Curriculum Learning with a progression function

Andrea Bassich

Doctor of Philosophy

University of York

Computer Science

April 2022

Abstract

Whenever we, as humans, need to learn a complex task, our learning is usually organised in a specific order: starting from simple concepts and progressing onto more complex ones as our knowledge increases. Likewise, Reinforcement Learning agents can benefit from structure and guidance in their learning. The field of research that studies how to design the agent's training effectively is called Curriculum Learning, and it aims to increase its performance and learning speed.

This thesis introduces a new paradigm for Curriculum Learning based on progression and mapping functions. While progression functions specify the complexity of the environment at any given time, mapping functions generate environments of a specific complexity. This framework does not impose any restriction on the tasks that can be included in the curriculum, and it allows to change the task the agent is training on up to each action.

The problem of creating a curriculum tailored to each agent is explored in the context of the framework. This is achieved through adaptive progression functions, which specify the complexity of the environment based on the agent's performance. Furthermore, a method to progress each dimension independently is defined, and the progression functions derived from our framework are evaluated against state-of-the-art Curriculum Learning methods.

Finally, a novel variation of the Multi-Armed Bandit problem is defined, where a target value is observed at each round, and the arm with the closest expected value to the target is chosen. Based on this framework, we define an algorithm to automate the generation of a mapping function.

The end result of this thesis is a method that is learning algorithm agnostic, is able to translate domain knowledge into an increase in performance (providing similar benefits if such domain knowledge was not available), and can create a fully automated curriculum tailored to each learning agent.

Contents

Abstract **1**

List of Figures **5**

List of Tables **9**

Acknowledgements **10**

Author's declaration **11**

1 Introduction **12**

1.1 Contributions 14

1.2 Thesis Outline 15

2 Background and Context **17**

2.1 Reinforcement Learning 17

2.1.1 Terminology and Bellman Equations 18

2.1.2 From discrete to continuous variable space 20

2.2 Transfer Learning 22

2.2.1 Multi-Task Learning 22

2.2.2 Terminology and Basic Principles 23

2.2.3 Types of transfer 24

2.2.4 Performance evaluation 25

2.3 Curriculum Learning 26

2.3.1 Curricula in Supervised Learning 27

2.3.2 Curricula in Reinforcement Learning 29

2.3.2.1 Single task curricula 30

2.3.2.2 Curricula with restrictions 31

2.3.2.3 Curricula with no restrictions 35
2.3.2.4 Curriculum Learning with a Progression Function . 38

3 Curriculum Learning Framework **40**
3.1 Curriculum 41
3.2 Framework Details 42
3.3 Progression functions 45
3.4 Mapping functions 47
3.5 Progression with parallel environments 49
3.6 Curriculum granularity 49
3.7 Special Cases 50
3.7.1 Progression in a task-based setting 50
3.7.2 Reward shaping 51
3.8 Experimental setting 52
3.8.1 Evaluation domains 53
3.8.2 Curriculum implementation details 55
3.8.3 Learning algorithm 57
3.8.4 Experimental method 57
3.9 Results 57
3.9.1 Comparison on all domains 58
3.9.2 In depth analysis 61

4 Adaptive progression **63**
4.1 Performance functions 64
4.2 Friction-based progression 65
4.3 Rolling Sphere progression 70
4.4 Progression on multiple dimensions 74
4.5 Experimental Setting 76
4.5.1 Baseline choice 76
4.5.2 Implementation details 77
4.6 Results 78
4.6.1 Friction-based progression 79
4.6.2 Effects of Multiprocessing 81
4.6.3 Robustness of adaptive progression functions 84
4.6.4 Comparison with the state of the art 89

5 Automatic mapping functions **93**
5.1 Problem Specification 94
5.2 Background 95
5.2.1 Multi-Armed Bandits 96
5.2.2 Dynamic Multi-Armed Bandits 97
5.2.3 Contextual Bandits 99
5.3 Dynamic Value Bandits 99
5.3.1 Using out of the box DMAB methods 100
5.3.2 Custom DVB algorithms 103
5.3.2.1 Dynamic Value Thompson Sampling 103
5.3.2.2 Dynamic Value BESA 103
5.3.2.3 Value Sliding Window UCB 104
5.4 DVB generated mapping functions 105
5.5 Initial Evaluation 106
5.5.1 DMAB and DVB testing environments 106
5.5.2 Data driven environment 107
5.5.3 Experimental results 109
5.5.3.1 DMAB 109
5.5.3.2 DVB 110
5.5.3.3 Adversarial DVB 110
5.5.4 Data driven environment 111
5.5.5 Initial evaluation summary 112
5.6 Fine-tuning VDTS 113
5.7 Experimental Evaluation 115

6 Conclusion and Outlooks **119**
6.1 Contributions 119
6.2 Limitations 121
6.3 Future work 122

A Additional Results **124**
A.1 Friction Based progression variants 124
A.2 Effects of Multiprocessing 125
A.3 Robustness of adaptive progression functions 126
A.4 Adversarial DVB 128

List of Figures

1.1 Chart representing the relationship between chapters. An arrow between two chapters represents a dependency of a chapter from another. 16

2.1 Agent-environment interaction. 18
2.2 On the left a representation of the source task where a policy is learnt. Transferring this policy to the task in the center would result in positive transfer, whereas transfering it to the task on the right would result in negative transfer. 24
2.3 Transfer Learning metrics visualised. The x axis represents the number of time steps the agents are trained for, whereas the y axis represents their performance. 26

3.1 Visualisation of the Point Mass Maze domain. The red area in the top right of the maze represents the goal, and the curriculum is built by changing the starting position of the agent in the maze. 42
3.2 Example of different progression functions on an environment with one dimension of complexity. The figure on the left shows the effect of changing the value of t_e in a Linear progression. The figure on the right, on the other hand shows the effect of changing the value of s in an Exponential progression. 45
3.3 The figures show how the complexity of the task the agent is training on changes over time in curricula with different granularity. The curricula are generated by the linear (left) and exponential (right) progression functions, with the "continuous" one being the standard fine-grained curriculum defined by our framework. The legends report the number source tasks in each curriculum, where a jump in the value of c_t results in a new task being added. 50

3.4 Visualisation of each one of the domains used in our testing. (From top left to bottom right: Grid World Maze, Point Mass Maze, Directional Point Maze, Ant Maze, Predator Prey and HFO). 53
3.5 Performance of the Exponential progression, the Exponential progression limited to 5 tasks and PPO on six test domains. In these plots each time step represents an action made by the agent in the environment. 58
3.6 Performance of the Exponential progression on the Point Mass Maze domain varying the granularity of the curriculum. The plot on the left reports the best performance achieved by each method, whereas the plot on the right represents the area under the performance curve for each method (normalised with respect to the best performing method plus its confidence interval). 61

4.1 Integration of adaptive progression functions within RL. After taking an action the performance is calculated if possible, and passed on to the progression function. The reward and observation are then provided to the agent. Finally, the complexity vector is passed onto the mapping function that generates the updated environment if the environment needs to be modified. 64
4.2 Plot representing the *Friction-based* progression. The change in performance of the agent is measured over an interval; this value is then used to calculate F_f, the friction force that slows down the object as the agent improves. This, in turn, results in the complexity of the environment increasing. 66
4.3 The figure on the left shows the effect of changing the magnitude of the interval, i, on a Friction-based progression, where the performance used for the progression is shown in the figure on the right. 68
4.4 Plot representing the *Rolling Sphere* progression. The change in performance of the agent is measured over one step; this value is then used as the angle in radians of the plane the sphere is rolling on. The sphere's position is then used in order to specify the complexity vector. 71

4.5 The figure on the left shows the effect of changing the simulation time, t, on a Rolling Sphere progression, where the performance used for the progression is shown in the figure on the center. The figure on the right shows the linear approximation (in red) of the physically accurate function (in blue) used to derive acceleration from a certain Δ_p (with a 2:1 ratio of the x axis w.r.t the y axis). 73
4.6 Rolling Sphere progression on multiple dimensions. A line search is performed in each dimension, as a result of which the sphere's velocity is updated. Finally the new velocity is used to determine the new position of the sphere. 75
4.7 Examples of the three different types of noise: Local (left), Global (center) and Random (right). The blue dashed line represents $\vec{c_t'} = \vec{c_t}$, i.e. no noise. 85
4.8 Performance of different progression functions, *Reverse Curriculum* and *HTS-CR* on six test domains, with the dashed line representing the final performance for *HTS-CR*. In these plots each time step represents an action made by the agent in the environment. 90

5.1 The figure on the left shows the regret of different methods on the DMAB problem. The figure in the center, on the other hand, reports the regret on the DVB environment. Finally, the figure on the right reports the regret on the adversarial DVB setting against the most effective adversary. Note that the higher the regret, the worse the performance. 109
5.2 Average error for each algorithm when the complexity is randomly sampled (left), or specified by the Rolling Sphere progression (right). Each line is the average of 200 runs smoothed with a Savitzky–Golay filter. 111
5.3 Performance of a manually defined mapping function against VDTS and ITS. 116

A.1 Performance of the three formulations of the Friction-based progression on six test domains. 124
A.2 Performance of the Friction-based progression, Rolling Sphere progression and Exponential progression on five test domains with (m.) and without (s.) multiprocessing. 125

A.3 Performance of the Friction-based progression with local (sh), global (lo) and random (rn) noise on six test domains. 126
A.4 Performance of the Rolling Sphere progression with local (sh), global (lo) and random (rn) noise on six test domains. 127

List of Tables

2.1 This table contains all the algorithms mentioned in this chapter that are aimed at creating a curriculum in a RL setting. They appear in the same order they are discussed and the type of restriction imposed on the tasks that can be part of the curriculum, alongside the method used for generation, selection and sequencing are reported. 39

4.1 Best performance on each domain. All the methods within a 95% confidence interval from the best performing method are in bold. . . 79

4.2 Best performance on each domain with and without multiprocessing, divided by method. If the algorithm with and without multiprocessing perform within a 95% confidence interval, both performances are in bold, otherwise the highest of the two is in bold. 82

4.3 Average execution time on each domain with and without multiprocessing, divided by method. 84

4.4 Best performance on each domain. All the variants of each algorithm within a 95% confidence interval from the method without noise are in bold. 89

4.5 Best performance on each domain. All the methods within a 95% confidence interval from the best performing method are in bold. . . 92

A.1 Results for adversarial DVB with $\sigma^2 = 0$ 128

A.2 Results for adversarial DVB with $\sigma^2 = 0.001$ 128

A.3 Results for adversarial DVB with $\sigma^2 = 0.005$ 128

A.4 Results for adversarial DVB with $\sigma^2 = 0.008$ 128

A.5 Results for adversarial DVB with $\sigma^2 = 0.01$ 129

A.6 Results for adversarial DVB with $\sigma^2 = 0.02$ 129

Acknowledgements

I would like to start this thesis by thanking all the people that helped and supported me throughout my PhD. This work would not have been possible if not for the guidance of Daniel Kudenko, who introduced me to the area of Curriculum Learning and has given me the opportunity and motivation to embark on this journey. I deeply appreciate the support and encouragement Daniel has given me throughout the years.

I am grateful for James Cussens and Rob Alexander, for their guidance and taking on my supervision, alongside a big thank you to Simos Gerasimou for his supervision in the later stages of my PhD and the helpful feedback on this thesis.

I would also like to thank Matteo Leonetti and Francesco Foglino for their help in shaping this work into what it is today, and for the many insightful discussions.

Finally, I would like to thank my family and friends for the support and encouragement throughout this journey.

Author's declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Some of the ideas behind Chapter 3 and Chapter 4 appeared in:

- Bassich, A. and Kudenko, D., 2019. Continuous Curriculum Learning for Reinforcement Learning. In Proceedings of the 2nd Scaling-Up Reinforcement Learning (SURL) Workshop. IJCAI.

Moreover parts of the work in both Chapter 3 and Chapter 4 appear in:

- Bassich, A., Foglino, F., Leonetti, M. and Kudenko, D., 2020. Curriculum Learning with a Progression Function. arXiv preprint arXiv:2008.00511.

Chapter 1

Introduction

Whenever we, as humans, need to learn a complex task, our learning is usually organised in a specific order: starting from simple concepts and progressing onto more complex ones as our knowledge increases. The order in which concepts are learnt is often referred to as curriculum, and it is not a concept limited to human learning. Curricula are also used to teach animals very complex tasks, like in Skinner (1951), where a pigeon was trained to recognise the suits in a pack of cards. Likewise, machines can benefit from structure and guidance in their training. This guidance can take various forms, one of which is to organise the learning process in a similar way to the human educational system: progressively introducing more concepts and more complex examples. This is the principle behind Curriculum Learning (CL), the field of research that studies how to maximise the efficiency of the training of Machine Learning models. While Curriculum Learning can be used with different types of learners, this thesis will focus on its application to Reinforcement Learning.

Reinforcement Learning is the area of Machine Learning that deals with the problem of teaching agents to make decisions in order to maximise the expected reward. While Reinforcement Learning's potential has been shown even before the turn of the millenium, for instance when Tesauro et al. (1995) (TD-Gammon) achieved super-human play in the game of Backgammon, it is not until more recently that the field has become increasingly popular. This rise in popularity can be attributed to Deep Reinforcement Learning, a field that was introduced by Mnih et al. (2013), and that has resulted in Reinforcement Learning methods beating the world champion in the game of Go (Silver et al., 2016), creating a revolution in the world of chess engines (Silver et al., 2017), and achieving Grandmaster level in the game of StarCraft (Vinyals et al., 2019).

Despite these impressive feats, Reinforcement Learning agents require a significant amount of time and experience for their training: TD-Gammon played 1.5 million games of Backgammon, whereas Alpha Zero played 19.6 million games of Chess. In order to obtain the same amount of experience, a human player would need to play about 11 consecutive years of Backgammon and 74 consecutive years of Chess respectively (with the conservative estimate of two minute per Chess game and four minutes per Backgammon game). Furthermore, the more complex and high dimensional the problem is, the more samples are needed to learn, which is one of the main reasons RL's poor sample efficiency is its limiting factor to being scaled to more complex applications (Narvekar et al., 2020).

One way to increase the sample efficiency of Reinforcement Learning is the previously introduced Curriculum Learning, a technique that revolves around training an agent on a sequence (or a DAG) of intermediate tasks, intending to increase the performance and learning speed on a target task. While Curriculum Learning for Reinforcement Learning is a relatively young field of research, three distinct classes of methods have emerged: single task (Section 2.3.2.1), with restrictions (Section 2.3.2.2) and no restrictions (Section 2.3.2.3). The former generates a curriculum by only training on the target task and organising the agent's experience to maximise sample efficiency. On the other hand, Curricula with restrictions impose a limitation, often deriving from the way the curriculum is generated, on the type of intermediate task that can be part of the curriculum. Finally, as the name suggests, methods with no restrictions do not impose any limitations on the nature of the tasks.

The motivation behind this thesis exploring Curriculum Learning is both Reinforcement Learning's potential and Curriculum Learning being aimed at mitigating its weaknesses. Furthermore, this thesis can be placed in the area of no restrictions Curriculum Learning, as one of our aims is to allow any MDP[1] to be part of the curriculum. The research hypothesis that governs this thesis is the following:

High-level domain knowledge can be used to support the generation of an informed curriculum that can train an RL agent more efficiently than other state-of-the art Curriculum Learning methods.

In an effort to either prove or disprove this hypothesis, this thesis initially sought to address the following research question:

[1]Markov Decision Process, defined in Section 2.1.1

1. How can high-level domain knowledge be leveraged in the creation of a curriculum?

An example of such domain knowledge would be: increasing the reaction time of an agent learning to drive a car will result in the problem being harder to solve. The answer to the former question resulted in the following questions presenting themselves:

2. How can the complexity of the environment be selected for a given learning agent?

3. How can the complexity of an environment be assessed for a given learning agent?

1.1 Contributions

In an effort to answer the three research questions mentioned above, this thesis makes the following contributions to the field of Curriculum Learning in Reinforcement Learning:

Defines a novel Curriculum Learning framework In order to address the questions posed above, we introduce a novel framework for curriculum generation based on two components: *progression* functions that specify the environment's complexity, and *mapping* functions that generate environments of a specific complexity. (Addressed in Chapter 3, relevant to research questions *1* and *2*)

Defines a way to exploit high-level domain knowledge This framework allows for simple yet significant knowledge about the environment to be used in the generation of a curriculum tailored for each agent. This is a problem that no other Curriculum Learning algorithm from the literature addresses to the same extent. (Addressed in Chapter 3, relevant to research question *1*)

Determines how long should be spent on each task This thesis also implicitly answers an open question in the Curriculum Learning literature: how long should be spent on each task. Progression functions specify this within the framework defined in this thesis.
(Addressed in Chapter 3, relevant to research question *2*)

Builds an online curriculum tailored to the agent This thesis defines several methods to automatically build a curriculum tailored to the agent's ability. These methods are all online and result in the possibility to change the target task up to every time step.
(Addressed in Chapter 4, relevant to research question *2*)

Automates the generation of a mapping function This thesis automates the generation of a mapping function, therefore answering the question of how complex different environments are for each agent. Answering this question allows us to select tasks of the appropriate complexity based on the agent's specific ability.
(Addressed in Chapter 5, relevant to research question *3*)

Defines a new variation of the Multi-Armed Bandit problem In order to automate the generation of a mapping function, this thesis also defines a new variation of the Multi-Armed Bandit[2] (MAB) problem. This variation, named Dynamic Value Bandits, requires the agent to pull the arm whose expected payoff is closest to a target value that changes each round. In this thesis, three MAB algorithms were also adapted to solve this new problem.
(Addressed in Chapter 5, relevant to research question *3*)

1.2 Thesis Outline

The structure of this thesis, as well as the dependency between chapters, can be seen in Figure 1.1. This chapter introduced the questions this thesis seeks to answer and outlined the contributions made to the field of Curriculum Learning while trying to answer these questions.

Chapter 2 covers the necessary background information concerning Reinforcement Learning, Curriculum Learning as well as some related paradigms. In this chapter, Curriculum Learning methods are also separated into three categories based on the type of restriction they impose on the tasks in the curriculum.

