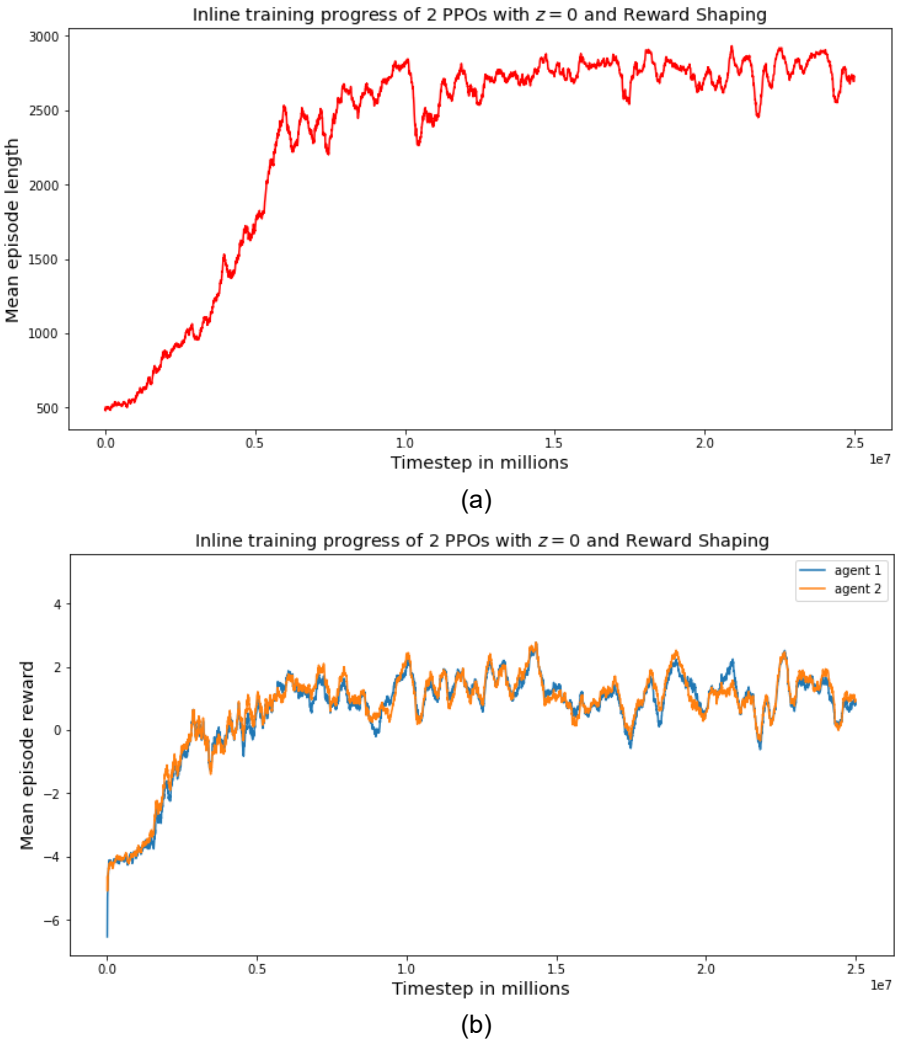


Figure 8.19: Training progress of a team of two Independent PPOs in 2D with a shaped reward function. (a) The mean episode length. (b) The average reward for the respective agent.

3. Experiment summary

The are many forms of reward shaping that we did not investigate to make the team collaborate better. We just illustrated one among multiple possibilities. We have shown that the team of independent PPOs can collaborate to play the game in 2D almost perfectly. We thus decided to take the problem further by training them in 3D using Incremental

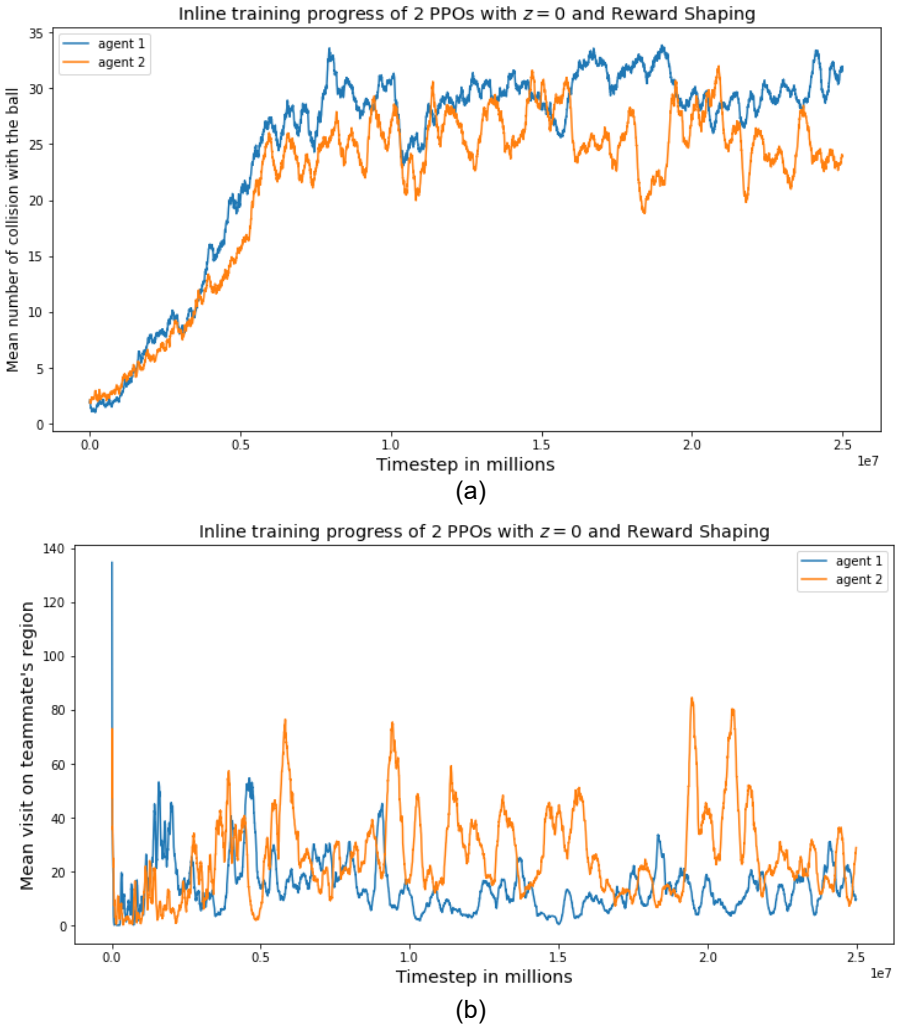


Figure 8.20: Training progress of a team of two Independent PPOs in 2D with a shaped reward function. (a) The mean number of timesteps agents collide with the ball during an episode. (b) The plot shows how many timesteps on average each agent follows the ball up to the teammate's region.

Learning (as in the single agent experiment). The experiment results are discussed in the next session.

8.2.2. Training in Incremental Environment

Recall the result of the experiments in Section 8.1 which showed that the PPO algorithm can incrementally learn to solve environments with increasing MDP. Referring also to the previous section result, where Independent PPOs solved the 2D version of the game. We discuss in this section the results of experiments using Incremental Learning in a multi-agent setting.

In this series of experiments, the challenge is more difficult than it has been in all previous experiments. Each team member has to collaborate with a non-stationary teammate in a

non-stationary environment with a non-stationary opponent team. We present results of mixing Multi-Agent RL, Reward Shaping, Incremental RL, and Self-Play Learning in the same experiment.

1. Common Objective Function in 3D

Recall the experiment in Section 8.2 which generated the lazy and selfish agents in 2D. We repeat the same experiment of a shared objective function along with Incremental Learning to investigate each team member's behavior in an incremental scenario. For this experiment, we used a threshold of $\delta = 1200$ and an incremental step $\eta = 1$.

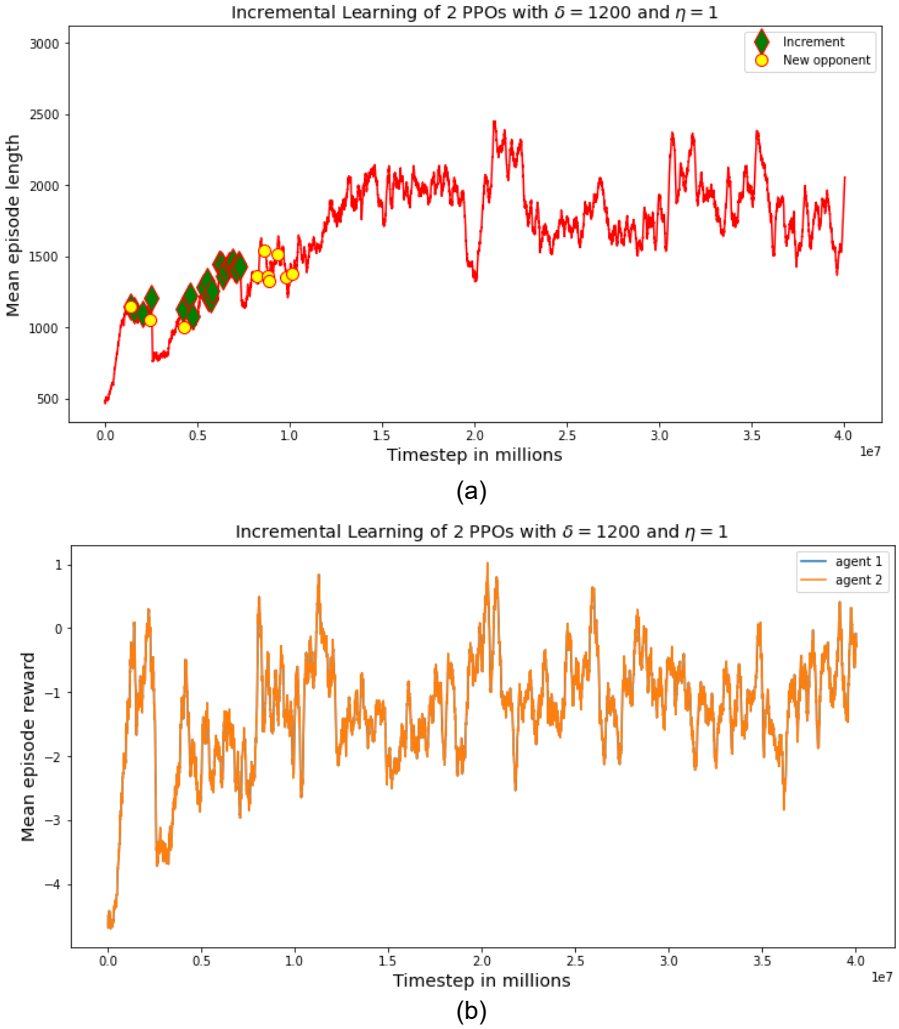


Figure 8.21: Incremental Training progress of a team of two Independent PPOs in 3D. (a) The mean episode length with indications of increment. (b) The average reward signal is the same for both agents.

As expected, Figure 8.21 shows a similar result as Section 8.2. Although the team managed to reach the depth of 24, we can see in 8.21 (a) that the team didn't manage to approximate the maximum average episode length. The team performance fluctuates between 1500 and 2000 after 40 million timesteps. In (b) the fluctuation of the reward was expected to be

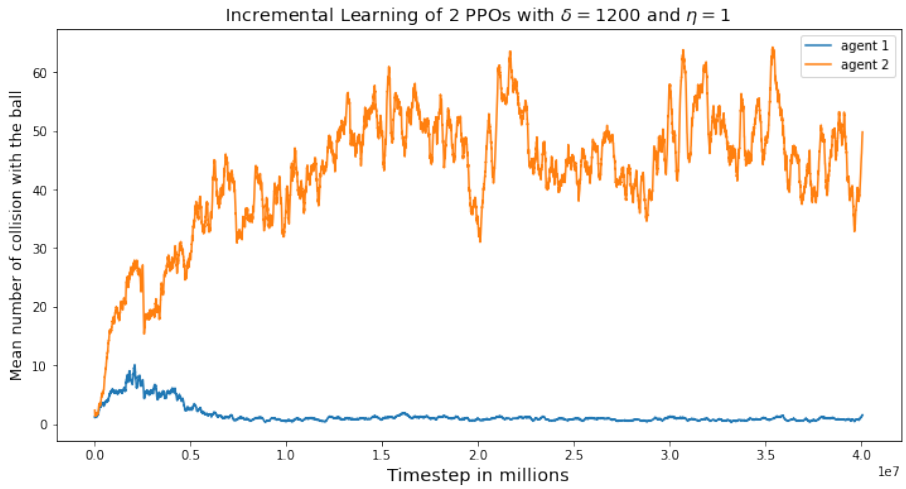


Figure 8.22: Incremental Training progress of a team of two Independent PPOs in 3D according to the average number of times they collide with the ball per episode.

centered around 0, as we are in a Self-Play training mode. However, it stayed lower and barely got close to zero. In Figure 8.22 we notice that the blue agent was lazy and the orange agent was selfish during the training. The orange agent was the one generating the increment.

As the results were not unexpected, we proceeded to examine the efficacy of incorporating reward shaping into the training process.

2. Reward Shaping in 3D

We used the same reward shaping setting of the experiment in Section 8.2.

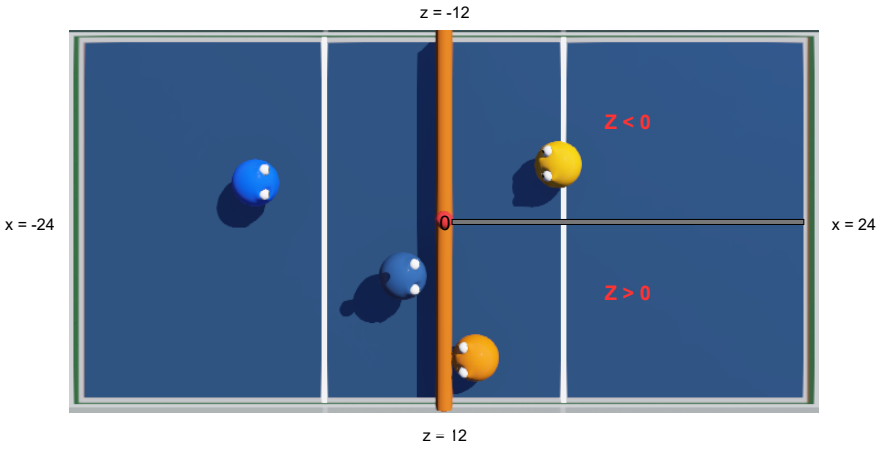


Figure 8.23: Top view illustrating a subdivision of the 3D environment applicable to reward shaping.

However, the team area is divided into two regions The agents are split into two regions according to the $z-$axis[8], as shown in Figure 8.23 of the team member is responsible for

[8]In Webots the $y-$axis is the height.

the area with $z < 0$ and the other is responsible for the area with $z \geq 0$. Both agents receive an additional reward of 0.01 if they stay on their respective sides, and a penalty of 0.001 if they step on the teammate's area. They also get an additional penalty of 0.05 if a collision occurs in the teammate's region.