Chapter 3 defines our Curriculum Learning framework, as well as two separate progression functions that result in a pre-determined curriculum. This chapter also addresses the problem of adapting our framework to parallel workers, and how often the complexity of the environment should be changed during the curriculum.

[2]This problem is explained in detail in Section 5.2.1

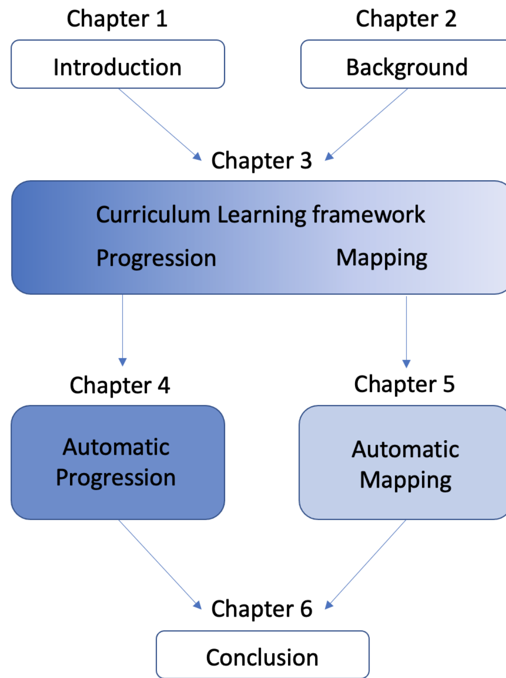

Figure 1.1: Chart representing the relationship between chapters. An arrow between two chapters represents a dependency of a chapter from another.

Chapter 4 introduces adaptive progression functions that change the complexity of the environment based on the performance of the agent. This new class of functions allows us to generate an online curriculum tailored to the agent's ability. This section also discusses how to progress each dimension of complexity independently. Finally, this section performs several ablation studies on our methods and compares them against state-of-the-art Curriculum Learning algorithms.

Chapter 5 addresses the automatic generation of mapping functions by first accurately specifying the problem and then formalising a new Multi-Armed Bandit problem. After specifying a principled way of adapting existing methods to solve this problem, three Multi-Armed Bandits algorithms are adapted to this new variation. Their performance is evaluated on a range of domains, and the best performing algorithm is adapted to the problem of generating a mapping function. The newly generated algorithm is finally evaluated against a manually defined mapping function.

Finally, Chapter 6 gives a summary of the work presented in this thesis and addresses the contributions and limitations of the work. The chapter will end with ideas on how to address these limitations in future work.

Chapter 2

Background and Context

This chapter presents the background information that is relevant to the research undertaken in this thesis. It will also put my work in the broader context of Curriculum Learning by providing an overview of the existing literature in the area.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a popular field in Machine Learning used to solve tasks requiring an agent to take actions in an environment. Unlike Supervised Learning, where the goal is to minimise a loss function relative to a pre-labelled training set, RL aims to maximise the reward gathered by an agent in an environment. The environment is modelled as a Markov Decision Process[1] (MDP), that is defined by a state space, an action space, a transition function and a reward function. The agent-environment interaction takes place over discrete time steps $t \in \mathbb{N}$, and it starts with the agent receiving an observation of the environment's state s_t. Based on this observation and its internal parameters, the agent chooses an action a_t and executes it in the environment. This results in a change in the environment's state according to its transition function P, that specifies the probability of state s_{t+1} occurring when taking action a_t in state s_t. After acting in the environment, the agent receives a reward r_{t+1} and observes the state s_{t+1}, a cycle that will repeat throughout the agent's training.

In this setting, the agent's goal is to perform actions that maximise the reward received, resulting in RL algorithms trying to solve a challenging optimisation problem where the reward is often delayed. An environment where this property is

[1]Defined more formally in Section 2.1.1

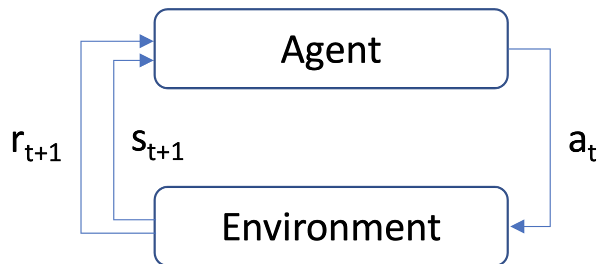

Figure 2.1: Agent-environment interaction.

evident is the NES game "Super Mario Bros": if the agent jumps and falls into the lava, it receives a negative reward upon dying, not jumping; it is up to the agent to learn the correlation between the two. However, this example also highlights another challenge posed by RL: the agent's actions directly affect the data it receives. Furthermore, this environment poses an additional hurdle to a learning agent by naturally having a "sparse" reward function, a quality shared by many complex environments and one of the sources of the scalability issues behind standard RL methods. Sparsity in the reward function implies that in a given environment, only a small subset of states provide the agent with a reward signal; for Super Mario Bros, the agent would receive a positive reward for completing a level and a negative reward for dying, with all other states providing a constant reward (usually 0 or -1). To learn in such an environment, the agent needs to move towards the goal over obstacles before experiencing a positive reward signal. The risk is for the agent to get stuck in a dangerous local optimum where it stays stationary to avoid the negative reward caused by the obstacles on the way to the goal. This simple example motivates the need for additional techniques such as Curriculum Learning to supplement RL algorithms in learning more complex environments.

2.1.1 Terminology and Bellman Equations

In this thesis, tasks are modelled as Markov Decision Processes (MDPs). An MDP is formally defined as a tuple $\langle S, A, P, R, \gamma \rangle$, where S is the set of states, A is the set of actions, $P : S \times A \times S \to [0, 1]$ is the transition function, $R : S \times A \to \mathbb{R}$ is the reward function and $\gamma \in [0, 1]$ is the discount factor. Episodic tasks have *absorbing* states, which cannot be left and from which the agent only receives a reward of 0. An example of an absorbing state would be the state in which Mario completes the level in the previously mentioned Super Mario Bros environment. By definition,

MDPs also need to satisfy the Markov property, whose principle is that "the future is independent of the past given the present" (Silver, 2015). The following equation defines this property:

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_{[0,t]}, a_{[0,t]}) \tag{2.1}$$

This implies that any state contains all the relevant information for a given MDP, rendering the history of the task superfluous. Some domains can be modelled using MDPs, but do not allow the agent to have direct access to the state; an example of this is poker, where the agent is not allowed to know the opponent's hand. In order to model such problems, Partially Observable MDPs (POMDPs) were introduced by Jaakkola et al. (1994). The principle behind POMDPs is that the agent only has access to an observation of the state, which is provided by the environment.

As mentioned above, the agent-environment interaction requires the agent to select an action after observing the environment's state; given a state, an action is selected according to the agent's policy. A policy $\pi : S \times \mathbb{R} \to A$ is a function that maps a state s and a time step t to an action a. A policy π is regarded stationary if $\pi(s, t)$ is independent of t (Papadimitriou and Tsitsiklis, 1987). In order to evaluate and improve the agent's policy, two distinct value functions are used in Reinforcement Learning: the state-value function and the action-value function. Given a certain policy π and state s, the state-value function, denoted $V_\pi(s)$, is the expected return when starting in s and following policy π thereafter. The action-value function $Q_\pi(s, a)$, on the other hand, is the expected return starting from s, taking action a, and thereafter following policy π (Sutton and Barto, 1998). The equations that define $V_\pi(s)$ and $Q_\pi(s, a)$ are the foundations of RL and are called the Bellman equations:

$$V_\pi(s) = E_\pi[r_{t+1} + \gamma V_\pi(s_{t+1})|s_t = s] \tag{2.2}$$

$$Q_\pi(s, a) = E_\pi[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a] \tag{2.3}$$

The relevance of these equations lies in the fact that they can be optimised to find the optimal policy π^*. The optimisation can either happen on V_π or Q_π resulting in the value iteration method, or directly on the policy resulting in the policy iteration method. The value iteration algorithm consists of iteratively refining the estimate of Q or V in order to derive either one of the below:

$$V^*(s) = max_{\pi} V_{\pi}(s) \quad \forall\ s \in S \tag{2.4}$$

$$Q^*(s,a) = max_{\pi} Q_{\pi}(s,a) \quad \forall\ s \in S, a \in A \tag{2.5}$$

V^* and Q^* are described as the optimal value functions, and the value iteration method is guaranteed to converge to the optimal value function in a discrete environment (Sutton, 1988). On the other hand, the policy iteration method starts with a randomly initialised policy, uses one of the value functions to evaluate the policy, and improves the policy by using the information gathered during the evaluation phase. This improvement is achieved by modifying the policy to maximise the value function for each state (or state-action pair). Given a finite MDP, policy iteration is also guaranteed to converge to the optimal policy in a finite number of steps (Santos and Rust, 2004).

2.1.2 From discrete to continuous variable space

The methods outlined above assume that the MDP they are applied on is discrete, both with respect to the state space and the action space, which allows for a lookup-table to store the policy or value function (depending on the method). An example of an environment satisfying these assumptions would be a Grid World maze with a limited set of squares. Nevertheless, numerous domains in RL have continuous state and action spaces, which would require an infinite lookup table, as one entry is reserved for each state. Domains with these properties include ATARI games, robotics simulations, financial portfolio management etc. As an infinite lookup table cannot exist, continuous policies and value functions are represented using function approximators, with the drawback of losing the guarantees of optimality mentioned previously.

A possible solution to dealing with a continuous variable space is to turn it into a discrete one using Tile Coding (Sutton and Barto, 1998). A tiling involves dividing the space in n-dimensional tiles, making it possible to define a point in space by the tile it is contained in. Tile coding approaches often use multiple tilings offset from one another, turning a continuous space into a binary vector. However, one drawback of this technique is that it is sensitive to parameter selection, like the number of tilings to be used, the size of the tiles and offset between different tilings. For this reason, algorithms to automate the parameter choice of tile coding were designed (Sherstov and Stone, 2005), although the reliance on lookup tables results

in tile coding not being practical for very complex high dimensional environments, such as when learning from images.

An alternative to tile coding is to use another function approximator, the most widely used being neural networks. The first attempts at applying neural networks to Reinforcement Learning aimed to approximate the value function and use value iteration as a learning algorithm. However, Boyan and Moore (1995) concluded that function approximators, neural networks included, were too unreliable for the task and focused on trying to increase their reliability. Tsitsiklis and Van Roy (1997) on the other hand, found that when using a non-linear function approximator, Temporal Difference learning tends to have diverging behaviour. Since then, the computational power available has vastly increased, and with it, the size of the neural networks used, mainly due to the amount of time and computational power required to train them. In more recent years, this has caused a renewed interest in the application of deep neural networks to RL, with algorithms like Deep Q-Networks (DQN) (Mnih et al., 2015). This algorithm uses a deep neural network to model the Q function and addresses the instability issues reported in the past by utilising experience replay. This technique, first introduced by Lin (1992), involves storing the agent's experience and letting the network learn it randomly to reduce the correlation between samples. DQN's update rule used makes the target values that the Q function is adjusted towards change periodically instead of changing at every time step, resulting in a further increase in the stability of the method.

A popular class of learning algorithms used in conjunction with neural networks are Actor-Critic methods. They incorporate principles from both value and policy iteration methods and are composed of an actor (a policy) and a critic (a value function). The actor's task is to act in the environment, whereas the critic evaluates the policy and is used to derive the gradient update for the policy. An example of an algorithm in this class of methods is A3C (Mnih et al., 2016), which uses the critic in order to estimate the advantage function $A(s, a) = Q(s, a) - V(s)$ which is in turn utilised in the policy gradient update. The peculiarity of this algorithm is that it allows for parallel learners to perform gradient updates independently. A synchronous version of this algorithm, named A2C, was also developed in order to avoid the possibility of different threads overwriting updates. While these algorithms perform linear gradient-descent, TRPO (Schulman et al., 2015) uses Trust Region optimisation in order to update the policy. This method involves defining a region constrained by KL divergence, in which a local search is performed for an improved policy. Wu et al. (2017) (ACKTR) improves the scalability of this method by using

Kronecker-factored approximation, which has the effect of reducing the computational complexity of TRPO. The downside of trust policy methods is that they are hard to implement, and their sample efficiency can be further improved upon. These limitations are addressed by PPO (Schulman et al., 2017), which defines a clipped objective function that penalises large policy updates. This function is optimisable with gradient descent (first-order optimisation) algorithms while still retaining some of the benefits of TRPO and ACKTR, resulting in an easily implementable state-of-the-art RL algorithm which will be used extensively in the experimental evaluation in this thesis.

2.2 Transfer Learning

Before introducing Curriculum Learning, it is necessary to consider the closely related area of Transfer Learning. During a curriculum, the agent is trained on a sequence of tasks, and Transfer Learning defines how various types of information regarding one task can be transferred to another. However, before talking about Transfer Learning, another closely related method worth mentioning to introduce this area is Multi-Task Learning.

2.2.1 Multi-Task Learning

Multi-task Learning is a Machine Learning paradigm that enables the learning of multiple tasks simultaneously. This method requires using the same (or very similar) model to solve multiple tasks, achieved through parameter sharing. There are two types of parameter sharing: hard and soft. Hard parameter sharing is the most commonly used form of Multi-task Learning (Ruder, 2017), and it was introduced by Caruana (1993). It involves using a model with some task-specific layers and other layers which are in common between tasks. Hard parameter sharing can be beneficial, as it reduces the chance of over-fitting the samples (Baxter, 1997), encouraging the hidden layers in common to learn high-level level features rather than to solve a task. On the other hand, soft parameter sharing does not have any shared layers, but it requires a separate neural network for each task. Each network shares the same structure, and their parameters are encouraged to be similar throughout the training (Ruder, 2017).

The concept of hard parameter sharing can also be found in modern Reinforcement Learning, where it is often the case that in actor-critic algorithms, the networks

used for the policy and value function have some layers in common, with the aim of learning high-level features, as mentioned above. However, in Curriculum Learning, the tasks are not learnt simultaneously but rather in a sequence, which can be achieved using Transfer Learning.

2.2.2 Terminology and Basic Principles

Transfer Learning is the field that studies how information that was learnt in a task can be transferred to another task. This principle can also be found in human learning, where in order to learn complex tasks, like riding a motorbike, we often rely on previous knowledge on related activities, like riding a bicycle. In Transfer Learning, there are two types of tasks: source and target. The agent learns source tasks to acquire some form of domain knowledge, which is then transferred to the target task. In the example mentioned above, the source task would be riding a bicycle, whereas the target task would be riding a motorbike. Unlike Multi-task Learning, Transfer Learning is solely focused on improving the performance on the target task, regardless of the final performance on the source task. Relating this to the example above, it is immaterial if we cannot ride a bicycle after learning how to ride a motorbike.

In Transfer Learning, there are three main problems to solve: what to transfer, how to transfer, and when to transfer. It is the way that one goes about solving them that will then define the method used. What to transfer determines what knowledge should be learnt in the source task to transfer to the target task. This "knowledge" could be as simple as a policy, action value, or state value function or more high-level information such as the general structure of an MDP. Once the information to transfer has been identified and learnt, the next step is to define a method to transfer it from one task to another. The transfer method is a core part of the algorithm and depends entirely on what is transferred. One point that is often overlooked is: when to transfer (Pan and Yang, 2010), which poses the question of when it is appropriate to transfer between tasks. It could be when an arbitrary criterion is met, like a certain performance threshold was reached or after a certain number of time steps. Note that in some instances, the answer could be that the transfer should not happen at all. This question is also related to Curriculum Learning, as a critical issue in the field is how long to spend on each source task, as will be seen later.

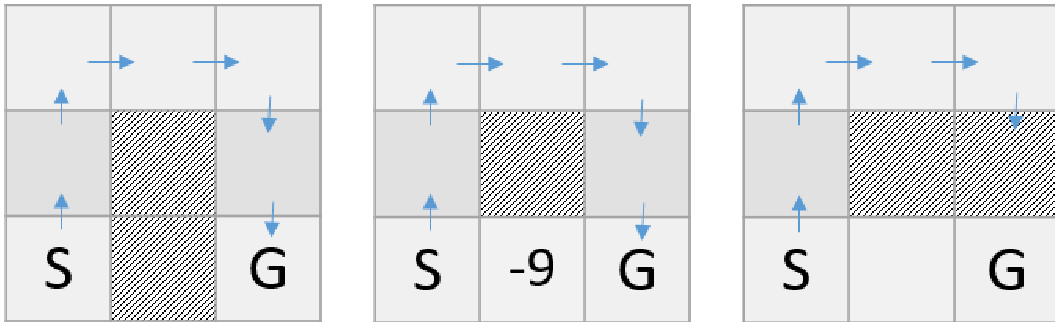

Figure 2.2: On the left a representation of the source task where a policy is learnt. Transferring this policy to the task in the center would result in positive transfer, whereas transfering it to the task on the right would result in negative transfer.

2.2.3 Types of transfer

As previously mentioned, Transfer Learning is a valuable approach as it allows knowledge acquired from solving source tasks to be leveraged in order to better solve a related task. Transfer Learning is particularly relevant when a solution to the source tasks is already known or when the target task is too complex to be solved independently. On the other hand, Transfer Learning can also hinder the agent's training on the target tasks depending on the choice of source tasks. There are, in fact, two types of transfer that can occur: positive and negative. Positive transfer is the desired effect of Transfer Learning, and as suggested by the name, it results in the knowledge transferred positively impacting the learning on the target task. On the other hand, a negative transfer can be experienced when the knowledge is transferred from "highly irrelevant sources" (Ge et al., 2014) and is detrimental to the agent's learning. An example is presented below to better highlight positive transfer and negative transfer and how each might arise.

For this example, the Transfer Learning algorithm used is the one defined by Taylor et al. (2007) which includes four steps. Firstly, the learner is trained on the source task using policy iteration methods. A partial mapping is then performed between "very similar" action/state variables, leaving novel ones unmapped. Next, the policy obtained on the source task is transformed based on this mapping. Finally, the resulting policy is used as a starting policy to learn the target task.