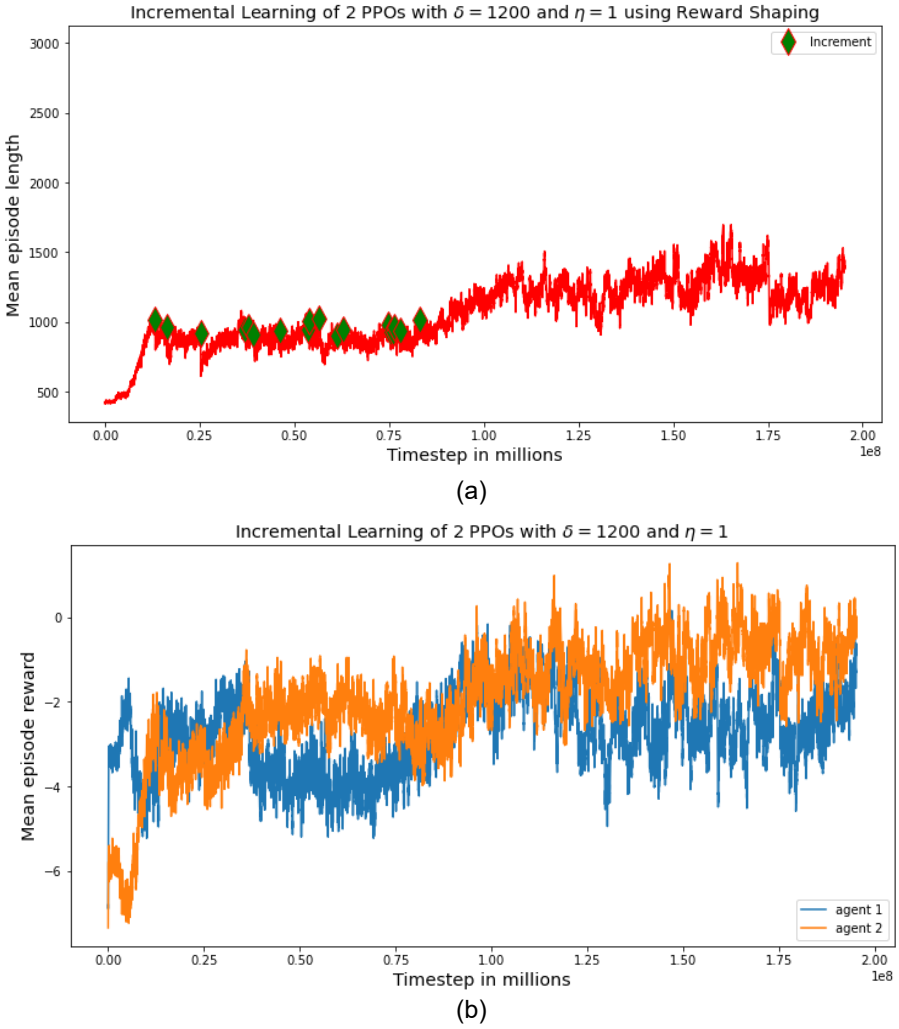


Figure 8.24: Incremental Training progress of the average episode length and reward of two Independent PPOs in 3D with $\delta = 1200$, $\eta = 1$ and using reward shaping. (a) The mean episode length with the indication of increment. (b) The average reward signal is the same for both agents.

Let's start with Figure 8.25 (b) which shows how quickly both agents stopped moving to the prohibited areas. And Figure 8.25 (a) shows that on average both agents kept the same performance in terms of collisions with the ball. Figure 8.24 (b) shows that their reward was similar throughout the training horizon. However, in Figure 8.24 (a) we notice that although the team managed to increment up to the depth of 24, after reaching the full environment space, the average lifetime in the game didn't progress up to the optimum with a training horizon close to 200 million timesteps. It stayed below the average. We assume that agents needed a longer training horizon according to the progress of the

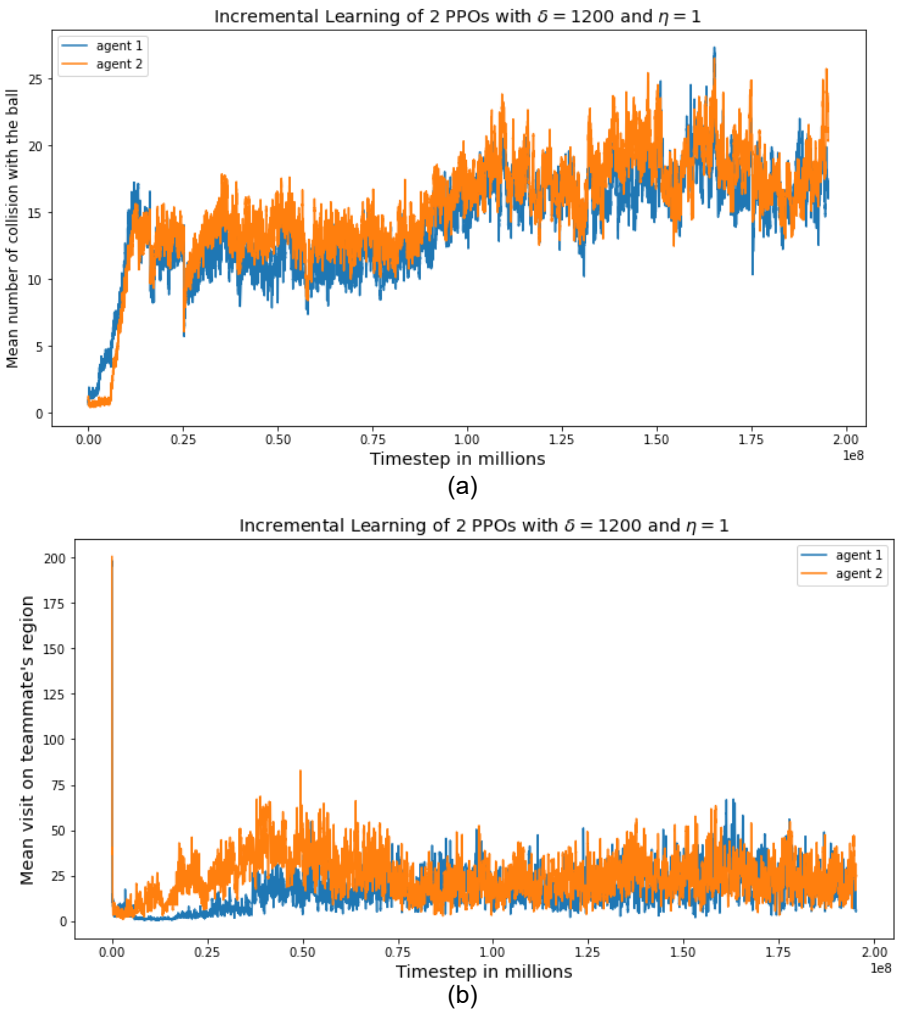


Figure 8.25: Incremental Training progress of two Independent PPOs in 3D with $\delta = 1200$, $\eta = 1$ and using reward shaping (a) The average number of times they collide with the ball per episode. (b) Shows how on average they move along the x-axis depending on their initial position.

average episode length.

3. Experiment Summary

This series of experiments brought similar results to the experiments of the previous section. We have seen that the team of PPO agents managed to play the game in the full 3D environment. However, they could not make it to the optimal lifetime of the game. The subplot (a) and (b) of both Figure 8.21 and Figure 8.24 show how unstable and noisy the progress of the average episode length and average reward were along the training. This can be explained by the non-stationary nature of the problem we mentioned in Chapter 2.2. Especially in the incremental experiment with Reward Shaping, it was challenging for agents to adapt to the teammate's behavior after each increment. The non-stationary was not a problem in Chapter 8.2 because the environment dynamic was not as complex as it was in the 3D case.

8.3. Summery

This chapter discussed the evaluation of experiments we did to achieve the project objective. We experimented with cooperative training of independent PPO agents in a 3D volleyball game. As an initial experiment, we investigated the ability of a single PPO agent to solve the game environment in a single competitive scenario. However, the result in Section 8.1 showed that the task was complex to solve from scratch by the PPO agent. The investigation in Section 8.1 showed that Incremental Learning is an effective approach to solving the problem by gradually increasing the state space of the game while starting with a small portion of the state space. Facing a lack of benchmark models in the environment, we used Self-Play Leaning as a training paradigm with Incremental Learning. Furthermore, we investigated the combined effects the performance threshold and the incremental step have on the training. The results showed that their combined effects depend on the initialization of the agent model. But we saw that the success of training is less sensitive to a combination of a high threshold with a large incremental step brings.

Based on the result in 8.1 with the single agent scenario, we investigated in Section 8.2 on the training in a multi-agent setting. We investigated the 2D training as a testbed experiment to observe collaboration in a simple scenario. The results showed that sharing the same objective function generates a lazy and a selfish agent. However, altering the original reward function permits improvement for each team member. Based on the testbed result, we investigated the team's ability to solve the task incrementally. The results showed that with a shared objective function, the team managed to increment to the full environment. However, it also generates a lazy and selfish agent, where the selfish agent is the one generating the increment. It also showed that with appropriate reward shaping both team members managed to increment to the full environment with similar improvements. We should point out that in both incremental scenarios, the team performance did not get close to the optimal average episode length after a very long training horizon.

Chapter 9

Conclusion

In this thesis, we investigated the ability of Reinforcement Learning (RL) algorithms to learn collaboration in a dynamic environment with continuous state space. The task we aimed to solve is a cooperative volleyball game in 3D [9]. However, after observing the failure of the Proximal Policy Optimization (PPO) algorithm to solve the game in a single-agent scenario, we investigated applying Incremental RL [14, 15] as a learning paradigm to solve the task in both single and multi-agent scenarios. We also refer to additional techniques such as Self-Play Learning [29] because the experimental environment lacks a benchmark expert to train agents with, and Reward Shaping [21, 22, 23, 24] which is an effective approach to address issues such as the lazy and selfish agent phenomenon, and sparse rewards of episodic tasks.