Assume that the task involves solving one of the mazes in Figure 2.2. The possible actions are to move North, South, East or West, and the aim is to get from the initial state S to the goal state G. The squares filled with a pattern are blocked, and if the agent performs an illegal move (towards a blocked square or out

of bounds), its position will be unchanged. Finally, the agent experiences a reward of -1 per action plus any reward indicated in the square and a reward of 10 for reaching the goal.

As mentioned earlier, the chosen transfer method will be to learn a policy for the source task and apply it to the target tasks as a starting point for the training. The optimal policy for the source task is shown in each figure as a series of blue arrows. If this policy is transferred to the target task in the centre of Figure 2.2, it will have a positive effect on the training (positive transfer), as the agent will start with an already optimal policy. On the other hand, if the optimal policy for the source task is used as a starting point for the target task on the right in Figure 2.2, it will result in the agent getting stuck on the top right state. As there is still a degree of exploration involved in the training, the agent will eventually find its way to the goal; however, this will happen at a much slower rate than when not using Transfer Learning.

As Curriculum Learning uses Transfer Learning whenever transferring information between tasks, the concept of positive and negative transfer is relevant when choosing which tasks to include in a given curriculum and their order. This is especially relevant because negative transfer could result in a lower performance than plain RL, as shown in the example above. Furthermore, as a curriculum transfers information from one task to the next until the target task is reached, an instance of negative transfer in the middle of a curriculum could result in the benefits of applying a curriculum being lost altogether. For this reason, Svetlik et al. (2017) tries to estimate "transfer potential", a measure of how beneficial it would be to transfer information between two tasks.

2.2.4 Performance evaluation

As Transfer Learning tries to solve a problem which is different to that of standard RL, it also needs some specific metrics to evaluate the performance of a method. Performing transfer between two tasks can, in fact, have a lot of different effects on the behaviour of the learner on the target task. Some of the common metrics used in Transfer Learning can be seen in Figure 2.3, and include: jump-start, asymptotic performance, total reward and time to threshold (Taylor and Stone, 2009). A jump-start occurs whenever there is an increase in the performance of the agent at the start of the training as a result of transfer. On the other hand, the asymptotic performance achieved by a method is defined as the final performance achieved by

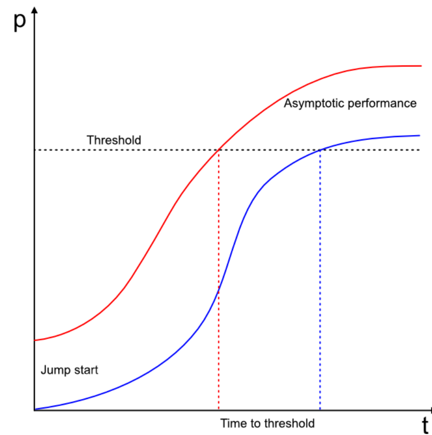

Figure 2.3: Transfer Learning metrics visualised. The x axis represents the number of time steps the agents are trained for, whereas the y axis represents their performance.

the agent, where the asymptote represents the performance which the agent is bound by. The total reward metric measures the total reward obtained by the agent during the training. Finally, time to threshold measures the number of time steps required for an agent to reach a set performance threshold.

While these metrics are helpful to highlight the effects of different Transfer Learning algorithms, they are also applicable to Curriculum Learning. When applying these metrics to Curriculum Learning, each metric is calculated on the target task and compared between the methods to evaluate.

2.3 Curriculum Learning

Whenever we, as humans, need to learn a complex task, we usually organise our learning in a specific order: starting from simple concepts and progressing onto more complex ones as our knowledge increases. This method of learning in humans was explicitly evaluated by McCandliss et al. (2002), where the aim was to teach Japanese adults to distinguish between the **r** and *l* consonants in spoken English (e.g. **r**ock *l*ock). The study found that the most successful way to teach the participants was to start with examples where the difference between the two consonants was exaggerated and later learn more complex samples. The order in which concepts are learnt is often referred to as curriculum, which is not limited to human learning. Curricula are, in fact, also necessary to teach animals complex tasks, like in Skinner (1951) where a pigeon was trained to recognise the suits in a pack of cards. Besides

being an efficient way to learn, curricula of progressively increasing complexity also seem to be the natural way humans teach a task. Khan et al. (2011) explored this by observing the method used by human participants to teach the concept of graspability to a robot. In this experiment, participants were presented with objects ranging from a toothbrush (easily graspable) to a building (impossible to hold), with some objects being at a boundary between graspable and non-graspable. The experiment aimed to observe how participants would choose a sequence of training examples to maximise the learning of the robot. All of the contestants started by showing examples of either clearly graspable objects or clearly non-graspable objects to avoid ambiguity in the initial phases of training. As the training progressed, the examples became less evidently graspable or non-graspable, and as a result, the complexity increased.

While Curriculum Learning is a method that was used before its formalisation (Elman, 1993; Tesauro et al., 1995), it was officially introduced by Bengio et al. (2009) in the context of Supervised Learning. It was then formalised in the context of Reinforcement Learning by Narvekar et al. (2016) and has evolved over time with the goal to automate the generation of curricula as much as possible. A more detailed history and general review of the Curriculum Learning literature can be found in survey papers such as Narvekar et al. (2020), Wang et al. (2021) and Soviany et al. (2021).

This section will first discuss how the concept of a curriculum evolved in Supervised Learning and then discuss the relevant literature in Reinforcement Learning. In particular, three classes of Curriculum Learning algorithms for Reinforcement Learning are identified based on the type of restrictions they impose on the tasks that can be added to the curriculum.

2.3.1 Curricula in Supervised Learning

The idea of applying a curriculum to Supervised learning dates back to Elman (1993) where it is suggested that whenever training a neural network, it might be advantageous to expose it to an increasingly complex subset of the training data. This idea was then formalised by Bengio et al. (2009) and named Curriculum Learning. In particular, they framed curriculum generation as a continuation method, where a smoothed objective function is optimised first, and the level of smoothing decreases over time (Allgower and Georg, 1980). For Curriculum Learning, the aim is to minimise a cost function C_λ, where C_1 is the target cost function, and C_0 is

the most easily optimisable (or smoothed) cost function. The approach described in the paper requires a human expert to provide a set of probability distributions Q_λ, which define the probability of a sample being chosen. Given a sequence of λ values, a curriculum is defined as a sequence of probability distributions $(Q_{\lambda 0}, ..., Q_{\lambda n})$ of increasing entropy (resulting in an increased complexity).

Khan et al. (2011) aimed to observe the way humans are naturally inclined to teach a task by performing the experiment outlined in the introduction to this chapter (teaching a robot graspability), to then use this as the basis for a novel Curriculum Learning framework aimed at minimising the learner's error at each iteration.

Kumar et al. (2010) defines an alternative paradigm, Self-paced Learning, aimed at automatically generating a curriculum, where the curriculum is learnt alongside the weights for the network. As the generation of the curriculum is added to the objective function as a regularisation term, any loss function can be used. The curriculum generation is parametrised by K, which determines the number of samples to be considered. K is decreased over time, resulting in a smaller set of "easy" samples being selected at the start of the training (large value of K), with the whole set being selected as $K = 0$.

Jiang et al. (2015) expands on this concept by observing the fact that Self-paced Learning is prone to overfitting and is not able to account for any prior knowledge. On the other hand, Curriculum Learning is compared to instructor-driven human learning, where the curriculum is pre-defined and does not consider each learner's ability. Therefore, they define a new framework that encompasses both paradigms and aims to combine the benefits of each approach. They assume that a "total order curriculum" is provided by an oracle, specifying the order in which samples should be learnt. This ordering is then used to define a curriculum region that bounds the space of tasks included in the creation of a curriculum, which is generated using a self-paced function that selects the learning scheme within the bounds imposed by the region.

More recent work in the field of CL for Supervised Learning has focused on learning a curriculum rather than generating one manually. In particular, Graves et al. (2017) aims to minimise the loss function as quickly as possible by formulating the problem of curriculum creation using the dynamic Multi-Armed Bandits framework. While measuring the effect of pulling an arm (selecting a state) on the target objective is often unfeasible, to provide a reward for selecting a state, two factors are considered: learning progress and complexity of the neural network. The Exp3.S

algorithm (Auer et al., 2002b) is used in this setting in order to select training samples that maximise the reward. This method results in an automatic curriculum generation without any prior knowledge.

As mentioned in the introduction to this thesis, Chapter 5 will apply the Multi-armed bandit framework to curriculum generation in Reinforcement Learning. While Graves et al. (2017) formulates the problem of selecting a task as maximising a heuristic meant to estimate the utility of adding it to a curriculum, our approach consists of selecting the task whose complexity is closest to a given value. Using the framework for curriculum generation that will be defined in the next chapter allows us to separate selecting the appropriate complexity (progression) from generating the task of the closest complexity to the desired one (mapping). This implies that no heuristic estimating the benefit of adding a task to the curriculum is needed, which could be hard to define for Reinforcement Learning.

2.3.2 Curricula in Reinforcement Learning

Curriculum Learning applied to Reinforcement Learning is the principal area of focus of this thesis, and it builds on the same principles of its Supervised Learning counterpart. Whilst the principles are similar, generating a curriculum for an RL agent introduces its specific issues. In particular, while in Supervised Learning the aim is to minimise a loss function over a set of samples, in RL, the curriculum is aimed at maximising various metrics on a single MDP: the *target task*. Moreover, in Supervised learning, a curriculum usually has the effect of modifying the probability of the learner being exposed to each sample in a well-defined training set. In RL, on the other hand, defining the set of *source tasks* that can be included in the curriculum is a problem that each algorithm needs to tackle. Generating a curriculum for an RL agent can be divided into three distinct sub-problems: task generation, task sequencing, and transfer method used (Narvekar et al., 2020). The former addresses how the tasks included in a curriculum are generated; task sequencing refers to the method used to determine in which order tasks should be learnt; finally, different algorithms will transfer different knowledge from task to task during the curriculum, such as a policy or a reward function. This section describes how algorithms in the Curriculum Learning literature solve one or more of the problems outlined above. It then concludes by describing the novelty of the proposed framework with respect to the existing literature.

Curriculum Learning methods can be classified based on the restrictions imposed

on the MDPs included in the curriculum. In particular, we divide algorithms into three distinct classes: single task, with restrictions and with no restrictions. Single task methods only train on the target task and focus on organising the agent's experience as efficiently as possible. On the other hand, curricula with restrictions impose a constraint on the MDPs that are part of the curriculum, such as that only the starting state distribution or the reward function can be changed. Finally, as the name suggests, curricula with no restrictions have complete control over the MDPs in the curriculum, allowing any MDP to be a source task.

2.3.2.1 Single task curricula

Single task Curricula, as previously mentioned, include all methods that generate a curriculum on the target task alone. As these methods only act on one task, Transfer Learning techniques are not needed. The principle behind single task Curricula is to organise the agent's experience in the target task to maximise the sample efficiency of RL algorithms.

The first method to generate a Single task curriculum was Prioritised Experience Replay (Schaul et al., 2016), which aims to increase the frequency at which critical transitions are presented to the agent by the replay buffer. The metric chosen for prioritising samples in the buffer is the TD error[2], which prioritises unexpected transitions. If the prioritisation were to be implemented greedily, the algorithm would be prone to overfitting, and tasks with stochastic rewards would pose a significant challenge. For these reasons, the method used to select samples to be replayed is stochastic, interpolating between greedy selection and uniform sampling while maintaining monotonicity with respect to the priority of a sample and the probability of it being selected. Subsequent work defined more sophisticated metrics of prioritising samples, such as Ren et al. (2018), where a combination of a self-paced function and a coverage penalty is used to define the priority of each sample. This work addresses the issues of using TD error to determine priority, as a noisy reward function, bootstrapping, and replaying unnecessary transitions can negatively affect the agent's training.

Other approaches, such as Hindsight Experience Replay (HER) (Andrychowicz et al., 2017), address the problem of over-engineered reward functions being needed when learning in complicated domains. Creating such a reward function often needs both Reinforcement Learning and domain knowledge, whereas using a sparse and

[2]The Temporal Difference (TD) error is the difference between the estimated and the discounted value of a transition.

binary reward function allows for learning in environments where this knowledge might not be available. Thus, the HER algorithm was developed in order to learn from such sparse rewards, building on the idea of Universal Value Function Approximators (UVFA) (Schaul et al., 2015). UVFAs differ from standard Value Function Approximators by needing a state and a goal as input and evaluating the utility of being in a state when aiming to reach a specific goal. HER uses this framework by adding the agent's experience to the replay buffer, replacing the agent's actual goal with one that was achieved in the episode[3] (pseudo goal). While vanilla HER randomly samples experiences from the replay buffer, (Fang et al., 2019) recognises that not all experience is equally valuable (reminiscent of experience replay versus Prioritised Experience Replay) and defines a method, curriculum-guided HER, that prioritises samples based on two main factors. The first factor is goal diversity, which encourages a higher variation of the goals to be replayed. On the other hand, the second criterion is proximity to the goal, calculated (alongside the first criterion) using a distance metric between states.

The advantage of the algorithms described in this section over other methods is that no additional MDPs need to be defined when generating a curriculum, and as such, no Transfer between tasks is needed. However, the following sections describe algorithms that can tailor the MDPs in the curriculum based on the agent's ability.

2.3.2.2 Curricula with restrictions

A popular category of methods in the Curriculum Learning literature imposes restrictions on the elements of the environment that can be modified when generating source tasks. The restrictions imposed by these methods reduce the search space over MDPs whenever automatically generating a curriculum, usually focusing on modifying a specific aspect of the target task. This section categorises each method based on the type of restriction imposed on MDPs in the curriculum.

Starting state distribution The earliest work in Curricula with restrictions can be traced back to at least Asada et al. (1996), where the principle of *Learning from Easy Missions* (LEM) was introduced when training a robot to shoot a ball into a goal utilising visual inputs. While previous work (Connell and Mahadevan, 1993) focused on using a divide and conquer approach to decompose the problem in independent sub-tasks (find the ball, dribble, shoot), LEM starts with initialising the

[3]The absorbing state reached by the agent.

agent close to the goal and then moving it further away in "steps" as the training advances. The experimental evaluation of this early approach showed the potential of Curricula created by modifying the starting state distribution; however, this method required a handcrafted set of tasks. The idea of a curriculum where the starting state progressively moves away from the goal as the agent's ability increases was also explored by (Florensa et al., 2017). The algorithm, named "Reverse Curriculum Generation" (RCG), assumes that a goal state is provided and gradually moves the start state away from the goal using a uniform sampling of random actions. The agent's performance from the given start states is then assessed to determine which states should be used to generate starts in the algorithm's next iteration, with the aim to initialise the agent in states that are neither solved nor unsolvable by the agent. In order to increase the stability of the algorithm, starts from previous iterations are stored in a buffer and re-used in later iteration (provided their performance is within certain bounds). This method automatically generates a curriculum that modifies the starting state distribution according to the agent's ability.

Goal generation Other methods instead focus on modifying the set of terminal states, in particular, Florensa et al. (2018) generates a curriculum by using the Generative Adversarial Networks (GAN) framework. This framework was introduced by Goodfellow et al. (2014), and it entails training two networks: a generator and a discriminator with opposite goals. The discriminator aims to distinguish between training samples and samples generated by the generator, which in turn aims to generate samples that the discriminator cannot distinguish from the training samples. The algorithm starts by utilising the generator network to set "Goals of Intermediate Difficulty" for the agent to reach, that is, goals that have an expected return between a minimum and a maximum threshold under the agent's current policy. The agent is then trained to achieve different goals on different episodes, made possible by the use of Universal Value Function Approximators (Schaul et al., 2015) (described in Section 2.3.2.1 when discussing HER). The goals the agent trained on are then labelled based on whether they are of intermediate difficulty and are used to train the GANs on both positive and negative examples, aiming to train the generator to output appropriate goals. This loop is repeated throughout the training and results in an automatically generated curriculum. Racaniere et al. (2019) introduces an alternative method for generating a goal-based curriculum centred around three components: a solver, a setter and a judge. The solver is simply the Reinforcement Learning agent, which is given a goal to reach at the start of each episode, and a

reward of 0 or 1 based on whether the goal was reached. The judge has the task to predict the probability of success of the solver on a given goal. Finally, as the name suggests, the setter generates goals for the solver based on an estimated probability of success. The setter is trained with three loss functions, each encouraging a different quality in the goals: validity, feasibility and coverage. The first loss function encourages the setter to select goals that the solver has already achieved. The second loss function ensures that the probability of success estimated by the judge matches the setter's estimation for the selected goals. Finally, the last loss function encourages variety in the selected goals.

Self-play Other methods for creating a curriculum with restrictions exist, such as self-play, where the general goal is to implicitly generate a curriculum by modifying the skill of the opponent faced by the agent. Early implementations of this kind of curriculum, like TD-Gammon (Tesauro et al., 1995), relied on the agent playing against itself; however, this method results in overfitting when applied to more complex domains. On the other hand, when using self-play on Go (Silver et al., 2016), the agent was matched against a randomly selected previous iteration in order to stabilise the training. Finally, more complex environments such as Starcraft (Vinyals et al., 2019) required more sophisticated self-play methods, such as introducing leagues of agents with different goals, such as exploiting weaknesses in the main agent's play.

Other paradigms Sukhbaatar et al. (2017) introduces a novel method of automatically generating a curriculum aimed at aiding exploration of environments with sparse reward functions. Agents are split into two entities: "Alice" and "Bob". Alice's role is to set a task, while Bob aims to learn by solving the task. Tasks are set by Alice performing a sequence of actions, and if the environment is reversible, Bob's goal is to reverse Alice's task by reaching the state Alice was initialised in, from the final state reached by Alice. If the environment can be reset to an arbitrary state, on the other hand, Bob aims to solve the task by reaching the same final state as Alice from the same initial state. Bob is rewarded by minimising the time taken to complete a task, while Alice is rewarded based on the difference of time taken to set the task against the time Bob takes to solve it. This type of curriculum, named "asymmetric self-play", enables the agent to learn about the environment's mechanics without a reward function. Pinto et al. (2017) on the other hand, aims to improve the robustness and stability of the training by using "Robust Adversar-

ial Reinforcement Learning". This method entails training an agent that aims to maximise a reward function while also training an adversary that aims to minimise it. The adversary acts in the environment by applying "destabilising forces", for example, if the goal is to keep a pendulum upright, the adversary will be able to push the pendulum in either direction; an alternative approach for developing an adversary could also focus on perturbating the agent's actions. As the adversary learns the task alongside the agent and becomes more skilled, the complexity of the tasks to solve increases, resulting in a curriculum being generated. (Riedmiller et al., 2018) introduces the "SAC-X" algorithm that aims to aid the environment's exploration by only changing the reward function in the final task. This is achieved by learning a scheduler that selects the sequence of auxiliary tasks to be learnt, and a set of policies or "intentions" to execute. Self-Paced Contextual Reinforcement Learning (Klink et al., 2020a) applies the same principles behind Self-paced Learning to Reinforcement Learning. This framework uses a contextual variable $c \in R^n$ to define a Markov Decision Process in a given domain, similar to a parameter vector in (Narvekar et al., 2016), although in this specific case limited to MDPs where transfer is not necessary between tasks. Assuming that the target context distribution is known, the agent is tasked with learning a policy and updating a context distribution, seeking to balance the current reward with the proximity to the target distribution. The work in Klink et al. (2020a) is then extended by Klink et al. (2020b), where the context distribution is updated separately from the policy (in a "block-coordinate ascent manner"), resulting in any RL algorithm being usable for learning. Finally, Milano and Nolfi (2021) defines a Curriculum Learning method to be used in conjunction with evolutionary algorithms. Their method divides different environmental conditions in various subsets based on the normalised performance of recently evolved agents. During the training, the agents will be presented with a randomly sampled task from each subset, ensuring the variety of the conditions experienced by the agents, and preventing overfitting. The curriculum is defined by a "difficulty function", that influences the composition of the subsets by changing the range of performance they represent. In the paper, difficulty functions are defined using power functions[4], as a way to increase the number of subsets at lower performances, biasing the selection of environmental variables towards tasks that are hard for the current population of learners.

[4]The paper uses x^n for $n \in [1, 4]$

2.3.2.3 Curricula with no restrictions

Whilst the methods discussed so far were designed with a specific set of restrictions in mind, this class of methods allows any MDP to be a source task in the curriculum. This is also the class of methods the work in this thesis belongs to. Not imposing restrictions on the MDPs to be included in the curriculum complicates the generation of a curriculum as the space of possible source tasks increases; however, it also results in a more flexible curriculum often needing Transfer Learning between different tasks.

Curricula with no restrictions can be traced back to at least Karpathy and Van De Panne (2012), where an Acrobot[5] is trained to perform various skills, such as hopping, flipping, rolling or a combination of the previous. This training is structured using a two-level curriculum, where the highest level specifies the order in which the skills are learnt, and the lowest level controls how each skill is learnt. The low-level curriculum includes three phases: "achieve", aimed at completing a skill; "explore", where the agent explores the neighbourhood of the successful experience by varying initial conditions and actions; and "generalise" aimed at generalising the experience collected in the exploration phase. However, it was not until Narvekar et al. (2016) that a general framework for Curriculum Learning in RL was outlined. This framework defines a domain D as a set of MDPs with various degrees of freedom (or parameters) f and a generator τ that generates an MDP given a specific domain and parameter vector. In this setting, a curriculum was defined as a sequence of MDPs to be learnt by the agent in a specific order before learning the target task. Whilst Narvekar et al. (2016) set out a framework for Curriculum Learning, the set of source tasks, their ordering and the amount of time to train on each task in the curriculum are assumed to be provided by a domain expert. Recent work by Manela and Biess (2022) showed how this type of pre-determined curriculum can be augmented with task-level curriculum methods, more specifically HER (Andrychowicz et al., 2017). In particular, they observed the poor performance of HER in tasks that require multiple sub-goals to be achieved before being successfully completed. They show the benefits of applying HER to each individual task within a higher level curriculum in domains that require reaching an object and throwing it towards a target, with the curriculum being composed of two tasks: reaching and throwing. While this section has so far only discussed curricula with a given set of source tasks

[5] "Planar, physically-simulated articulated figure that consists of two rigid bodies (links) connected by one actuated joint and an attached, fixed foot." (Karpathy and Van De Panne, 2012)

with a specified order, other work in no restrictions Curriculum Learning aims to further automate the generation of a curriculum.

The first part of curriculum generation that was automated (and the main focus of a considerable proportion of the literature in this field) is task sequencing. (Narvekar et al., 2017) aims to tackle the problem of automatically sequencing a pre-defined set of source tasks by defining the generation of a curriculum as an MDP. The state-space includes all the policies that can be represented by the agent, the action space is the set of all tasks that the agent can train on at any given time step, the transition function represents the change in the agent's policy before and after learning a task, and the reward function is formulated as to reward the lowest time to threshold. Once the curriculum generation is defined as an MDP, it is possible to use an RL technique to learn a solution. In this case, the algorithm, named Monte Carlo Approximation, learns all the source tasks until a computational budget is exceeded and adds the one that modifies the agent's policy the most to the curriculum. As such, it needs access to the agent's function approximator, which might not always be possible. This concern is addressed by Teacher-Student Curriculum Learning (Matiisen et al., 2017) which is based on a Partially Observable MDP formulation. The teacher component selects tasks for the student whilst learning, favouring tasks in which the student is making the most progress or that the student appears to have forgotten. The aim is for the agent to be able to solve all the tasks in the curriculum. The progress is measured through a change in reward on a task-by-task basis, thus not needing special access to the agent.

Sequencing a curriculum can also be formulated as a combinatorial optimisation problem, where the goal is to find the optimal sequence of tasks; as such, classic meta-heuristic algorithms can be used in this setting. Foglino et al. (2019b) compared four meta-heuristics on two domains concluding that trajectory-based methods[6], such as Filtered beam search (Ow and Morton, 1988) and Tabu search (Glover and Laguna, 1998) performed better than population-based methods[7], such as Ant colony optimisation (Dorigo et al., 2006) and Genetic Algorithms Goldberg and Holland (1988). Later, Foglino et al. (2019a) introduced a Curriculum Learning specific heuristic for task sequencing: Heuristic Task Sequencing for Cumulative Return (HTS-CR). This is a complete anytime algorithm converging to the optimal curriculum of a pre-defined maximum length. The algorithm starts by evaluating all curricula of length 2, and uses the information gathered in order to estimate which

[6]Heuristics that guide the search to certain areas of the search space.

[7]Methods that use multiple independent entities to search the search space.

tasks are better "heads" and which are better "tails" for the curriculum. Based on this initial estimate different curricula up to a maximum length are generated. The guarantee of optimality is derived from this algorithm eventually exploring all possible curricula up to the maximum length.

Another approach to automatic curriculum generation is to consider a curriculum as a Directed Acyclic Graph (DAG) where tasks are represented as nodes, and an edge from node a to node b represents task a being a source task for task b. Svetlik et al. (2017) defined a way to automatically generate a curriculum graph that relies on a metric named "transfer potential", aimed at evaluating the benefit of transferring knowledge from a given source task to a target task. This metric eliminates the need to learn each task individually before choosing which one should be added to a curriculum (like in Narvekar et al. (2017)). The first step for automatically generating a curriculum graph using this method is eliminating tasks whose transfer potential is below a given threshold and group them based on a (coarser) binary representation of a parameter vector. Once tasks with the same descriptor are grouped, they are connected within their respective group based on their transfer potential. Finally, edges between different groups are added, resulting in a completed curriculum graph. This framework lays the foundations to be able to have a further layer of automation in the creation of a curriculum by using Object-Oriented MDPs (OOMDPs), like in Silva and Costa (2018). OOMDPs introduce the concept of class to add a layer of abstraction to the state representation. Like in Object-Oriented Programming, a class has different attributes, each one with a different domain. Therefore, a particular state can be represented as a set of instances of different classes or the value of the attributes of each instance of the classes. This makes it possible to automate the creation of source tasks by randomly selecting objects from the target task to build a set of source tasks. It is assumed that a source task obtained using this process is easier to solve than the target task because it has fewer objects, and therefore the task should be less complex. While in Svetlik et al. (2017) transfer potential is defined as a function of the size of the state space and the applicability of the source task's value function to the target task, Silva and Costa (2018) expresses it as a comparison between the objects present in the source and target task.

While this class of methods gives the most amount of freedom in the source tasks that can be included in a curriculum, it also requires the definition of a Transfer Learning method to be used when applying the knowledge learnt in one source task to the next. The transfer method can be as simple as directly using the policy

or value function trained in one task to the following one. There are however some curricula where a simple transfer method is not applicable, such as whenever there is a difference in the state and/or action spaces of the MDPs in the curriculum. These cases require the use of more complex Transfer Learning algorithms, that might require additional domain knowledge (such as mappings between states and actions between different MDPs), and extensive testing for their fine-turning. In practice, this might result in more time and effort being used to define (and fine-tune) the Transfer Learning method rather than the Curriculum Learning method to be used.

2.3.2.4 Curriculum Learning with a Progression Function

The framework for Curriculum Learning that is proposed by this thesis looks at the problem of curriculum creation from a new angle: focusing on the complexity of the task the agent trains on and its evolution as the agent's abilities change during the training. The framework is placed in the class of curricula with no restrictions, and it splits the problem of curriculum generation in two sub-problems: defining the complexity of the environment, and generating tasks of the appropriate complexity. Unlike existing algorithms in the literature, this key division within the framework allows the introduction of high level domain knowledge in the task generation, while still having an online curriculum tailored to the agent's ability. Moreover unlike other curriculum learning algorithms, that tend to either fully automate all aspects of curriculum generation or require significant expert knowledge (e.g. manually constructed tasks), our framework can generate curricula that are either fully automated, manually generated, or a hybrid between the two. As will be discussed in Chapter 5, our method also introduces a new[8] version of the Multi-armed Bandit problem, and uses it to automate the generation of tasks of a required complexity.

[8] to the best of our knowledge

Algorithm	Restrictions	Task generation	Task selection	Task sequencing
Schaul et al. (2016)	Can only train on target task	Automatic (based on agent's experience)	All tasks within buffer are selected	Prioritising transitions based on TD error
Ren et al. (2018)	Can only train on target task	Automatic (based on agent's experience)	All tasks within buffer are selected	Prioritising transitions based on self-paced function and coverage
Andrychowicz et al. (2017)	Can only train on target task	Automatic (based on agent's experience)	All tasks within buffer are selected	Random
Fang et al. (2019)	Can only train on target task	Automatic (based on agent's experience)	All tasks within buffer are selected	Goal diversity and proximity to the goal
Asada et al. (1996)	Can only change the initial state	Domain experts	Domain experts	Domain experts
Florensa et al. (2017)	Can only change the initial state	Uniform sampling of random actions	Based on agent's performance	Random
Florensa et al. (2018)	Can only change the goal state	GAN	GAN	GAN
Racaniere et al. (2019)	Can only change the goal state	Setter	Setter	Setter
Tesauro et al. (1995)	Can only change the opponent	Automatic	Automatic	Automatic
Silver et al. (2016)	Can only change the opponent	Automatic	Automatic	Automatic
Vinyals et al. (2019)	Can only change the opponent	Automatic	Automatic	Automatic
Sukhbaatar et al. (2017)	Can only change the goal state	Automatic (Alice)	Automatic (Alice)	Automatic (Alice)
Pinto et al. (2017)	Can only oppose the agent	Adversary	Adversary	Adversary
Riedmiller et al. (2018)	Can only change the reward function	Scheduler	Scheduler	Scheduler
Klink et al. (2020b)	Can only train on target task's domain	Automatic	Automatic	Automatic
Milano and Nolfi (2021)	Can only train one domain	Domain experts	Based on difficulty function	Cycle through each subset and repeat
Karpathy and Van De Panne (2012)	No restrictions	Domain experts	Domain experts	Domain experts
Narvekar et al. (2016)	No restrictions	Domain experts	Domain experts	Domain experts
Manela and Biess (2022)	No restrictions	Domain experts	Domain experts	Domain experts
Narvekar et al. (2017)	No restrictions	Domain experts	CMDP	CMDP
Matiisen et al. (2017)	No restrictions	Domain experts	Teacher agent	Teacher agent
Foglino et al. (2019a)	No restrictions	Domain experts	Automatic	Automatic
Svetlik et al. (2017)	No restrictions	Domain experts	Automatic	Automatic
Da Silva and Costa (2018)	No restrictions	OOMDP	Automatic	Automatic
This thesis	No restrictions	Mapping functions	Progression functions	Progression functions

Table 2.1: This table contains all the algorithms mentioned in this chapter that are aimed at creating a curriculum in a RL setting. They appear in the same order they are discussed and the type of restriction imposed on the tasks that can be part of the curriculum, alongside the method used for generation, selection and sequencing are reported.

Chapter 3

Curriculum Learning Framework

As discussed in the background section, previous work has focused on gradually modifying the agent's experience within a given restricted set of MDPs (Sections 2.3.2.1 and 2.3.2.2) or scheduling sequences of different and increasingly complex tasks (Section 2.3.2.3). This chapter introduces a framework for Curriculum Learning that encompasses both paradigms, centred around the concept of task complexity and its progression as the agent becomes increasingly competent. The framework enables ample flexibility as the tasks can be selected from an infinite set without restrictions, and, most significantly, the task difficulty can be modified at each time step. Furthermore, this framework is learning-algorithm agnostic, as it focuses on modifying the environment according to the agent's ability and does not need access to the agent's internal state. The framework is based on two components: a *progression function* calculating the appropriate complexity of the task for the agent at any given time, and a *mapping function* modifying the environment according to the required complexity. The progression function encompasses task selection and sequencing, while the mapping function is responsible for task generation. Generation, selection and sequencing are the central components of Curriculum Learning (Narvekar et al., 2016), seamlessly integrated into the proposed framework.

Unlike previous work (discussed in Section 2.3.2), our method can directly leverage simple high-level domain knowledge in generating a curriculum for the agent. Even in highly complex domains, defining how the environment's complexity could be modified over individual dimensions is often straightforward. For example, if an agent's objective is to learn how to drive a car on the road, decreasing the friction

between the wheels and the ground, simulating slippery conditions will result in the environment becoming more complex. Likewise, adding a delay to the execution of actions to simulate a human's reaction time would also make the environment harder to solve. Our framework can leverage this simple domain knowledge to generate a curriculum for the agent over multiple dimensions of complexity.

This chapter starts with a formal definition of a curriculum, and then discusses the details of the proposed framework. Progression functions are then discussed in more detail, and the linear and Exponential progression are introduced. After explaining the details behind the generation and properties of mapping functions, some additional properties of our framework are introduced: progression in a setting where multiple parallel workers are present and curriculum granularity (how often the intermediate task is updated). Some special cases of our framework are also discussed, such as progression in a task-based setting, and using our framework in conjunction with reward shaping. Finally the details of the setting of the experiments used to evaluate approaches derived from the framework is clarified, and some preliminary experiments are undertaken in order to assess the potential of the proposed framework.

3.1 Curriculum

Before defining the specific elements of our framework, we first formally define a curriculum and the core components needed to generate one.

Let $\mathcal{M}$ be a possibly infinite set of MDPs that are candidate tasks for the curriculum, and $m_f \in \mathcal{M}$, be the *target* task. The target task is the task the designer wants the agent to learn more efficiently through the curriculum. We define a curriculum as a sequence of tasks in $\mathcal{M}$:

Definition *Given a set of tasks* $\mathcal{M}$*, a* curriculum *over* $\mathcal{M}$ *of length* l *is a sequence of tasks* $\mathcal{C} = \langle m_1, m_2, \ldots, m_l \rangle$ *where each* $m_i \in \mathcal{M}$*.*

The tasks in the curriculum are called the *intermediate* tasks. Each task in a curriculum is part of a family of MDPs called a *domain* $\mathcal{D}$, in which a specific task is defined by a parameter vector $\psi \in \mathbb{R}^n$. As our framework does not impose restrictions in the generation of a curriculum, intermediate tasks can be from any domain, provided that an appropriate transfer method is used when necessary. The curriculum aims to achieve the highest performance possible within the number of learning

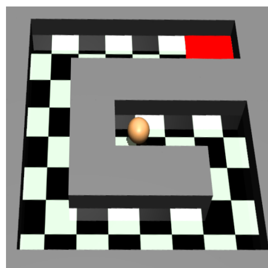

Figure 3.1: Visualisation of the Point Mass Maze domain. The red area in the top right of the maze represents the goal, and the curriculum is built by changing the starting position of the agent in the maze.

steps[1] allocated. Another aim of using a curriculum is to achieve this performance as quickly as possible. The specific performance metric optimised normally depends on the Curriculum Learning algorithm; however, to keep our framework as flexible as possible, we allow the optimisation of both environment-specific metrics (such as success rate in a navigation domain) and more traditional metrics such as cumulative reward. This will be expanded in Chapter 4.

3.2 Framework Details

As previously introduced, our Curriculum Learning framework comprises two elements: a *progression* function, and a *mapping* function. The role of the progression function is to specify the appropriate complexity vector at each time step, each component being between 0 and 1. The mapping function takes as input the complexity vector and generates an MDP with the desired complexity. A curriculum is generated over one or more dimensions of complexity, where each dimension corresponds to one or more parameters of a domain.

The Point Mass Maze domain[2] will be used as a running example for the rest of the chapter. The agent is required to navigate through a G-shaped maze, shown in Figure 3.1, to reach the goal in the top right corner of the maze. In this domain, the complexity of the environment is a one-dimensional vector representing the starting position of the agent in the maze, where the easiest possible task (of complexity [0]) has a starting position right next to the goal. In contrast, the most challenging task (of complexity [1]) has a starting position as far as possible, along the maze, from

[1]Refers to the computational budget, expressed as the number of actions, allocated for the training of the RL algorithm.

[2]Domain previously introduced by Florensa et al. (2017)

the goal. In this case, the first (and only) dimension of complexity corresponds to two parameters: the agent's initial x and y coordinates.