9.1. Summary

This project focused on the effectiveness of Incremental RL in solving problems with large MDPs. We started by investigating the application of Incremental RL in a competitive single-agent scenario. Our investigation showed that agents can learn a complex task by starting simply and gradually increasing the difficulty of the task. Particularly, as discussed in Section 6.3, the investigation showed agents can solve a large MDP, by breaking down the MDP into a sequence of small MDPs where each previously solved MDP is included in the next MDP in the sequence. This approach confirmed the discussion in 6.4, that agents can learn sub-optimal policies on sub-task by referring to a performance threshold, and still benefit from an optimal policy at the final and main task that generalizes over all previous sub-tasks in the sequence.

Using a threshold to move between tasks in the sequence led us to investigate the influence the selected threshold and the incremental step may have on the training. We assumed that the confluence of the two parameters could potentially engender significant effects on the training process. We investigated to see if there is an optimal combination that expedites incremental learning. The experimental evaluation investigated in Section 8.1

showed that there is no optimal combination between the threshold and incremental step because each model initialization brings a particular training experience for the same combination. However, we have concluded that a combination of a high threshold and a large incremental step has a high potentiality of adapting to an increment.

We finally investigated the ability of a team of two PPOs to solve the environment using Incremental RL. Agents in the team learned decentralized policies while sharing the same objective function. The investigation in Section 8.2 showed that the team managed to increment to the full environment. However, it generated a lazy agent stuck in a corner and a selfish agent that quickly found the optimal behavior to generate positive feedback for the team. Further investigation done in Section 8.2 showed that Reward Shaping is an effective approach that permits efficient decentralized training. The experiments showed that altering the original reward function can effectively motivate equitable improvement among teammates.

9.2. Future Work

We were expecting optimal behaviors in the cooperative scenario. However, the team just reached a sub-optimal performance in the game. We assume that the task was more complex for each teammate, as the training process involved Incremental Learning which requires each agent to adapt to a non-stationary environment and a non-stationary teammate. We believe that employing a more rigorous experimental design can yield superior outcomes in contrast to the ones that were observed. If time permitted we could have tried the following methods:

- Turn-Based learning: We assume that keeping one team member stationary while the other is training, and reversing the role according to a given condition, may effectively encourage adaptation between teammates' behavior.

- Prioritized experience replay: In our experiments, agents used the same historical experience for their respective gradient updates. We assume that by including individual Prioritized Experience Replay [113], discussed in Section 5.1, each agent can benefit from relevant experience for individual efficient learning.

- Parallel Training: We limited our investigation of cooperative Incremental Leaning with a single initialization training. As discussed in Section 2.1, with Multi-task RL [39], parallel training can be an effective approach if multiple environments run at the same time, and similar to the A3C (asynchronous advantage actor-critic) [40], after each iteration, the best model of each team member is used as initialization for the next iteration.

We investigated the problem using state observation, where each agent receives relevant information regarding the states of the environment. However, the simulator used in the project permits image processing from a camera placed on each agent. Using a camera for state observation makes the problem more realistic as the state space will be partially observable from each agent's perspective. This will imply additional techniques such as communication between teammates for effective learning.

Many scenarios can be developed in the experimental environment, such as having more than 2 members per team. Due to time, we limit our project to the scenarios presented in the experimental chapter.

Bibliography

[1] Y. Fenjiro and H. Benbrahim, "Deep reinforcement learning overview of the state of the art," *Journal of Automation Mobile Robotics and Intelligent Systems*, vol. 12, no. 3, pp. 20–39, 2018.

[2] A. Lazaridis, A. Fachantidis, and I. Vlahavas, "Deep reinforcement learning: A state-of-the-art walkthrough," *Journal of Artificial Intelligence Research*, vol. 69, pp. 1421–1471, 2020.

[3] J. Markovska and D. Šoberl, "Deep reinforcement learning compared to human performance in playing video games," 2022.

[4] G. Joshi and G. Chowdhary, "Adaptive policy transfer in reinforcement learning," *arXiv preprint arXiv:2105.04699*, 2021.

[5] W. Fu, C. Yu, Z. Xu, J. Yang, and Y. Wu, "Revisiting some common practices in cooperative multi-agent reinforcement learning," *arXiv preprint arXiv:2206.07505*, 2022.

[6] C. Yu, A. Velu, E. Vinitsky, Y. Wang, A. Bayen, and Y. Wu, "The surprising effectiveness of ppo in cooperative, multi-agent games," *arXiv preprint arXiv:2103.01955*, 2021.

[7] C. S. de Witt, T. Gupta, D. Makoviichuk, V. Makoviychuk, P. H. Torr, M. Sun, and S. Whiteson, "Is independent learning all you need in the starcraft multi-agent challenge?" *arXiv preprint arXiv:2011.09533*, 2020.

[8] M. Samvelyan, T. Rashid, C. S. De Witt, G. Farquhar, N. Nardelli, T. G. Rudner, C.-M. Hung, P. H. Torr, J. Foerster, and S. Whiteson, "The starcraft multi-agent challenge," *arXiv preprint arXiv:1902.04043*, 2019.

[9] J. Bakambana, "Slimebot volleyball: A multi-agents 3d gym environment for slime volleyball game," https://github.com/jbakams/slimebot-volleyball, 2022.

[10] D. Ha, "Slime volleyball gym environment," https://github.com/hardmaru/slimevolleygym, 2020.

[11] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[12] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone, "Curriculum learning for reinforcement learning domains: A framework and survey," *arXiv preprint arXiv:2003.04960*, 2020.

[13] Z. Bing, D. Lerch, K. Huang, and A. Knoll, "Meta-reinforcement learning in non-stationary and dynamic environments," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[14] Z. Wang, C. Chen, H.-X. Li, D. Dong, and T.-J. Tarn, "Incremental reinforcement learning with prioritized sweeping for dynamic environments," *IEEE/ASME Transactions on Mechatronics*, vol. 24, no. 2, pp. 621–632, 2019.

[15] Z. Wang, H.-X. Li, and C. Chen, "Incremental reinforcement learning in continuous spaces via policy relaxation and importance weighting," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 6, pp. 1870–1883, 2019.

[16] K. Kim, Y. Gu, J. Song, S. Zhao, and S. Ermon, "Domain adaptive imitation learning," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5286–5295.

[17] W. Zhang, L. Deng, L. Zhang, and D. Wu, "A survey on negative transfer," *IEEE/CAA Journal of Automatica Sinica*, 2022.

[18] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[19] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *Journal of artificial intelligence research*, vol. 13, pp. 227–303, 2000.

[20] S. Kapetanakis and D. Kudenko, "Reinforcement learning of coordination in cooperative multi-agent systems," *AAAI/IAAI*, vol. 2002, pp. 326–331, 2002.

[21] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Icml*, vol. 99, 1999, pp. 278–287.

[22] Y. Hu, W. Wang, H. Jia, Y. Wang, Y. Chen, J. Hao, F. Wu, and C. Fan, "Learning to utilize shaping rewards: A new approach of reward shaping," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 931–15 941, 2020.

[23] E. Wiewiora, G. W. Cottrell, and C. Elkan, "Principled methods for advising reinforcement learning agents," in *Proceedings of the 20th international conference on machine learning (ICML-03)*, 2003, pp. 792–799.

[24] T. Brys, A. Harutyunyan, H. B. Suay, S. Chernova, M. E. Taylor, and A. Nowé, "Reinforcement learning from demonstration through shaping," in *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[25] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 1.

[26] B. Price and C. Boutilier, "Accelerating reinforcement learning through implicit imitation," *Journal of Artificial Intelligence Research*, vol. 19, pp. 569–629, 2003.

[27] B. D. Ziebart, A. L. Maas, J. A. Bagnell, A. K. Dey *et al.*, "Maximum entropy inverse reinforcement learning." in *Aaai*, vol. 8. Chicago, IL, USA, 2008, pp. 1433–1438.

[28] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2011, pp. 627–635.

[29] A. DiGiovanni and E. C. Zell, "Survey of self-play in reinforcement learning," *arXiv preprint arXiv:2107.02850*, 2021.

[30] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[31] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[32] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines," https://github.com/hill-a/stable-baselines, 2018.

[33] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey." *Journal of Machine Learning Research*, vol. 10, no. 7, 2009.

[34] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, C. Finn, and S. Levine, "Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning,"

in *Conference on robot learning*. PMLR, 2020, pp. 1094–1100.

[35] A. Kaplan, "Lifelong learning: conclusions from a literature review," *International Online Journal of Primary Education*, vol. 5, no. 2, pp. 43–50, 2016.