A progression function Π over n dimensions is defined as follows:

$$\Pi : \mathbb{N}^{+} \times \mathbb{P} \rightarrow [0,1]^{n} \tag{3.1}$$

where $\vec{c_t} = \Pi(t, \mathbb{P})$, is the *complexity vector* at time t, and $\mathbb{P}$ is a set of parameters specific to each progression function. The vector $\vec{c_t}$ reflects the difficulty of the MDP relative to each dimension where the agent trains at time t. The MDP corresponding to $\vec{c_t} = \vec{1}$ is the final task, whereas the MDP corresponding to $\vec{c_t} = \vec{0}$ is the least complex task in the set of candidate tasks $\mathcal{M}$ (where $\vec{x}$ is the vector of length n where each element is x).

To generate tasks of a specific complexity, we introduce the mapping function, which maps a specific value of $\vec{c_t}$ to a Markov Decision Process m created from the set $\mathcal{M}$:

$$\Phi_D : [0,1]^{n} \rightarrow \mathcal{M} \tag{3.2}$$

The co-domain of Φ_D contains the set of all possible intermediate tasks. The mapping function encodes some high-level knowledge about the domain, such as, in the Point Mass Maze domain: the further from the goal, the more complex the task. This knowledge is then used to generate a task of the desired complexity, mapping a complexity vector (e.g. [0.3]) to the MDP with the corresponding parameter vector (e.g. distance = 0.3 * maximum distance). Assuming that the learning time of an MDP is defined as the number of actions needed to converge to the optimal policy from a randomly initialised policy; for each dimension $i \in [0, n]$ an ideal mapping function generates the MDP with the lowest learning time when the complexity vector is $\vec{1} - \vec{e_i}$ [3] and the MDP with the highest learning time when the required complexity is $\vec{1}$. Moreover, given two MDPs with complexity vectors $\vec{1} - \vec{e_i} \cdot (1-a)$ and $\vec{1} - \vec{e_i} \cdot (1-b)$ with $a, b \in [0,1]$, the difference between a and b should be proportional to the difference in their learning time. Complexity is defined on each dimension independently, and it is up to the progression function to specify a curriculum in all dimensions.

At any change of the value of $\vec{c_t}$ according to the progression function, a new intermediate task is added to the agent's curriculum. Given a mapping function Φ, the value of $\vec{c_t}$ at time t corresponds to a new intermediate task:

[3] $\vec{e_i}$ is the vector of length n where the ith element is 1 and all other components are 0.

Algorithm 1 CL with a progression function

Inputs: *RL algorithm*, Progression function Π, Parameter set $\mathbb{P}$, performance function p, *domain knowledge*, *transfer learning method*

```
1: Generate mapping function Φ using domain knowledge
2: history := new list
3: t := 1
4: i := 0
5: m_0 := Φ(0)
6: current_c⃗ := 0⃗
7: while learning do
8:    s_t, r_t := result of RL algorithm's action in m_i
9:    insert (s_t, r_t) in history
10:   if p(history) is valid then
11:      c⃗_t := Π(t, p(history), ℙ)
12:      if c⃗_t ≠ current_c⃗ then
13:         current_c⃗ := c⃗_t
14:         m_{i+1} := Φ(c⃗_t)
15:         use transfer learning method from m_i to m_{i+1} on RL algorithm
16:         i := i + 1
17:      end if
18:      update mapping function Φ if necessary
19:      remove all elements from history
20:   end if
21:   t := t + 1
22: end while
```

$$m_i = \Phi(\vec{c_t}) \tag{3.3}$$

where i is the number of updates to the value of $\vec{c_t}$ since the start of the progression; specifying the way $\vec{c_t}$ changes over time through a progression function selects which tasks are added to the curriculum and in which order, resulting in the curriculum:

$$\mathcal{C} = \langle m_0, ..., m_i \rangle \tag{3.4}$$

Algorithm 1 shows how our approach is integrated with the agent's learning (line 8 is where the agent solves the MDP), and it considers all the extensions to the core framework described in Chapters 4 and 5. For the purpose of this chapter the reader can assume that: the condition at line 10 is always true, $p(history)$ does not have any influence on the complexity vector at line 11, and that line 18 can be ignored.

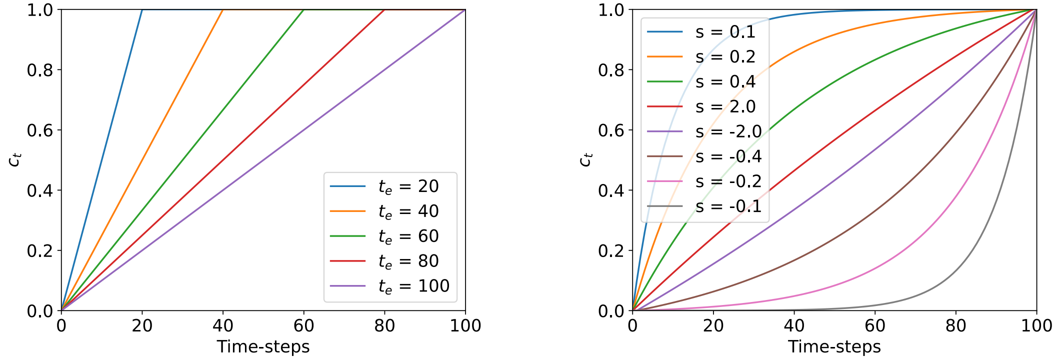

Figure 3.2: Example of different progression functions on an environment with one dimension of complexity. The figure on the left shows the effect of changing the value of t_e in a Linear progression. The figure on the right, on the other hand shows the effect of changing the value of s in an Exponential progression.

3.3 Progression functions

As mentioned above, progression functions determine how the complexity of the environment changes throughout the training, and in doing so, they generate a curriculum. The first class of progression functions introduced in this thesis is called *fixed* progression, and it includes progression functions that define the curriculum before execution. The most basic example of one such progression function is the *linear* progression. The only parameter in this function is the time step in which the progression ends, t_e. The equation of this progression function is as follows:

$$\Pi_l(t, t_e) := (\frac{t}{t_e})^{\top 1} \cdot \vec{1} \tag{3.5}$$

where $a^{\top b}$ is equivalent to the minimum value between a and b. Whilst in principle, each dimension of complexity could have its distinct value of t_e, this would make the parameter selection process difficult for environments with multiple dimensions of complexity. Therefore, this chapter defines fixed progression functions as only selecting tasks where each dimension has the same complexity. When using a Linear progression function, as the name suggests, the complexity of the environment will increase linearly until reaching a value of $\vec{1}$ at $t = t_e$.

We also introduce a second progression function, the *exponential* progression, with equation:

$$\Pi_e(t, \{t_e, s\}) = (\frac{\left(e^{\frac{t}{t_e}}\right)^{\alpha} - 1}{e^{\alpha} - 1})^{\top 1} \cdot \vec{1} \tag{3.6}$$

$$\alpha := \frac{1}{s} \tag{3.7}$$

Like the Linear progression, it includes a parameter to indicate when the progression should end, t_e, and a parameter s, which influences the progression's slope. A positive value of s results in the progression being initially steeper and slower towards the end of the training. On the other hand, a negative value of s results in the initial progression being slower. Figure 3.2 shows how different values of s affect the progression for one dimension by plotting the complexity over the course of the curriculum for several values of s.

In Equation 3.6 the numerator is responsible for the progression, starting at $\vec{0}$ when t $= 0$ and increasing as t increases. On the other hand, the denominator serves to guarantee that the value of $\vec{c_t}$ is equal to $\vec{1}$ as $t = t_e$.

Theorem *An Exponential progression converges to a Linear progression as s tends to infinity*

Proof The statement above can be expressed by the following limit

$$\lim_{s \to \infty} \Pi_e(t, \{t_e, s\}) = \Pi_l(t, t_e) \tag{3.8}$$

By substituting the equations of the two progression functions and accounting for the fact that $\alpha = \frac{1}{s}$ we get the following equation:

$$\lim_{\alpha \to 0}((\frac{\left(e^{\frac{t}{t_e}}\right)^{\alpha} - 1}{e^{\alpha} - 1})^{\top 1} \cdot \vec{1}) = (\frac{t}{t_e})^{\top 1} \cdot \vec{1}$$

In this equation, the clipping ($^{\top 1}$) is used only to ensure that the value returned by the progression functions is always in the $[0, 1]$ interval, however, if we prove that the left hand side equals to the right hand side without clipping their value, the proof will also be valid after the clipping is added. Moreover as the vector $\vec{1}$ multiplies both sides of the equation it can be simplified. These considerations result in the equation above becoming:

$$\lim_{\alpha \to 0}(\frac{\left(e^{\frac{t}{t_e}}\right)^{\alpha} - 1}{e^{\alpha} - 1}) = \frac{t}{t_e}$$

As the limit converges to the indeterminate form $\frac{0}{0}$, l'Hôpital's rule can be applied:

$$\lim_{\alpha \to 0} (\frac{\frac{t}{t_e}\left(e^{\frac{t}{t_e}}\right)^{\alpha}}{e^{\alpha}}) = \frac{t}{t_e}$$

The above equation converges to:

$$\frac{t}{t_e} = \frac{t}{t_e}$$

therefore proving Equation 3.8 and in turn the statement in the Theorem.

■

As a Linear progression function can be approximated by an Exponential progression function with a large value of s, our evaluation will only include the latter. However on more simple applications of a curriculum, a Linear progression might still be useful as it is easier to implement and only has one parameter.

3.4 Mapping functions

Alongside progression functions, our framework also relies upon mapping functions in order to generate a curriculum. As previously mentioned, mapping functions cover the task generation aspect of a curriculum. In fact, once the progression function specifies the desired complexity, the role of the mapping function in our framework is to generate the MDP associated with that complexity. This can be seen in Algorithm 1 at line 12, where m_i is the ith MDP in the curriculum. As mapping functions deal with each dimension of complexity separately, we will discuss the details in the generation of a mapping function only with respect to one dimension.

Mapping functions are also how high-level domain knowledge can be leveraged in creating a curriculum: to demonstrate this, we expand on the example previously introduced in Section 3.2, which dealt with creating a mapping function for a navigation domain. Let us assume the domain in question requires two parameters to define an MDP: the x and y coordinates at which the agent should be initialised. Let the domain knowledge we want the mapping function to represent be: the further away from the goal the agent is initialised, the more complex the MDP. Assuming a one dimensional complexity vector $\vec{c_t} = [d_t]$ this knowledge can be encoded by this simple mapping function:

$$\Phi(d_t) = M_{d_t * 10} \tag{3.9}$$

where M_x is the MDP where the agent's initial distance from the goal is x. It is

therefore sufficient to define a function converting distance from the goal to x and y coordinates in order to generate a mapping function for this domain. This is also an example where one dimension of complexity affects multiple parameters. This technique is useful, especially when there are parameters whose correlation, like x and y coordinates, simplifies the generation of a mapping over considering each dimension individually. In this example, there is also a linear correlation between complexity and distance, although this does not need to be the case. In the next chapter we will show that when progressing the complexity of the environment while taking into account the agent's performance, as long as the proposed mapping function does not violate the domain knowledge, the specific way the mapping function is defined does not greatly affect the learning speed of the agent.

If the monotonicity of a parameter's complexity can be assumed, the value of this parameter relative to the environment's complexity can be easily calculated using one of the two equations below, assuming $i \in [1, |\psi|]$ is the index representing a dimension in the parameter vector ψ:

$$\psi_i^h > \psi_i^e \rightarrow \psi_i^t = \psi_i^e + (\psi_i^h - \psi_i^e) * c_t^i \tag{3.10}$$

$$\psi_i^e > \psi_i^h \rightarrow \psi_i^t = \psi_i^e - (\psi_i^e - \psi_i^h) * c_t^i \tag{3.11}$$

where ψ_i^h is the value of parameter ψ_i corresponding to the most complex environment, ψ_i^e is the value of parameter ψ_i corresponding to the least complex environment, ψ_i^t is the value of parameter ψ_i at time t, and c_t^i is the component relative to dimension i in the complexity vector at time t. The advantage of generating mapping functions using the equations above is that if the assumption holds, the only thing that needs to be ascertained is whether the minimum value of a parameter a_{min} results in an easier or harder environment compared to the maximum value of the same parameter a_{max}. If the assumption does not hold naturally for a domain, for example, if the domain requires to specify the x and y coordinates, we would introduce distance as an abstraction of the coordinates of a state, satisfying the constraint above. If a domain did not allow for such an abstraction, the mapping function could still be generated not to violate the domain knowledge, although without using Equations 3.10 and 3.11. It is worth noting that the two simple equations defined above were used to generate all of the mapping functions in our domains and are applied independently to each dimension of complexity (or abstraction).

3.5 Progression with parallel environments

If it is possible to train with multiple instances of the domain simultaneously, such as whenever using a parallel implementation of an RL algorithm, each instance of the domain should have its own independent progression function. Having multiple progression functions allows for varying their parameters, resulting in the agent training on a range of complexities of the environment simultaneously. This also varies the experience of different learners and results in a higher stability of the training process, as the correlation of the updates between different learners is reduced.

When using the Exponential progression, all learners use the same t_e but vary the value of s, as in Figure 3.2. Once the number of processes is specified, a minimum and a maximum value for s are defined; firstly, two processes are assigned s corresponding to the minimum and the maximum. The rest of the processes then are assigned a value of s that ensures even spacing from the line $x = y$. For example, the four lines with $s > 0$ in Figure 3.2 would result from this parameter selection method using 4 processes. This technique also has the effect of requiring less variance in the parameter selection process. In fact, in all of the domains where multiprocessing is applicable, we used the same parameters for selecting s: the minimum value of s was always 0.1, and the maximum value of s was 2.

3.6 Curriculum granularity

Granularity is a key feature of a curriculum generated within our framework, and it is defined as the maximum frequency with which the complexity of an environment is changed during the curriculum. This is one of the factors that distinguishes the framework outlined in this chapter from the typical definition of a curriculum in the area of curricula with no restrictions. Our approach of defining a progression function as a continuous function generates a curriculum that can change the MDP up to every time step, resulting in a fine-grained curriculum. A coarse curriculum, on the other hand, would be the one defined by (Foglino et al., 2019a), where a task is trained for several episodes until convergence, resulting in a curriculum composed of a limited amount of source tasks. As curriculum granularity is a property that can be specified, our framework can be applied to both fine-grained and coarse curricula. While Algorithm 1 showcases a curriculum where a new intermediate task is generated up to every time step, line 9 could be modified in order to enforce

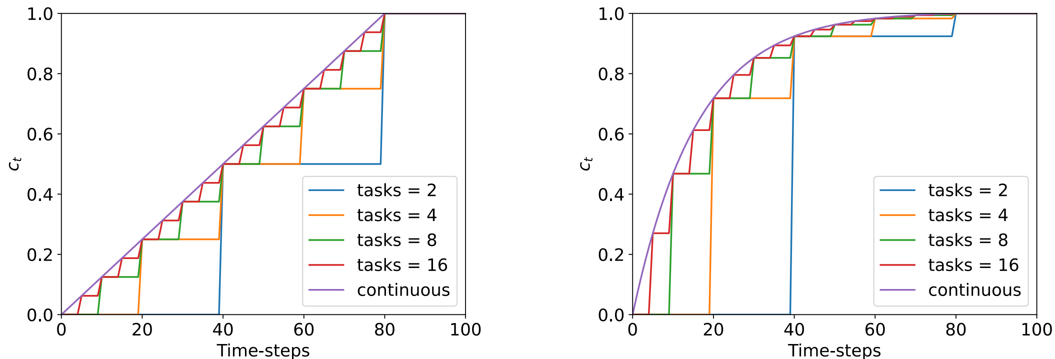

Figure 3.3: The figures show how the complexity of the task the agent is training on changes over time in curricula with different granularity. The curricula are generated by the linear (left) and exponential (right) progression functions, with the "continuous" one being the standard fine-grained curriculum defined by our framework. The legends report the number source tasks in each curriculum, where a jump in the value of c_t results in a new task being added.

a specific number of tasks by updating the intermediate tasks at set thresholds of complexity. This technique can be seen in Figure 3.3, where the complexity vector over a single dimension is plotted for curricula of different granularity generated by the linear and Exponential progression.

3.7 Special Cases

This section will discuss some special cases of this framework, in particular applying it to a task-based setting, and also how it can make non-potential reward shaping viable in an environment.

3.7.1 Progression in a task-based setting

The idea of a progression function can be applied to a"task-based" setting, where a curriculum is not created by a mapping function. By task-based setting we mean the kind of no restriction Curriculum Learning approach formalised in Narvekar et al. (2016), and continued by (Narvekar et al., 2017; Foglino et al., 2019a; Svetlik et al., 2017; Da Silva and Costa, 2018), where a curriculum is defined on a set of tasks that were meant to be learnt until convergence. However Narvekar et al. (2020) suggests that this is in fact unnecessary, and that a higher performance can be achieved by stopping the training early. An open question in this type of curriculum is therefore how long the agent should train on each source task before progressing to the target

task. Our framework, in particular progression functions, can be used to tackle this issue. Once a curriculum is known, one can use a one-dimensional progression function to specify when it's appropriate to switch the task the agent trains on. The range of a complexity vector $\vec{c_t}$ with only one dimension of complexity can in fact be divided into l parts, l being the number of source tasks in the pre-defined curriculum to execute. This would generate a set of thresholds, for example in a curriculum of four tasks the thresholds would be $[0.25, 0.5, 0.75, 1]$, that once reached cause a change in the source task the agent is trained on. Once the complexity reaches one, the agent is trained on the target task for a pre-defined length of time, like in other methods. This approach also allows for the value of $\vec{c_t}$ to be used to perform a progression within the task itself, only modifying a limited set of parameters.

Using this approach with fixed progression functions, that do not take into account the performance of the agent, is equivalent to specifying the amount of time steps to train on for each task. However, this approach can be used with the class of progression functions introduced in the next chapter (adaptive progression functions), that aim to tailor the curriculum to the agent's ability by using its performance when calculating the complexity of the environment. However, some source tasks might never converge to a performance that is sufficient to progress to the following task, such as when there is negative transfer. This would result in this approach paired with adaptive progression functions to never change the source task in this setting. One solution to this problem is to use this approach in conjunction with expert knowledge of the domain: one can still impose an upper bound on the number of episodes to spend in a specific environment, and move to the next task when either the limit has been reached or the progression function reached a threshold. If an upper bound was not available for each individual task, the upper bound for all tasks could be set to be the amount of time the target task is trained for, not needing any additional domain knowledge and still mitigating the issue of the curriculum never terminating.