[36] Z. Wang, C. Chen, and D. Dong, "Lifelong incremental reinforcement learning with online bayesian inference," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[37] X. Zheng, C. Yu, and M. Zhang, "Lifelong reinforcement learning with temporal logic formulas and reward machines," *Knowledge-Based Systems*, vol. 257, p. 109650, 2022.

[38] E. C. Johnson, E. Q. Nguyen, B. Schreurs, C. S. Ewulum, C. Ashcraft, N. M. Fendley, M. M. Baker, A. New, and G. K. Vallabha, "L2explorer: A lifelong reinforcement learning assessment environment," *arXiv preprint arXiv:2203.07454*, 2022.

[39] S. Sodhani, A. Zhang, and J. Pineau, "Multi-task reinforcement learning with context-based representations," in *International Conference on Machine Learning*. PMLR, 2021, pp. 9767–9779.

[40] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[41] Y. Teh, V. Bapst, W. M. Czarnecki, J. Quan, J. Kirkpatrick, R. Hadsell, N. Heess, and R. Pascanu, "Distral: Robust multitask reinforcement learning," *Advances in neural information processing systems*, vol. 30, 2017.

[42] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *International conference on machine learning*. PMLR, 2018, pp. 1407–1416.

[43] M. Hessel, H. Soyer, L. Espeholt, W. Czarnecki, S. Schmitt, and H. van Hasselt, "Multi-task deep reinforcement learning with popart," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 3796–3803.

[44] N. Vithayathil Varghese and Q. H. Mahmoud, "A survey of multi-task deep reinforcement learning," *Electronics*, vol. 9, no. 9, p. 1363, 2020.

[45] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling, "Revisiting the arcade learning environment: Evaluation protocols and

open problems for general agents," *Journal of Artificial Intelligence Research*, vol. 61, pp. 523–562, 2018.

[46] P. Stone and M. Veloso, "Multiagent systems: A survey from a machine learning perspective," *Autonomous Robots*, vol. 8, no. 3, pp. 345–383, 2000.

[47] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative agents," in *Proceedings of the tenth international conference on machine learning*, 1993, pp. 330–337.

[48] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[49] A. Nowé, P. Vrancx, and Y.-M. D. Hauwere, "Game theory and multi-agent reinforcement learning," in *Reinforcement Learning*. Springer, 2012, pp. 441–470.

[50] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, "Multiagent cooperation and competition with deep reinforcement learning," *PloS one*, vol. 12, no. 4, p. e0172395, 2017.

[51] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[52] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls *et al.*, "Value-decomposition networks for cooperative multi-agent learning," *arXiv preprint arXiv:1706.05296*, 2017.

[53] T. Rashid, M. Samvelyan, C. Schroeder, G. Farquhar, J. Foerster, and S. Whiteson, "Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning," in *International conference on machine learning*. PMLR, 2018, pp. 4295–4304.

[54] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. Pieter Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *Advances in neural information processing systems*, vol. 30, 2017.

[55] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[56] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[57] A. Pretorius, K.-a. Tessera, A. P. Smit, C. Formanek, S. J. Grimbly, K. Eloff, S. Danisa, L. Francis, J. Shock, H. Kamper *et al.*, "Mava: a research framework for distributed multi-agent reinforcement learning," *arXiv preprint arXiv:2107.01460*, 2021.

[58] M. Samvelyan, T. Rashid, C. S. de Witt, G. Farquhar, N. Nardelli, T. G. J. Rudner, C.-M. Hung, P. H. S. Torr, J. Foerster, and S. Whiteson, "The StarCraft Multi-Agent Challenge," *CoRR*, vol. abs/1902.04043, 2019.

[59] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "RLlib: Abstractions for distributed reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2018.

[60] W. Li, X. Wang, B. Jin, J. Sheng, and H. Zha, "Dealing with non-stationarity in marl via trust-region decomposition," in *International Conference on Learning Representations*, 2021.

[61] Y. Zhuang, Y. Hu, and H. Wang, "Scalability of multiagent reinforcement learning," in *Interactions in Multiagent Systems*. World Scientific, 2019, pp. 1–17.

[62] L. Canese, G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Re, and S. Spanò, "Multi-agent reinforcement learning: A review of challenges and applications," *Applied Sciences*, vol. 11, no. 11, p. 4948, 2021.

[63] W. Mao, K. Zhang, E. Miehling, and T. Başar, "Information state embedding in partially observable cooperative multi-agent reinforcement learning," in *2020 59th IEEE Conference on Decision and Control (CDC)*. IEEE, 2020, pp. 6124–6131.

[64] H. Mao, Z. Gong, and Z. Xiao, "Reward design in cooperative multi-agent reinforcement learning for packet routing," *arXiv preprint arXiv:2003.03433*, 2020.

[65] F. Memarian, W. Goo, R. Lioutikov, S. Niekum, and U. Topcu, "Self-supervised online reward shaping in sparse-reward environments," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 2369–2375.

[66] I. ElSayed-Aly and L. Feng, "Logic-based reward shaping for multi-agent reinforcement learning," *arXiv preprint arXiv:2206.08881*, 2022.

[67] B. Xiao, B. Ramasubramanian, and R. Poovendran, "Shaping advice in deep reinforcement learning," *arXiv preprint arXiv:2202.09489*, 2022.

[68] D. Hadfield-Menell, S. Milli, P. Abbeel, S. J. Russell, and A. Dragan, "Inverse reward

design," *Advances in neural information processing systems*, vol. 30, 2017.

[69] S. Singh, R. L. Lewis, and A. G. Barto, "Where do rewards come from," in *Proceedings of the annual conference of the cognitive science society*. Cognitive Science Society, 2009, pp. 2601–2606.

[70] J. Sorg, R. L. Lewis, and S. Singh, "Reward design via online gradient ascent," *Advances in Neural Information Processing Systems*, vol. 23, 2010.

[71] A. Y. Ng, S. Russell *et al.*, "Algorithms for inverse reinforcement learning." in *Icml*, vol. 1, 2000, p. 2.

[72] F.-H. Hsu, *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton University Press, 2002.

[73] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[74] S. Sukhbaatar, Z. Lin, I. Kostrikov, G. Synnaeve, A. Szlam, and R. Fergus, "Intrinsic motivation and automatic curricula via asymmetric self-play," *arXiv preprint arXiv:1703.05407*, 2017.

[75] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent tool use from multi-agent autocurricula," *arXiv preprint arXiv:1909.07528*, 2019.

[76] M. Jaderberg, W. Czarnecki, I. Dunning, L. Marris, G. Lever, A. Castaneda *et al.*, "Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. arxiv," *arXiv preprint arXiv:1807.01281*, 2018.

[77] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[78] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[79] M. Szubert and W. Jaśkowski, "Temporal difference learning of n-tuple networks for the game 2048," in *2014 IEEE Conference on Computational Intelligence and Games*. IEEE, 2014, pp. 1–8.

[80] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A

survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[81] S. Ivanov and A. D'yakonov, "Modern deep reinforcement learning algorithms," *arXiv preprint arXiv:1906.10025*, 2019.

[82] L. Weng, "A (long) peek into reinforcement learning," *lilianweng.github.io*, 2018. [Online]. Available: https://lilianweng.github.io/posts/2018-02-19-rl-overview/

[83] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[84] S. Manju and M. Punithavalli, "An analysis of q-learning algorithms with strategies of reward function," *International Journal on Computer Science and Engineering*, vol. 3, no. 2, pp. 814–820, 2011.

[85] C. E. Shannon, "Xxii. programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, 1950.

[86] S. Padakandla, P. KJ, and S. Bhatnagar, "Reinforcement learning algorithm for non-stationary environments," *Applied Intelligence*, vol. 50, no. 11, pp. 3590–3606, 2020.

[87] G. E. Monahan, "State of the art—a survey of partially observable markov decision processes: theory, models, and algorithms," *Management science*, vol. 28, no. 1, pp. 1–16, 1982.

[88] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Machine learning proceedings 1994*. Elsevier, 1994, pp. 157–163.

[89] Z. Zhang, Y.-S. Ong, D. Wang, and B. Xue, "A collaborative multiagent reinforcement learning method based on policy gradient potential," *IEEE transactions on cybernetics*, vol. 51, no. 2, pp. 1015–1027, 2019.

[90] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[91] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[92] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[93] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.

[94] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[95] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[96] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.