3.7.2 Reward shaping

Another special case where our framework can be utilised is its application to reward shaping, in particular (but not limited to) non-potential based shaping. Reward shaping is a popular technique used to guide the agent by encoding domain knowledge in the form of a shaping function $F : S \times A \times S$. While the agent is training, the shaping function is added to the standard reward function of the MDP.

Potential-Based Reward Shaping (Ng et al., 1999) is a special case of reward shaping where the shaping function is obtained from the difference between the potentials of two states. If there exists a function $\phi : S \Rightarrow R$, called potential function, such that

$$F(s, a, s') = \gamma\phi(s') - \phi(s)$$

then F is a Potential-Based Shaping Function. If F is potential based, then Ng et al. (1999) proves that "every optimal policy in M' will also be an optimal policy in M (and vice versa)".

Whilst previous work in this area focuses on defining a type of shaping that does not alter the optimal policy in an environment, our framework can also be used in conjunction with non-potential based shaping functions in order to make them viable. A shaping function can in fact be seen as another parameter that can be altered by the curriculum and that decreases the complexity of the environment (by providing guidance for the agent). As the main concern whenever non-potential shaping functions are used is a change in the optimal policy as a result, our framework can be used to annihilate the shaping function over time. This can be achieved by treating the coefficient of the shaping function as an independent dimension of complexity:

$$R'(s, a, s') = R(s, a, s') + \vec{c_t}[d] \cdot F(s, a, s')$$

assuming that $\vec{c_t}[d]$ is the value of the complexity vector relative to the shaping dimension, F is a shaping function, R is the environment's reward function and R' is the reward function the agent will try to optimise. This allows for any shaping function to be used and integrated within a curriculum, by mitigating the risk of the shaping function potentially altering the best policy in a given domain.

3.8 Experimental setting

This section discusses the details behind our evaluation process. Firstly the six domains used for the evaluation of the work in this thesis will be introduced, then the learning algorithm used and the experimental method will be briefly discussed.

[5] Figure for the HFO environment edited from `https://github.com/LARG/HFO/blob/master/img/hfo3on3.png`

Figure 3.4: Visualisation of each one of the domains used in our testing. (From top left to bottom right: Grid World Maze, Point Mass Maze, Directional Point Maze, Ant Maze, Predator Prey and HFO[5]).

3.8.1 Evaluation domains

Our testing domains include two Grid World environments, three MuJoCo (Todorov et al., 2012) environments and Half Field Offense (Hausknecht et al., 2016). A visualisation for each domain is available in Figure 3.4.

Grid World Maze The first test domain used in this paper is a grid-world domain previously used by Foglino et al. (2019a), where the agent needs to reach a treasure while avoiding fires and pits. The agent can navigate this domain by moving North, East, South or West, and the actions taken in this domain are deterministic. This domain's reward function is as follows: 200 for reaching the cell with the treasure, -2500 for entering a pit, -500 for entering a fire, -250 for entering one of the four adjacent cells to a fire and -1 otherwise. In this domain, the episodes terminate when the agent reaches the treasure, falls into a pit or performs 50 actions. The combination of fires and pits used can be seen in Figure 3.4, where **F** represents a fire, orange squares represent a cell next to a fire, **P** represents a pit, **S** represents the starting position for the target task, and **T** represents the position of the treasure.

The state space for this environment is a 5 by 5, four channel image, where one channel is reserved for fires, one for pits, one for the treasure and one for the squares next to a fire. Finally, the action space is comprised of the four cardinal directions.

MuJoCo Mazes Three additional domains are MuJoCo (Todorov et al., 2012) environments where the agent needs to navigate a maze to reach a goal area. The environment's complexity is determined by two factors: the model the agent controls and the maze the agent needs to solve. We use two configurations from Florensa et al. (2017) and introduce a new configuration as a further domain.

The first model, Point Mass, is a sphere controlled by modifying its acceleration on the x and y-axis. The second model, Directional Point, is a sphere with a cube added to one end of the point as a directional marker. The point has two degrees of freedom: it accelerates in the marker's direction and rotates around its own axis. Finally, the third model, Ant, is a sphere with four limbs, with a joint each, attached to it. This results in the model having 8 degrees of freedom and the navigation task being very challenging. Each of these models, when used, requires its own state space to be defined. The state space associated with the Point Mass model is simply the x and y coordinates, the Directional Point Maze also specifies the orientation of the model, whereas the state space corresponding to the Ant Maze model is comprised of 15 values that specify its position and the angle of its joints.

We used two different mazes in our testing process, the more simple one shaped like an inverted C, with the goal in the top left corner, the second one being a G shaped maze with the goal in the top right corner. As previously mentioned, we used three distinct configurations of maze shape and agent models: like in Florensa et al. (2017), the Point Mass model was paired with the G shaped maze, and the Ant model was paired with the C shaped maze. We also introduced a new combination, the Directional Point model paired with the G shaped maze, to add a navigation task of intermediate complexity to our testing environments.

In these environments, the agent receives a reward of 1 when entering the goal area and 0 otherwise. The episodes terminate when the agent reaches the goal or after 300 (Point Mass and Directional Point) or 2000 (Ant) time steps.

A visualisation of these environments can be seen in Figure 3.4, where the agent is shown in the initial position associated with the target task, and the area highlighted in red is the goal the agent learns to reach.

Predator Prey A further domain is a grid-world domain where the agent needs to survive by eating stationary food sources while escaping a predator. The agent starts with 100 health points and loses one health point per time step and a further 100 health points if a predator attacks the agent (if they are on the same square). Eating food restores 10 health points and spawns a new food source at a random location, and the agent reaching 0 health points terminates the episode. The reward function used is the difference between the agent's health before and after each action. The state space is as an 11 by 11 three-channel array, and the action space is comprised of the four cardinal directions. An agent taking an action that would result in an out of bounds position results in the agent being stationary for that time step. If the agent can survive for 300 time steps, the episode terminates without further positive or negative reward. This domain can be observed in Figure 3.4, where **A** represents the agent, **P** represents the predator, and **F** represents a food source.

Half Field Offense The final testing domain is Half Field Offense, a sub-task in RoboCup simulated soccer (Hausknecht et al., 2016). In this domain, the agent controls the closest player to the ball and belonging to the attacking team on a football pitch to score a goal against the defending team. The playing area is restricted to half of the pitch, and the number of players on each team is set to two. The state-space for this environment is the "High level" state space described in Hausknecht et al. (2016), which is composed of 24 features. The action space includes five discrete actions: shoot, pass, dribble, move and go to the ball. A further layer of complexity is added because if the agent tries to pass, dribble or shoot without being near the ball, the agent will take no action for that time step. The reward function awards the agent a utility of 1 for scoring a goal and -1 when the ball goes out of bounds, is captured by the defending team, or the episode lasts for longer than 500 time steps. This domain can be seen in the bottom right corner of Figure 3.4.

3.8.2 Curriculum implementation details

The Exponential progression was evaluated on each one of our testing domains, and as mentioned in Section 3.5, in an effort to highlight the robustness of our method, whenever using multiprocessing (in all domains except for HFO) our parameters were the same. The value of t_e (the time at which the progression ends) was always 80% of the total training time, and the minimum and maximum values of s were

always 0.1 and 2 respectively. Section 3.5 provides more information on how to translate the minimum and maximum value of s to a set of values given a certain number of processes. In the HFO environment, as multiprocessing was not used, the value of s was equal to 1, whereas t_e was still 80% of the total training time.

The mapping function used for all of our four navigation domains mapped a one dimensional complexity vector $[c_0]$ to the MDP where the normalised distance of the agent's starting state from the goal was c_0. In the case of the Grid World Maze domain this distance was measured using the Manhattan distance, whereas in the MuJoCo navigation domains it was measured as a simple approximation of the distance to the goal following the maze. The domain knowledge used in the creation of these mapping functions is: the further from the goal the more complex the environment. The mapping function for these domains does not modify the state space of the MDPs but simply the starting state.

The mapping function in the Predator Prey domain contained two separate dimensions of complexity: the number of time steps the predator would stay stationary after attacking the agent, and the amount of food sources available. The domain knowledge in this environment is that the more food sources there are the less complex the environment, and that increasing the amount of time the predator is stationary for after attacking the agent decreases the complexity. The mapping function for these domains modifies the state space (number of food sources) and transition function (predator being stationary) of the MDPs in the curriculum.

Finally the HFO domain contained three dimensions of complexity: the starting distance of the ball from the goal, the starting distance of the ball from the sidelines, and whether the offending team started with the ball. The mapping function was generated by observing that; the closer to the goal the ball is the less complex the environment, the closer to the side line the more complex the environment, and finally that starting with the ball results in an environment of lower complexity. The mapping function for these domains does not directly modify the state space of the MDPs, however some parts of the state-space might not be reachable from every parameter setting.

It is worth noting that while our algorithm allows for more control over the environment, in the navigation domains only the starting position was modified in order to make the curriculum generated by our approach directly comparable to Reverse Curriculum Generation (Florensa et al., 2017). This comparison will take place in Chapter 4 once all of the progression functions generated using our framework will have been introduced.

3.8.3 Learning algorithm

All the domains in this paper use the same learning algorithm: Proximal Policy Optimisation (Schulman et al., 2017), in particular, the implementation contained in the "Stable-baselines" package (Hill et al., 2018). Moreover in all the domains except for HFO, the parallel implementation of the method was utilised, where multiple independent workers collect experience which is used to update the model in batches.

3.8.4 Experimental method

To provide statistically reliable results, each algorithm was executed 50 times for each domain except for Predator Prey, where each algorithm was executed 20 times. The results section will report the mean and 95% confidence interval for each algorithm's performance.

The metric of comparison chosen for the Grid World domain was the probability of the agent reaching the goal without falling in a pit or being on or next to a fire from the designated starting coordinates. In contrast, the metric of comparison chosen for the MuJoCo navigation domains was the probability of success from a uniform sampling of the states in the maze, previously used by Florensa et al. (2017). In the Predator Prey domain, the survival time was chosen as the evaluation metric; finally, in HFO, the metric used was the probability of scoring a goal.

The experiments were conducted using Python on a computer with a 16 core AMD Ryzen 9 3950X CPU, 32 GB of RAM and an NVIDIA GeForce RTX 2080 Ti GPU.

3.9 Results

In order to showcase the potential of the proposed framework we perform an initial evaluation on the Exponential progression, comparing it against a coarse 5 task curriculum generated using the technique described in the section above. We also compare our method to standard PPO in order to highlight the benefits of Curriculum Learning as a whole. Finally we perform an in-depth evaluation on the Point Mass Maze domain, being the MuJoCo navigation domain with the lowest training time, in order to precisely explore the relationship between granularity and performance.

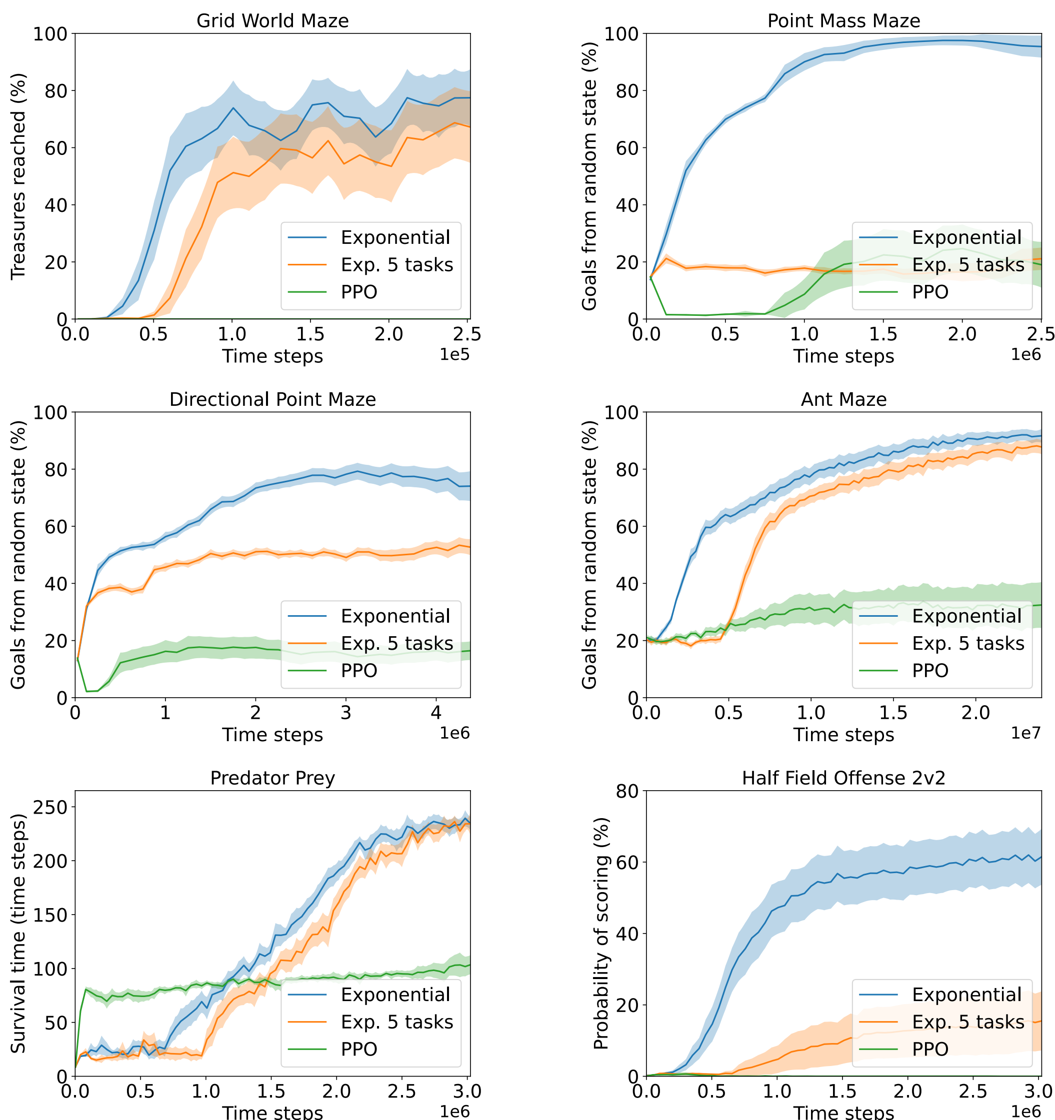

Figure 3.5: Performance of the Exponential progression, the Exponential progression limited to 5 tasks and PPO on six test domains. In these plots each time step represents an action made by the agent in the environment.

3.9.1 Comparison on all domains

The aim of this comparison is to motivate our approach of a curriculum without restrictions where the source task is continuously changing, by comparing two curricula generated using the same method, but differing only in the number of tasks allowed (coarseness of the curriculum). Comparing against standard PPO is also informative, as it shows that the domains used for our evaluation are indeed too com-

plex to be learnt outright. This comparison also shows the necessity of Curriculum Learning to make learning possible in complex domains, which is increasingly relevant in order to broaden the range of real world problems where RL can successfully be applied.

The first domain used for our testing was Grid World Maze, where standard Reinforcement Learning fails to learn a viable policy to reach the goal from the starting position. This result can be attributed to the same issue discussed in Section 2.1, where in order to reach the goal, the agent must successfully navigate past obstacles that yield a negative reward if encountered. While in principle this should not be a limiting factor for RL, as the observations received by the agent are in the form of a picture of the surrounding squares, a function approximator is necessary to act in the environment. With the introduction of a function approximator the guarantees of optimality of tabular RL are lost, and the agent's policy converges to one that avoids the obstacles rather than one that aims to reach the goal. In this domain the Exponential progression where the source task is allowed to change at every episode performs marginally better compared to its less granular counterpart. The standard Exponential progression outperforms the more granular one in the first third of the training, however afterwards their 95% confidence intervals overlap (albeit the mean of the standard Exponential progression always outperforms the more granular curriculum). This is the domain where having a fine-grained curriculum seems to have the least benefit over a coarser grain curriculum, which could be caused by the fact that being this a Grid World domain the distance to the goal is a discrete parameter between 1 and 15. Even the more granular curriculum therefore is limited to 15 source tasks.

The experiments in the MuJoCo navigation domains are consistent with respect to the performance of standard PPO, in that it only marginally improves over a randomly initialised policy. The Point Mass Maze domain is the one where the difference between a fine-grained curriculum and a coarse curriculum is the most evident, resulting in the fine-grained curriculum converging to a near-optimal performance, whereas the coarser curriculum does not significantly improve over a randomly initialised policy. The difference between the two curricula is reduced in the Directional Point Maze domain, where the coarser curriculum provides an improvement over standard PPO, however is clearly outperformed by the finer-grained curriculum. Finally in the Ant Maze domain the best performance of the fine-grained Exponential progression is higher compared to its coarser counterpart; whereas there is a considerable difference in performance over the first part of the training, the

two methods get closer in performance in the later stages.

It is worth noting that in both the Point Mass Maze and Directional Point Maze domains the performance of the finer grained curriculum drops in the final stages of the training. This is attributed to the progression being sometimes too fast for the agent: the drop in performance always happens at 80% of the total training time, when the progression is set to end and the agent trains only on the target task. If this shift happens too quickly, it might result in the agent not being able to get to the goal and slowly forgetting how to solve the maze. This is one of the motivations behind the need of a curriculum that is not pre-determined, but that can adapt to the agent's performance.

In the Predator Prey domain, PPO's performance is initially higher compared to the two curricula, however it converges to a survival time of 100 almost immediately. This behaviour can be easily explained by observing the properties of the domain: the agent starts with 100 health points and one hp is deducted after every action. PPO is once again stuck in a local optimum where it learns to evade the predator, but does not learn to collect food sources to prevent the agent from starving. In this domain, the two curricula's confidence intervals overlap at the end of the training, with the fine-grained curriculum outperforming the coarser grained curriculum initially.

Finally in the HFO domain both PPO and the more granular Exponential progression fail to learn a viable policy, resulting in the fine-grained Exponential progression outperforming both in this environment.

Both types of curriculum analysed in this experimental section perform a change in the intermediate task up to the end of every episode, with the "5 tasks" version being even more limited. This implies that the complexity value needs to be calculated at most after each episode, which adds a negligible computational time to the training. Given that the exponential progression can be calculated using Equation 3.6, no values need to be stored in memory (besides the parameter s). The only environment where there is a time penalty for changing the parameters in the environment is HFO, although this is again negligible when compared to the total training time.

The initial evaluation performed on the Exponential progression shows the potential of the novel framework introduced in this chapter, in particular the fine-grained curriculum outperformed the coarser curriculum in 4 out of 6 testing domains with respect to the final performance, but in all domains with respect to the area under the performance curve. This is a key metric especially in critical tasks

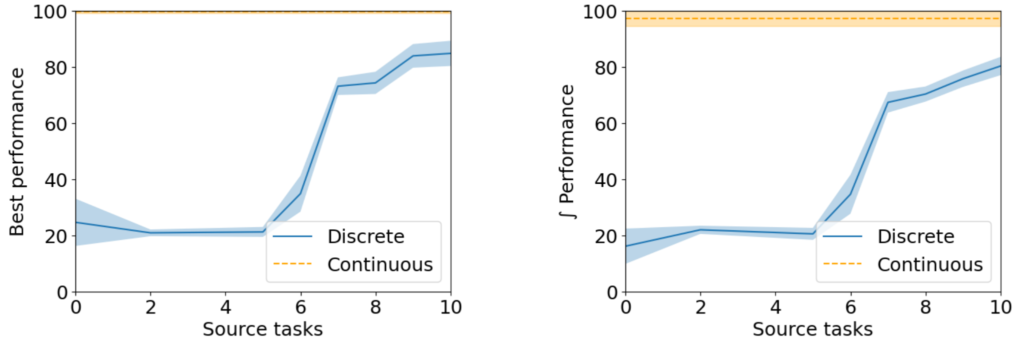

Figure 3.6: Performance of the Exponential progression on the Point Mass Maze domain varying the granularity of the curriculum. The plot on the left reports the best performance achieved by each method, whereas the plot on the right represents the area under the performance curve for each method (normalised with respect to the best performing method plus its confidence interval).

(Foglino et al., 2019a) where sub-optimal exploration is costly.

3.9.2 In depth analysis

In order to better establish the relationship between complexity and performance, we tested curricula of 1 to 10 tasks on the Point Mass Maze domain and reported the results in Figure 3.6. The performance of PPO without a curriculum is also reported in these two plots as a "0 task curriculum". The plot on the left shows a strong correlation between the amount of source tasks in the curriculum and the best performance achieved by the agent on the target task, the highest difference being between 6 and 7 source tasks. In this environment the fine-grained curriculum converges to almost the optimal performance (97.5 %), and with no time penalty or memory overhead introduced compared to its coarser counterpart. An interesting aspect of this plot is the curricula with a very low amount of source tasks actually hurting the agent's performance, probably a result of the sudden change in complexity between the tasks themselves.

The plot on the right reports the area under the performance curve for each curriculum, again showing a correlation between number of tasks and increase in this metric. We chose this metric as a substitute of the "total reward" metric introduced in Section 2.2.4, as in our case we always test on the target task, even when training on other tasks. The area under the performance curve is therefore representative of the performance of the agent throughout the training without the limitation of only being measurable when training on the target task. It should be noted that while

the final performance of PPO was higher when compared to curricula with a low amount of tasks, the area underneath the performance curve was lower; this can be explained by the plot Figure 3.5 (top right), where the performance of PPO falls to zero in the initial part of the training, unlike the curricula. As PPO needs to learn to solve the maze starting from the center, the initial performance dropping could be a result of it not ever reaching the treasure and therefore not knowing how to act in the last part of the maze. The plot on the right also shows an almost linear increase in the area under the performance curve in curricula with 7 to 10 tasks, whereas the increase in final performance was more pronounced between 8 and 9 tasks. Overall these experiments highlight the benefit of being able to generate a fine-grained curriculum by using our framework.

Chapter 4

Adaptive progression

While the last chapter introduced our framework and a way to generate pre-determined curricula, it does not consider that each time a randomly initialised agent learns in an environment, its performance will be different. A fixed progression function can only modify the complexity of the environment without any information about the agent's ability. This limitation can lead to the agent's learning being halted and even reversed if the progression is too fast. This problem is especially highlighted in the MuJoCo maze domains, where the agent needs to learn to turn corners to reach the goal. If the agent does not learn how to turn a corner, but the fixed progression keeps increasing the complexity of the environment, it might lead to the agent reaching an MDP it cannot solve with its current knowledge. A further increase in complexity would result in the agent eventually forgetting what it already learned, as it would always receive a reward of zero (not reaching the goal) for all episodes. A practical example of this phenomenon can be seen in the Directional Point Maze domain with the Exponential progression (Figure 3.5), where the agent's performance drops in the later parts of the training.

In order to overcome this flaw, this thesis introduces the concept of *adaptive* progression functions, that are generated online and adapt to the agent's performance in order to generate a curriculum. Such progression functions aim to keep the agent in tasks at the edge of its ability, a well-established concept in human learning called "zone of proximal development" Vygotsky (1980). Setting the complexity of the environment at the edge of the agent's ability causes the agent not always to succeed and not always fail, allowing for learning from both positive and negative examples.

This chapter starts with the definition of performance functions used to assess how well the agent can solve the current task. Then, two adaptive progression func-

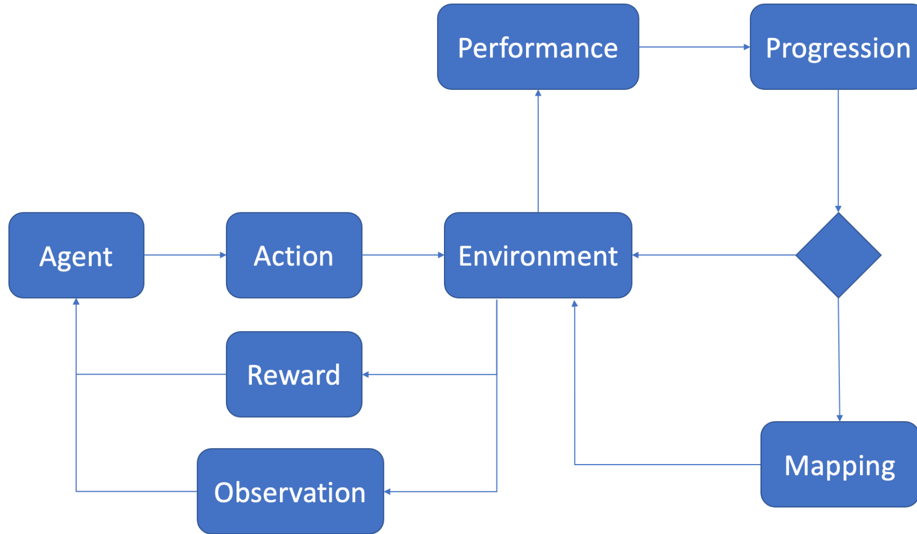

Figure 4.1: Integration of adaptive progression functions within RL. After taking an action the performance is calculated if possible, and passed on to the progression function. The reward and observation are then provided to the agent. Finally, the complexity vector is passed onto the mapping function that generates the updated environment if the environment needs to be modified.

tions are introduced, the Friction-based progression and the Rolling Sphere progression. The latter also allows to progress each dimension independently, which results in more control in the curriculum generation process. Finally, the experimental setting is discussed. An extensive experimental evaluation is performed to evaluate the benefits of adaptive progression functions and compare the methods derived from our framework against current state-of-the-art Curriculum Learning algorithms.

4.1 Performance functions

As mentioned in the introduction to this chapter, adaptive progression functions use information about the agent's performance to automatically determine the complexity of the environment. However, this performance can be expressed through a range of different metrics. For example, in the Predator Prey domain, one metric that could be used to assess the agent's performance is survival time, whereas another is the reward gathered by the agent. Because of the reward function used, which is the difference between the agent's health before and after an action, any

agent that can escape the predator but starves will gather a total reward of -100 during the episode. As this occurs regardless of whether its survival time is 100 or 299, it is not a good indication of the agent's ability in this domain. Motivated by this example, various domain-specific metrics can be used to assess the agent's performance in progression functions, such as survival time or whether the agent reached the goal in a maze domain; more traditional metrics can also be chosen, such as the reward, or total reward over an episode. We, therefore, define the *performance function* p as the metric used to evaluate the agent's performance in a given MDP.

Needing to calculate the value of the relevant performance function adds another element to the core framework described in Chapter 3. It also influences the time frame with which a progression function changes the complexity vector: some performance functions such as survival time can only be calculated at the end of an episode, whereas if the chosen performance was the reward, the complexity vector could be updated up to every time step while the episode is evolving. This was made explicit in the previous Chapter in Algorithm 1 (line 10), and it was referred to as the validity of the performance function. Algorithm 1 represents performance functions as functions of the history of states and rewards gathered by the RL algorithm while acting in the environment. However, in practice, the amount of information from the history used by the performance function will vary on a case-by-case basis. Accounting for this and in an effort to simplify the notation, for the rest of this thesis I will refer to the value of the performance function at time t as p_t, instead of explicitly stating $p(history_t)$. This is also consistent with what happens in practice, where if values of the performance function at different times are needed only the values themselves are stored, and not the whole history at each time-step.

The key role played by performance functions within the framework defined in this thesis can be seen in Figure 4.1, where a change in the complexity vector only happens if the performance function can be calculated.

4.2 Friction-based progression

The first adaptive progression function that was created as part of the research in this thesis is the Friction-Based progression. Being an adaptive progression function, it changes the complexity of the environment according to the agent's performance (defined by a performance function as seen above). This process, illustrated in Figure 4.2, starts by measuring the change in the agent's performance in the last i time

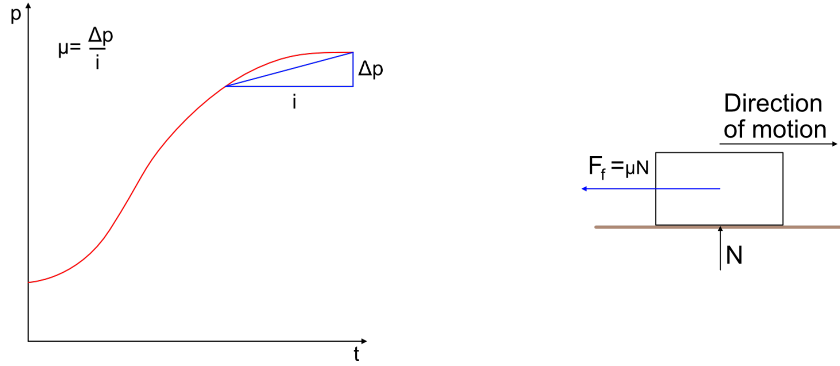

Figure 4.2: Plot representing the *Friction-based* progression. The change in performance of the agent is measured over an interval; this value is then used to calculate F_f, the friction force that slows down the object as the agent improves. This, in turn, results in the complexity of the environment increasing.

steps. This change is then divided by the magnitude of the interval i to get the rate of change of the agent's performance over the interval, μ. To utilise this information in creating a curriculum, we introduce the model of a box sliding on a plane (Figure 4.2 right) and use its speed to calculate the environment's complexity. The intuition is that the relationship between speed and complexity is $complexity = 1 - speed$. This model allows us to utilise μ as the friction coefficient between the body and the plane, resulting in the object slowing down and the complexity increasing as the agent's performance increases.

The intuition behind this progression function is that the more skilled the agent is at solving the current task, the quicker the agent should progress to a more difficult task. The physical model used provides a convenient way to translate a change in the agent's performance to a change in the difficulty of the environment, as can be seen in the equation below:

$$\Pi_f(t, m, s_{t-1}, s_{min}, p_{t-i}, p_t, i) = \vec{1} \cdot (1 - Uniform(s_t, s_{min})) \tag{4.1}$$

$$s_t = (s_{t-1} - m \cdot g \cdot \mu_t)^{\top 1}_{\perp 0} \tag{4.2}$$

$$\mu_t = \frac{p_t - p_{t-i}}{i} \tag{4.3}$$

where m is the mass of the object, g is the gravitational acceleration, s_t is the object's speed at time t, s_{min} is the minimum speed reached by the object, p_t is the agent's performance at time t, i is the time interval considered by the progression function, $a^{\top b}$ is equivalent to the minimum value between a and b and $a_{\perp b}$ is equivalent to

the maximum value between a and b. Equations 4.2 and 4.3 describe our physical model whose speed is adjusted according to the agent's performance, while Equation 4.1 specifies the relationship between the speed of the model and the complexity of the environment. To respect the constraint that $\vec{c_t} \in [0,1]^n$ (posed in Section 3.2), the speed of the object is clipped to fall within that same interval, as seen in Equation 4.2. The model used in our approach is not strictly physically accurate, as it considers the possibility of a "negative friction" (the force is applied in the direction of the object's motion), where the speed of the object increases. This is the desired behaviour because if the progression function cannot decrease the value of any component of a complexity vector, the agent might reach environments that are too difficult to solve. Without the possibility to revert to a solvable environment, the agent's policy or value function could deteriorate. In order to mitigate the risk of this happening, we introduced a uniform sampling in Equation 4.1, which takes place whenever the agent's performance drops. A drop in performance would result in the speed of the object increasing ($s_t > s_{min}$), which in turn would decrease the average complexity of the environment, but still allow the agent to train on a range of different complexities. The effect of the uniform sampling can be observed in Figure 4.3, especially when $i = 20$, where the complexity of the environment oscillates as a result of the performance decreasing. An example of the effect the value of i has on the progression can also be seen in Figure 4.3 (left). In the same Figure, the plot on the right shows the performance function used to generate the Figure on the left. The performance function was chosen to have an underlying linearly increasing trend but masked by adding a periodic function. This performance function was used to showcase the behaviour of the Friction-based progression in a controlled (manually generated performance function) but non-trivial scenario. This plot highlights how progressions with small intervals tend to react quickly to a change in the agent's performance. On the other hand, progressions with larger intervals provide a more constant increase in complexity.

Equation 4.3, used to calculate the friction coefficient μ_t, relies on the performance function's past values, which implies that Equation 4.1 is only defined when $t = i + 1$. However, not changing the complexity vector for the first i time steps would be highly detrimental to our approach, as the intervals used are often a not negligible percentage of the total training time. This is the motivation behind our choice to start the training at $t = i + 1$ and to consider the agent's performance for $t \leq i$ to be equivalent to the performance the agent would have when not performing any actions; this allows us to start the progression immediately. A further motiva-

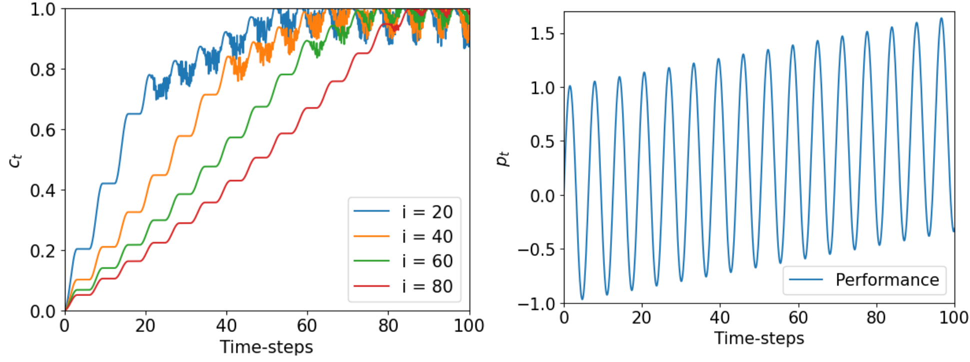

Figure 4.3: The figure on the left shows the effect of changing the magnitude of the interval, i, on a Friction-based progression, where the performance used for the progression is shown in the figure on the right.

tion behind this choice lies in the fact that the only values of p that influence when a progression ends ($\vec{c_t} = \vec{1}$) are $\{p_1, ..., p_i\}$ and $\{p_{t-i}, ..., p_t\}$. The following theorem supports this statement:

Theorem *Let $\bar{p}_{[x,y]}$ be the average value of the performance function between $t = x$ and $t = y$. Then the Friction-based progression with mass of the object m ends when:*

$$\bar{p}_{[t+1-i,t]} - \bar{p}_{[1,i]} = \frac{1}{mg} \tag{4.4}$$

Proof. To find the necessary conditions in order for the Friction-based progression to end we derive an alternative definition of Equation 4.2 where we expand s_{t-1}. Firstly, combining Equations 4.2 and 4.3 results in the following equation:

$$s_t = s_{t-1} + mg\frac{p_{t-i} - p_t}{i}$$

As mentioned previously, to avoid big fluctuations in the complexity of the environment, we start the progression at time $t = i + 1$. This implies that at $t < i + 1$, the object's speed does not change as no progression is taking place, resulting in the object's speed before then being equal to 1. This information can be used to change the equation above to:

$$s_t = 1 + \sum_{j=i+1}^{t} mg\frac{p_{j-i} - p_j}{i}$$

This equation can then be rearranged in the following way:

$$s_t = 1 + \frac{mg}{i} \sum_{j=i+1}^{t} p_{j-i} - p_j$$

$$s_t = 1 + \frac{mg}{i} (\sum_{j=1}^{t-i} p_j - \sum_{j=i+1}^{t} p_j)$$

$$s_t = 1 + \frac{mg}{i} (\sum_{j=1}^{i} p_j + \sum_{j=i+1}^{t-i} p_j - \sum_{j=i+1}^{t-i} p_j - \sum_{j=t-i+1}^{t} p_j)$$

$$s_t = 1 + \frac{mg}{i} (\sum_{j=1}^{i} p_j - \sum_{j=t-i+1}^{t} p_j)$$

$$s_t = 1 + mg(\frac{\sum_{j=1}^{i} p_j}{i} - \frac{\sum_{j=t-i+1}^{t} p_j}{i})$$

As mentioned in the proof statement above, we introduce the notation $\bar{p}_{[x,y]}$ for the average value of the performance function between $t = x$ and $t = y$. We can therefore use this notation on the equation above to simplify it to:

$$s_t = 1 + mg(\bar{p}_{[1,i]} - \bar{p}_{[t-i+1,t]})$$

By using this alternative formulation of Equation 4.2 we can finally assert that $s_t = 0$ (the progression ends) when:

$$\bar{p}_{[t-i+1,t]} - \bar{p}_{[1,i]} = \frac{1}{mg}$$

■

The theorem above can also be used to simplify the parameter selection for our algorithm. Equation 4.4 can be solved for m and be used to determine its appropriate value, resulting in the progression ending when a specific average performance is reached for i time steps. This results in the Friction-based progression only having two parameters in practice: the magnitude of the interval and the performance after which the agent should start training on the target task. In fact, s_{t-1} and s_{min} are initialised at one and kept in memory throughout the training, and p_t is provided to the progression function at every time step and stored in memory for i iterations.