[97] A. L. Maas, A. Y. Hannun, A. Y. Ng *et al.*, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, no. 1. Atlanta, Georgia, USA, 2013, p. 3.

[98] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," *Advances in neural information processing systems*, vol. 30, 2017.

[99] H. B. Curry, "The method of steepest descent for non-linear minimization problems," *Quarterly of Applied Mathematics*, vol. 2, no. 3, pp. 258–261, 1944.

[100] A. Ng and K. Katanforoosh, "Cs229 lecture notes deep learning," 2018.

[101] L. Bottou *et al.*, "Online learning and stochastic approximations," *On-line learning in neural networks*, vol. 17, no. 9, p. 142, 1998.

[102] V. Vapnik, *Estimation of dependences based on empirical data*. Springer Science & Business Media, 2006.

[103] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*. PMLR, 2013, pp. 1139–1147.

[104] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[105] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research*, vol. 12, no. 7, 2011.

[106] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," *Cited on*, vol. 14, no. 8, p. 2, 2012.

[107] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[108] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[109] X. Wang, S. Wang, X. Liang, D. Zhao, J. Huang, X. Xu, B. Dai, and Q. Miao, "Deep reinforcement learning: a survey," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[110] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[111] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[112] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[113] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[114] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1995–2003.

[115] T. Simonini, "Improvements in deep q learning: Dueling double dqn, prioritized experience replay, and fixed..." *freeCodeCamp.org*, Apr 2018.

[116] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, 1999.

[117] L. Weng, "Policy gradient algorithms," *lilianweng.github.io*, 2018. [Online]. Available: https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

[118] R. Willianms, "Toward a theory of reinforcement-learning connectionist systems,"

Technical Report NU-CCS-88-3, Northeastern University, 1988.

[119] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Advances in neural information processing systems*, vol. 12, 1999.

[120] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[121] J. M. Joyce, "Kullback-leibler divergence," in *International encyclopedia of statistical science*. Springer, 2011, pp. 720–722.

[122] A. Kwiatkowski, E. Alvarado, V. Kalogeiton, C. K. Liu, J. Pettré, M. van de Panne, and M.-P. Cani, "A survey on reinforcement learning methods in character animation," in *Computer Graphics Forum*, vol. 41, no. 2. Wiley Online Library, 2022, pp. 613–639.

[123] C. C.-Y. Hsu, C. Mendler-Dünner, and M. Hardt, "Revisiting design choices in proximal policy optimization," *arXiv preprint arXiv:2009.10897*, 2020.

[124] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

Appendix A

Links to Videos of Agents Trained during the Experiments

We provide links to videos that evaluate agents we trained during the experiment from the inline single agent training to the cooperative training in 3D:

- Evaluation of agents trained in the 2D projection of the full environment, discussed in Section 8.1.
- Illustration of Incrmental Learning with a incremental step $\eta = 4$.
- Evaluation of 2 PPO agents trained with different combination of (δ, η). The yellow agent was trained with the combination $(500, 1)$, the blue agent with the combination $(2500, 12)$.
- Illustration of the lazy and selfish agent phenomena in 2D generated by decentralized training of shared objective function discussed in Section 8.2.
- Illustration of the same situation as previously after successful increment in 3D.
- Illustration of effectiveness of Reward Shaping to motivate cooperation in 2D.
- Illustration of effectiveness of Reward Shaping to motivate cooperation after a successful incremental training.

Appendix B

Additional Results

Here we show graphs of different experiments we referred to on average the influence the threshold δ and the incremental step η have one another in Section 8.1.

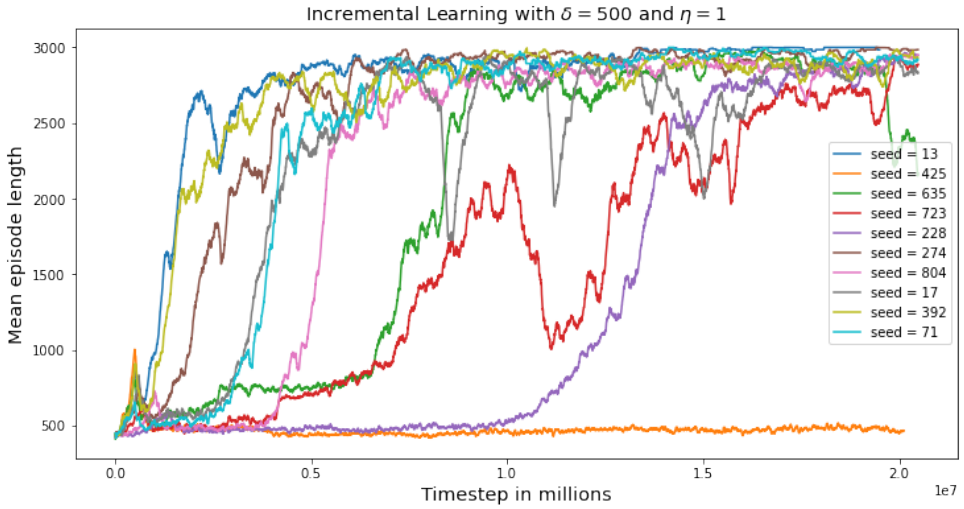


Figure B.1: Training progress of 10 different initialization with $\delta = 500$ and $\eta = 1$ within 20 million timesteps. That have been averaged in Figure 8.8 (a)

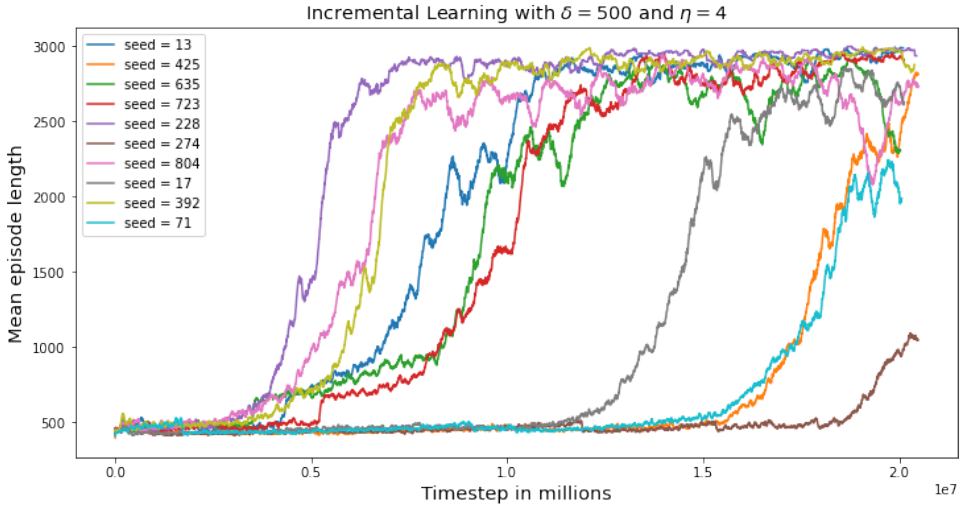


Figure B.2: Training progress of 10 different initialization with $\delta = 500$ and $\eta = 4$ within 20 million timesteps. That have been averaged in Figure 8.8 (b)

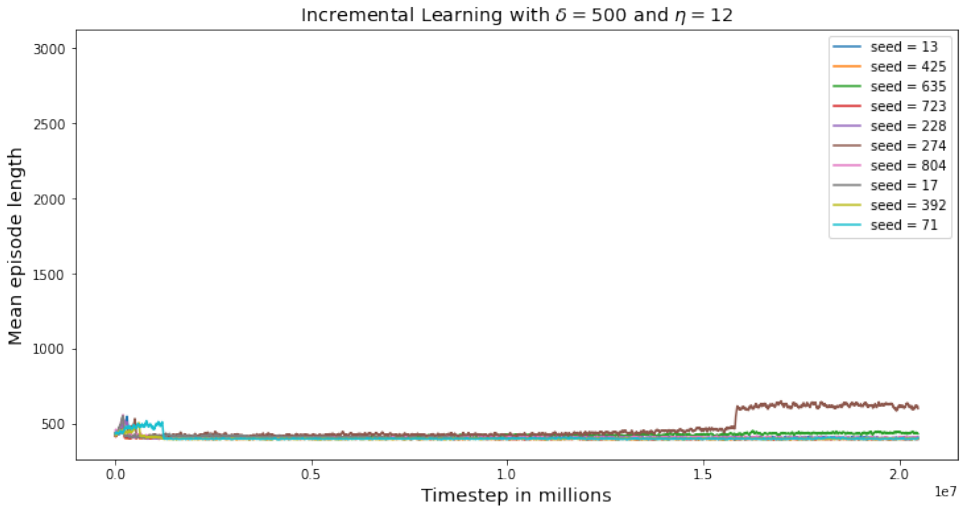


Figure B.3: Training progress of 10 different initialization with $\delta = 500$ and $\eta = 12$ within 20 million timesteps. That have been averaged in Figure 8.8 (c)

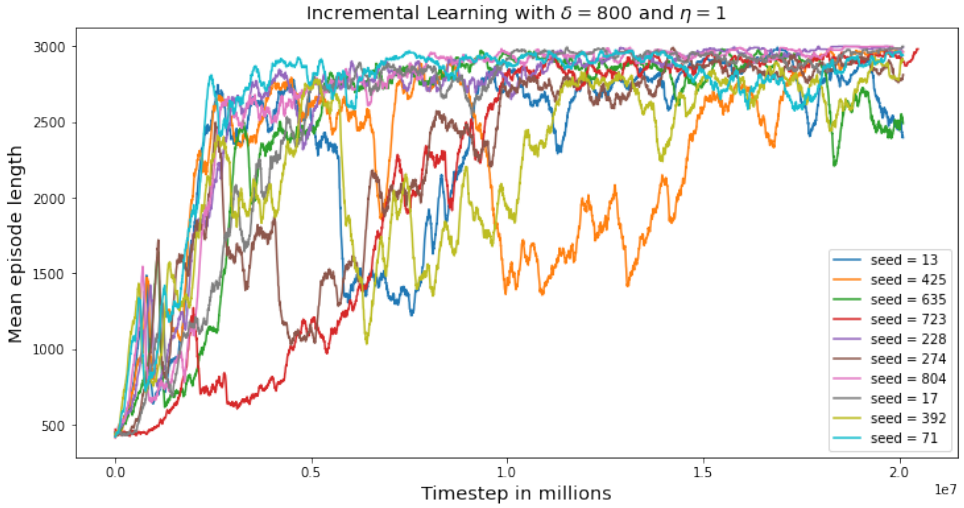


Figure B.4: Training progress of 10 different initialization with $\delta = 800$ and $\eta = 1$ within 20 million timesteps. That have been averaged in Figure 8.10 (a)

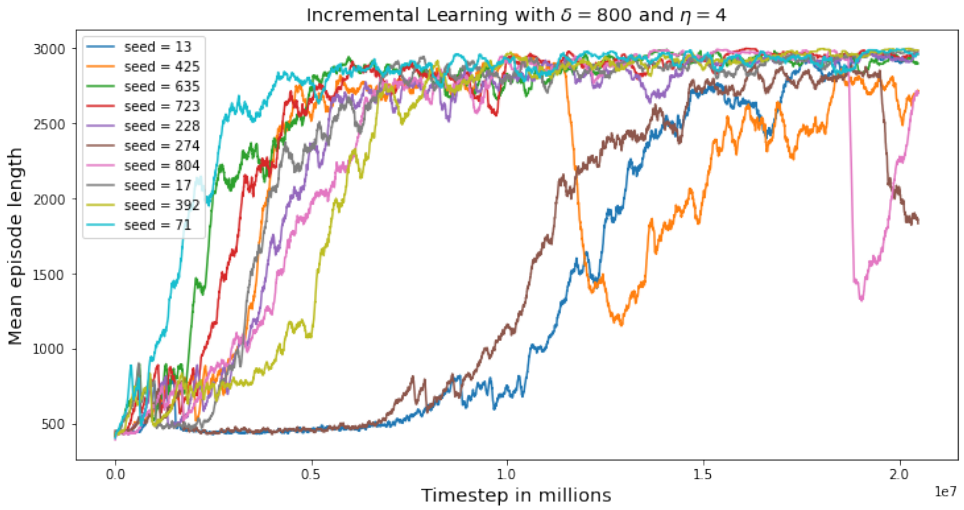


Figure B.5: Training progress of 10 different initialization with $\delta = 800$ and $\eta = 4$ within 20 million timesteps. That have been averaged in Figure 8.10 (b)

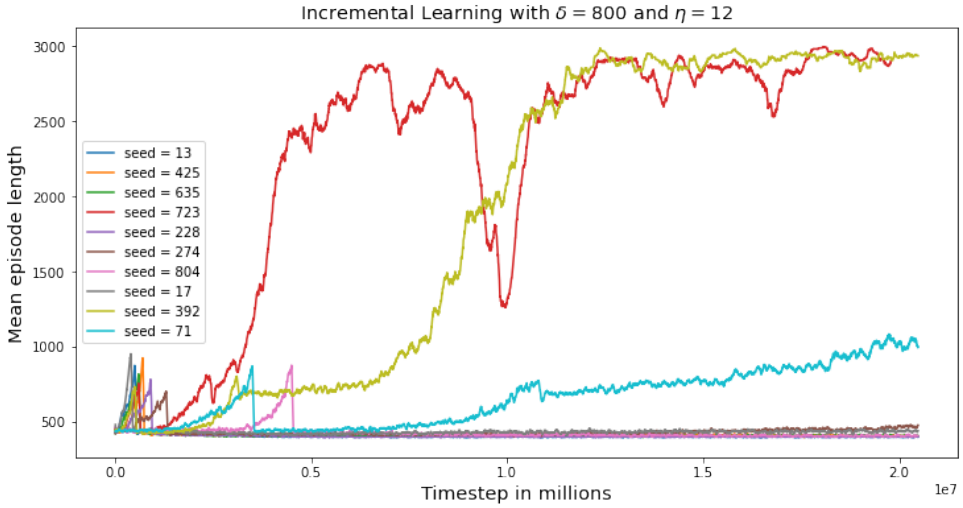


Figure B.6: Training progress of 10 different initialization with $\delta = 800$ and $\eta = 12$ within 20 million timesteps. That have been averaged in Figure 8.10 (c)

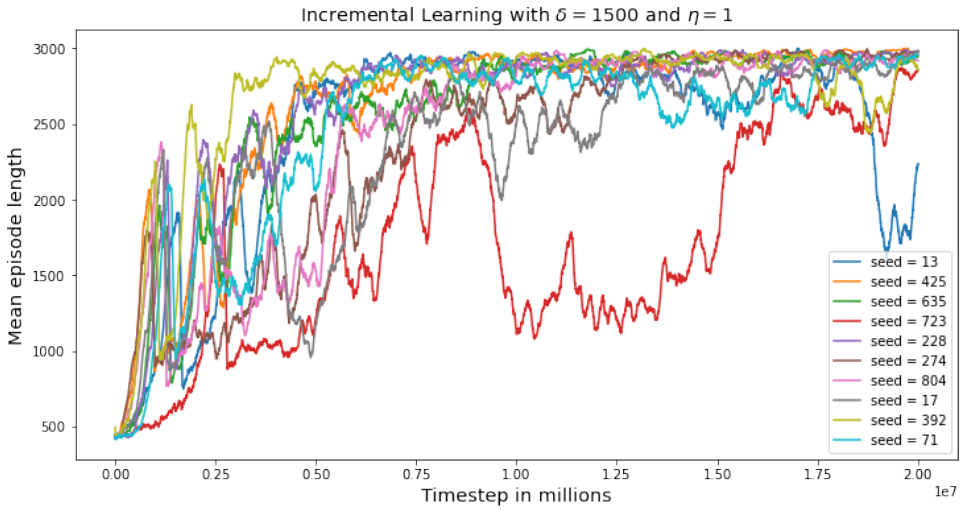


Figure B.7: Training progress of 10 different initialization with $\delta = 1500$ and $\eta = 1$ within 20 million timesteps. That have been averaged in Figure 8.12 (a)

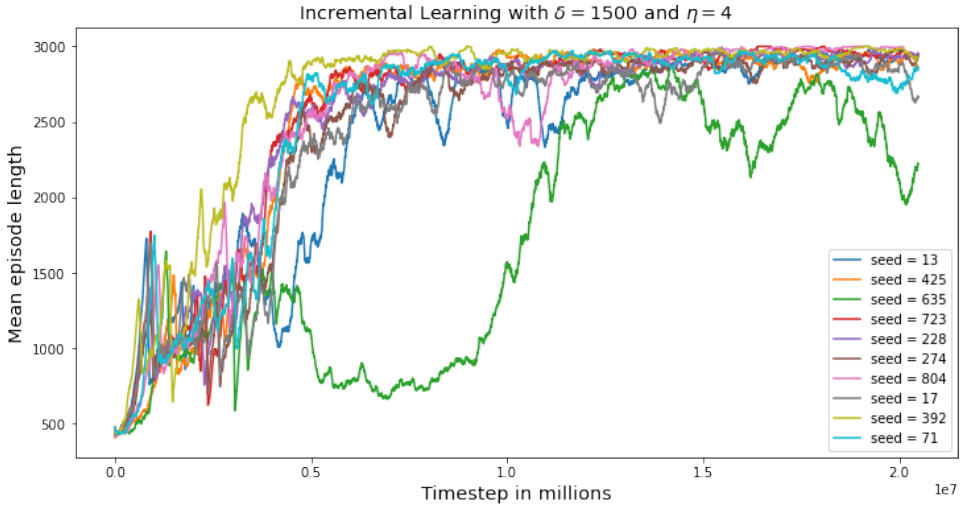


Figure B.8: Training progress of 10 different initialization with $\delta = 1500$ and $\eta = 4$ within 20 million timesteps. That have been averaged in Figure 8.12 (b)

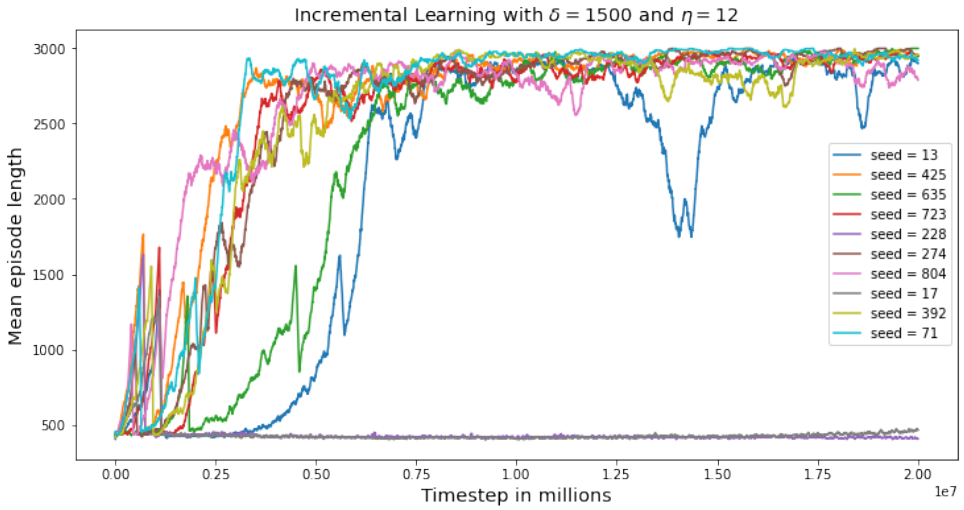


Figure B.9: Training progress of 10 different initialization with $\delta = 1500$ and $\eta = 12$ within 20 million timesteps. That have been averaged in Figure 8.12 (c)

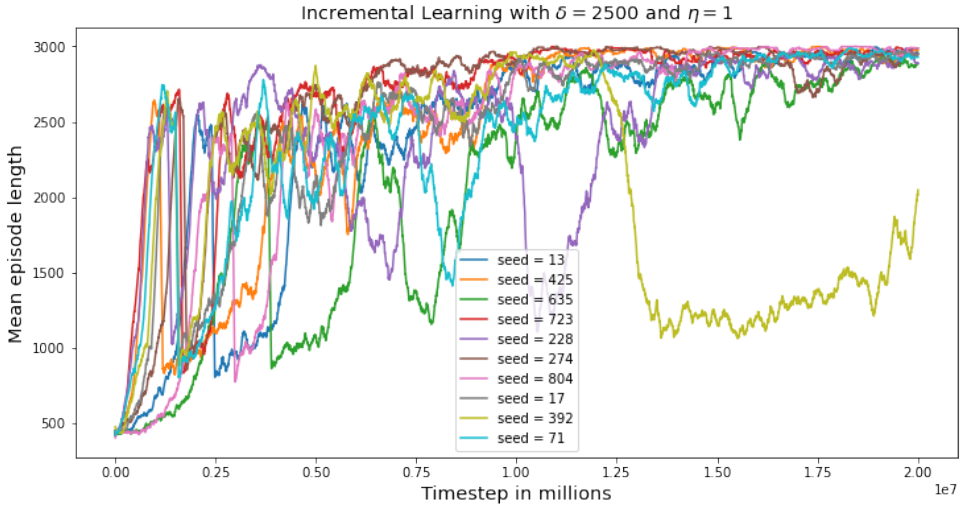


Figure B.10: Training progress of 10 different initialization with $\delta = 2500$ and $\eta = 1$ within 20 million timesteps. That have been averaged in Figure 8.14 (a)

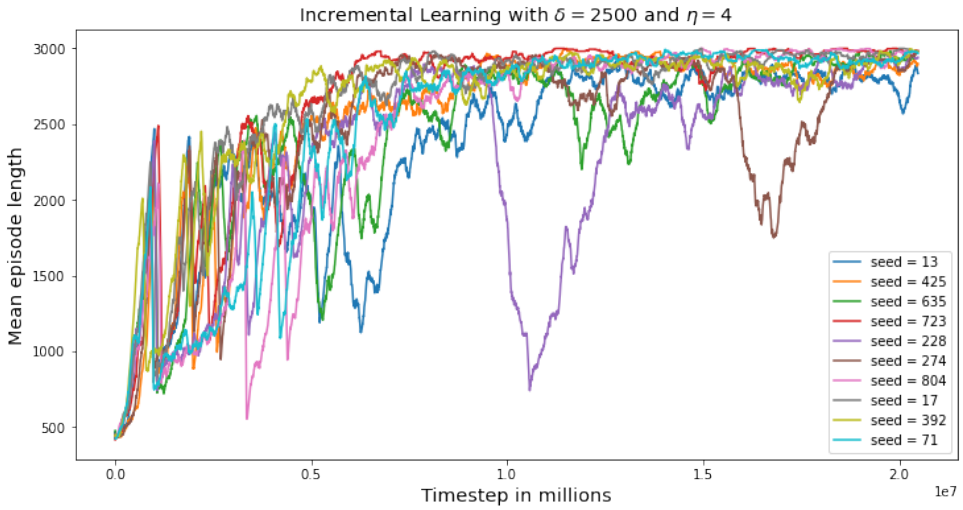


Figure B.11: Training progress of 10 different initialization with $\delta = 2500$ and $\eta = 4$ within 20 million timesteps. That have been averaged in Figure 8.14 (b)

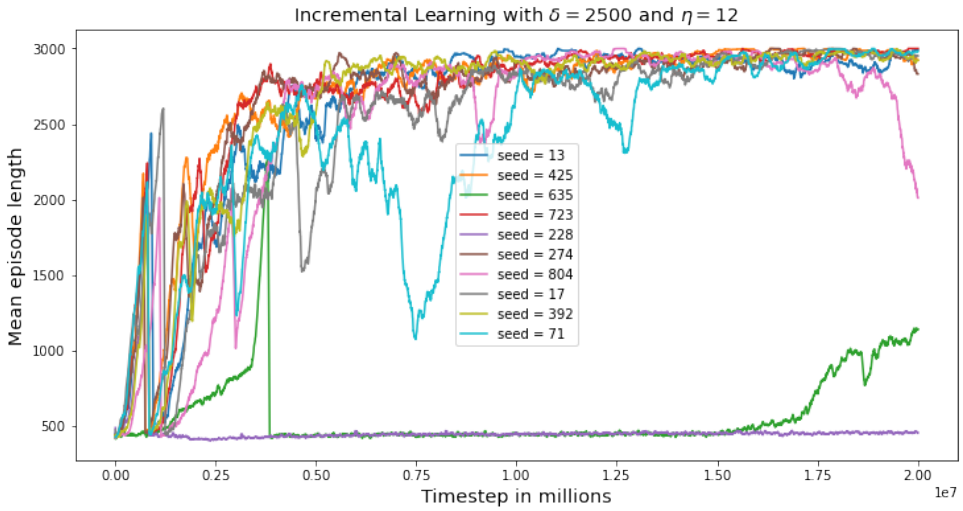


Figure B.12: Training progress of 10 different initialization with $\delta = 2500$ and $\eta = 12$ within 20 million timesteps. That have been averaged in Figure 8.14 (c)