If the environment or the curriculum were to require the progression to be monotonic, such as if some forms of transfer are involved, Equation 4.1 can be modified in the following way:

$$\Pi_{fm}(t, m, s_{t-1}, s_{min}, p_{t-i}, p_t, i) = \vec{1} \cdot (1 - s_{min}) \tag{4.5}$$

An example of a transfer learning algorithm that might require this monotonic progression is policy transfer using reward shaping (Brys et al., 2015). This method trains a randomly initialised policy using a potential based shaping function influenced by the policy to be transferred. If this method of transfer was necessary once a complexity threshold was reached and the agent's performance decreased, the uniform sampling in Equation 4.1 could result in the complexity decreasing below this threshold. This, in turn, would result in having to train a new randomly initialised agent, making it inefficient to reset the agent's weights potentially every episode. As mentioned above, Equation 4.5 mitigates this problem by not allowing the complexity of the environment to decrease.

The space complexity of the Friction-based progression is $\Theta(i)$, as every value of the performance function in the interval needs to be stored. Moreover, the time complexity of one iteration of the Friction-based progression is $\Theta(1)$, as the value of s_{min} is stored after every iteration and does not need to be re-computed. To accurately analyse our approach's time complexity, one also needs to consider the complexity of the mapping function, which is domain dependent.

4.3 Rolling Sphere progression

Whilst the Friction-based progression considers the value of the performance function over an interval, we also created the Rolling Sphere progression, whose aim is to change the complexity of the environment only using the last two values of the performance function. The main benefit of this progression function is that it is a lot more sensitive to sudden changes in the agent's performance. This property will later be helpful, especially when progressing on multiple dimensions simultaneously.

The Rolling Sphere progression is based upon yet another physical model: a sphere rolling without sliding on an inclined plane. As can be observed in Figure 4.4, the angle of the plane changes based on the difference between the two latest values of the performance function. The sphere then accelerates on the inclined plane for a pre-defined amount of time, and its x coordinate on the plane is used to derive the complexity vector for the environment. This progression function has a different set of properties compared to the Friction based progression: while in the former, a steady performance would result in the complexity not changing, the

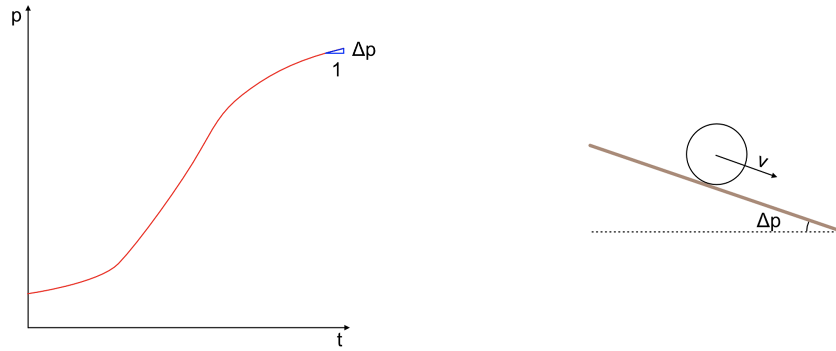

Figure 4.4: Plot representing the *Rolling Sphere* progression. The change in performance of the agent is measured over one step; this value is then used as the angle in radians of the plane the sphere is rolling on. The sphere's position is then used in order to specify the complexity vector.

Rolling Sphere would not be accelerated in any direction as the plane would be flat, resulting in preservation of the current speed.

The sphere is initialised with both a speed and x coordinate on the plane of zero. As complexity is bound between 0 and 1 by definition (Equation 5.1), if the sphere's x coordinate is less than one, its speed and position are set to zero. On the other hand, if the sphere's x coordinate exceeds one, its position is set at one, but its speed is preserved.

This progression function requires a normalised performance function between 0 and 1 to be effective. This requirement is imposed by the difference in performance being used to calculate the plane's angle. It also will allow for some approximation of the model to increase the stability of the progression. In practice, once the minimum and maximum values of the performance function are provided, the normalisation happens within the progression itself.

While the Friction-based progression's core concept was directly applicable to a practical scenario, the Rolling Sphere progression presents some additional challenges. Firstly, using a more physically accurate formula to calculate the acceleration of the sphere could result in the complexity being reduced even if the agent were to achieve the highest performance on the environment. This behaviour is undesired, as it could make progressing to the target task an impossibility. This behaviour is a direct result of the acceleration given Δ_p not being an additive function. In order to be additive, a function needs to satisfy the following equation:

$$f(x + y) = f(x) + f(y) \tag{4.6}$$

Given that the acceleration of a sphere rolling (whilst not sliding) on a plane for a certain difference in performance Δ_p can be calculated using the following formula:

$$acc(\Delta_p) = \frac{5}{7}\ g\ sin(tan^{-1}(\Delta_p)) = \frac{5}{7}\ g\ \frac{\Delta_p}{\sqrt{1+\Delta_p^2}} \tag{4.7}$$

It is evident that the function is not additive (the condition in Equation 4.6 does not hold):

$$\frac{5}{7}\ g\ \frac{x}{\sqrt{1+x^2}} + \frac{5}{7}\ g\ \frac{y}{\sqrt{1+y^2}} \neq \frac{5}{7}\ g\ \frac{x+y}{\sqrt{1+(x+y)^2}} \tag{4.8}$$

To illustrate why this might be an issue preventing a stable progression, assume the agent's performance keeps following the pattern $0, 0.5, 1, 0, 0.5, ...$ Assuming the acceleration is applied to the sphere over one second of simulation, the sphere's speed would start at 0, then increase to 3.13 as the performance reaches 0.5, then increase again to 6.26 as the performance reaches one. However, when the performance goes back to zero and $\Delta_p = -1$ the speed will only drop to 1.32. For each cycle, the speed when the performance is zero will increase by that amount, and as the performance is bounded between zero and one, no lower speed value would be achievable (unless the loop was reversed). This is an issue as the complexity of the environment should not keep increasing whenever the agent cannot solve the current one. On the other hand, if the pattern was reversed, the speed would decrease by 1.32 every iteration, eventually making it impossible to progress to a more complex environment even if the agent's performance is one (the maximum value). This problem is preventable by substituting Equation 4.7 with an approximation that is additive, while not deviating far from the original model. The approximation used is linear and is defined by the following equation:

$$acc(\Delta_p) = 4.95\ \Delta_p \tag{4.9}$$

which is clearly additive:

$$4.95\ x + 4.95\ y = 4.95\ (x+y) \tag{4.10}$$

while also being a close fit for Equation 4.7 in the $[0, 1]$ range, as evidenced by Figure 4.5 (right).

While using a linear function to approximate the acceleration of the physical model eliminates the instability in the progression, it poses a new problem: the

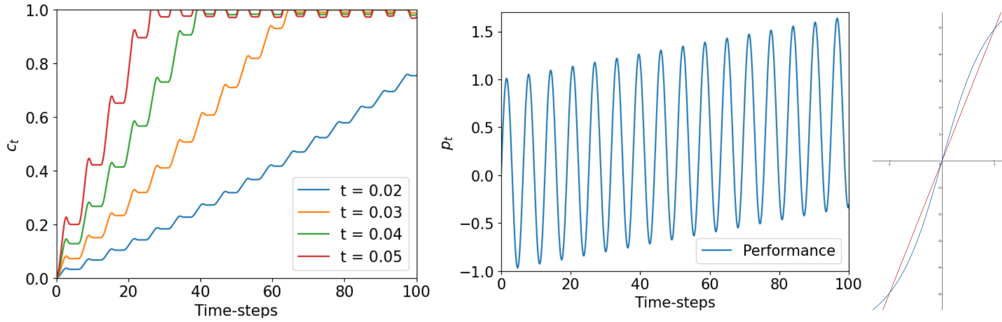

Figure 4.5: The figure on the left shows the effect of changing the simulation time, t, on a Rolling Sphere progression, where the performance used for the progression is shown in the figure on the center. The figure on the right shows the linear approximation (in red) of the physically accurate function (in blue) used to derive acceleration from a certain Δ_p (with a 2:1 ratio of the x axis w.r.t the y axis).

impossibility of backwards progression. As the speed and position of the object are initialised at zero, and the first value of the performance function is also assumed to be zero, this is the lowest speed achievable. This might not be desirable: a backwards progression is helpful, especially if the agent forgot what it has already learnt. A solution to this problem is making the rolling sphere more sensitive to accelerating in the direction opposite to its velocity. More specifically, whenever the sphere's velocity and acceleration have opposite signs, the acceleration is multiplied by a coefficient to allow for a "backwards" progression to occur. In order to ensure that the additional energy introduced in the system does not have the same destabilising effect of the issue discussed above, the speed resulting from this increase in the acceleration is tracked and released back into the system over a more extended period of time.

As previously mentioned, whenever using this progression function the performance function is normalised between zero and one. As such, the maximum and minimum values of the performance function are required as an input of the Rolling Sphere progression. Only one other parameter is necessary: the simulation time, which influences how quickly the complexity of the environment is changed due to a change in performance. The effect of changing t can be seen in Figure 4.5 (left), where the complexity of four different Rolling Sphere progressions is plotted, using the performance function in the Figure in the centre. This is the same performance function used for the Friction-based progression (Figure 4.3)

4.4 Progression on multiple dimensions

Up until this point, progression functions considered complexity a vector where all components were equal; however, as mentioned previously in Section 3.4 when discussing mapping functions, complexity can be independently specified over multiple dimensions.

Progressing multiple dimensions simultaneously poses several challenges and design choices in order to deal with multi-dimensionality. Firstly while our definition of mapping function assumes a certain degree of independence between dimensions, orthogonality cannot be assumed. Take as an example the previously mentioned problem of teaching an agent to drive a self-driving car with two dimensions of complexity: delay between actions and slippery road conditions. Adding one-tenth of a second of reaction time in good road conditions might not change the complexity of the environment, whereas adding the same amount of reaction time in slippery conditions might significantly increase the complexity of the environment. These inter-dependencies should not be addressed by over-engineering the mapping function but by the progression function specifying the appropriate complexity vector. Previously our approach consisted in progressing each dimension simultaneously, therefore this was not a concern. However, the approach to progress on multiple dimensions would need to consider this possibility and react to a sharp change in complexity over a relatively small change in the complexity vector. This motivates our choice of the Rolling Sphere progression to progress over multiple dimensions. As the progression only happens based on the last difference in performance and not over a larger interval, the Rolling Sphere progression will be more responsive to interactions between different dimensions of complexity, especially wherever discrete parameters are present.

One necessary design choice is how to deal with trying to calculate the speed of the rolling sphere in each dimension. Three options were considered: progressing all dimensions simultaneously, progressing one dimension at a time, and calculating the gradient in each dimension and progressing afterwards. The first option would be the most attractive, as it is the only case where all dimensions are progressed at all times. However, in order for this option to be viable, a method to estimate each dimension's accountability in the difference in performance needs to exist. As this method also needs to quickly adjust to inter-dimensional dependencies and be accurate even over several dimensions, the research efforts were focused on other methods instead. Nevertheless, the potential benefits of this technique make it an

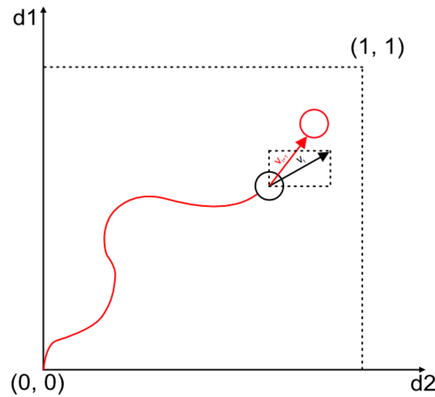

Figure 4.6: Rolling Sphere progression on multiple dimensions. A line search is performed in each dimension, as a result of which the sphere's velocity is updated. Finally the new velocity is used to determine the new position of the sphere.

exciting avenue for future work in this area. Progressing one dimension at a time would be a valid method if dimensions could be assumed to be orthogonal to each other; however, this is rarely the case, and it could result in sharp changes in the environment's complexity whenever progressing a large number of dimensions. The last option that was explored was calculating a partial gradient in each dimension of complexity and then progressing all dimensions simultaneously. This was found to be the most stable option, as it can address very complex dependencies between dimensions while still having a reliable measure of how modifying each dimension's complexity influences the overall complexity of the environment.

The algorithm used for progression in multiple dimensions starts with changing the simulation time of the Rolling Sphere progression (t) based on the number of dimensions to consider. The new value of t is selected such that for n dimensions, the new simulation time t' will allow for a maximum speed of the sphere in one direction n times that of t. This is necessary as simply multiplying t by the number of dimensions would not yield the required result (see Figure 4.5, left). As the progression happens over multiple dimensions, the speed and position are expressed as vectors rather than single numbers. This is also valid for the tracking of the extra speed introduced in the system (mentioned in Section 4.3). The algorithm starts with a line search in each direction to estimate a partial gradient of the performance function. This line search is executed by calculating the sphere's position in that dimension after the simulation time. The reason behind using the current speed for the line search rather than a constant value such as the maximum speed is that this

method acts as a "lookahead" step for each dimension to address inter-dimensional dependencies before they arise. Based on the performance experienced by the agent, the sphere's velocity in that direction is modified, the position of the sphere is reset, and the next dimension is considered. After all the dimensions have been evaluated and the speed in each dimension has been changed due to the progression, the sphere progresses to its new position, and the loop is restarted.

4.5 Experimental Setting

This section describes the reasoning behind the choice of baselines, and the implementation details for each baseline. As the evaluation domains, learning algorithm and experimental method are the same as in Chapter 3, refer to Section 3.8 for details regarding these.

4.5.1 Baseline choice

While our framework poses no restrictions on the MDPs generated by the mapping function, it also has some properties typical of Curriculum Learning algorithms with restrictions, such as a continuously changing task. As such, in order to validate our approach, our experimental section compares against one state-of-the-art algorithm from the no-restriction category (Section 2.3.2.3), and one that generates a curriculum while imposing restrictions on the source tasks (Section 2.3.2.2).

From the no-restriction class of Curriculum Learning algorithms, HTS-CR (Foglino et al., 2019a) was chosen as it guarantees to converge to the optimal curriculum for a given set of intermediate tasks; this removes the need to compare against other sequencing algorithms using a pre-defined set of intermediate tasks. One limitation of HTS-CR, and any other task sequencing algorithm, is that it cannot use the complete set of intermediate tasks used in our approach. This is because the first step of HTS-CR is to evaluate all curricula of length 2 to determine which tasks should be learnt first (heads) and which should be learnt at the end of a curriculum (tails). Assuming n is the number of source tasks to order, the number of curricula to be evaluated in this stage is:

$$\frac{n!}{(n-2)!} \tag{4.11}$$

Considering that the number of source tasks used by our approach is often infinite, and when it is not, it is composed of many tasks, using the same set of source tasks

would result in an evaluation only containing Curricula of length 2. To provide a fair comparison, the set of intermediate tasks was selected from a subset of the tasks used by our approach. More specifically, the set of source tasks considered by HTS-CR is $< \Phi_D(0), \Phi_D(0.25), \Phi_D(0.5), \Phi_D(0.75) >$, with the target task being $\Phi_D(1)$. This comparison helps to ascertain whether our approach of using a continuously changing intermediate task can outperform what is achievable with a limited set of tasks.

The second baseline chosen was Reverse Curriculum Generation (RCG) (Florensa et al., 2017), as the algorithm requires the same limited access to the environment and agent as our approach, and the pace of the curriculum created by this algorithm is akin to our progression functions. Furthermore, as RCG creates a curriculum by only modifying the agent's starting state, in the domains where RCG is applicable our algorithm's mapping function only modifies the agent's starting state in order to provide a direct comparison.

As a random curriculum was shown to be directly outperformed by at least one of our chosen baselines in their respective paper, we deemed this comparison unnecessary.

4.5.2 Implementation details

Reverse Curriculum Generation The version of the Reverse Curriculum Generation used as a baseline for our approach generates a curriculum via a random sampling of actions from starting states solvable by the agent but that have not been yet mastered. This algorithm also includes a replay buffer that considers the agent's performance over the course of the training to propose starting states to be replayed. Our implementation of this baseline is based on the author's original code.

As this algorithm generates a curriculum by modifying the starting state and assumes that given two states a and b in a certain MDP, you can get from a to b using a uniform sampling of random actions, it could not be applied to two of our testing domains: HFO and Predator Prey. They both contain an AI that plays against the agent with a specific policy that defends the goal (HFO) or chases the agent (Predator Prey). This violates the assumption mentioned above: in HFO, let state a be a normal starting state for the domain, where the agents are initialised next to the halfway line, and the defense team is initialised next to the goal. Let state b be the state where the agents are next to the goal, and the defenders are next to the halfway line. The agents can take no sequence of actions that will result in

state b occurring when the environment is initialised in state a, as whilst the agents might get close to the goal, the defense team would never abandon their task and go next to the halfway line. Likewise, in the Predator Prey domain, a similar issue arises where the predator would keep following the agent, limiting the portion of the state space that can be reached from a certain starting state.

HTS-CR Using HTS-CR (Foglino et al., 2019a) as one of our baselines, we used transfer between tasks; the transfer method consisted of directly transferring the neural network from one task to the next, using it as a better initialisation on the next task. One limitation of HTS-CR is that it cannot use the full set of intermediate tasks used in our algorithm, like any other task sequencing algorithm. In fact, this set is often infinite and, if used in its entirety, would result in an evaluation only containing Curricula of length 2. As previously mentioned, to provide a fair comparison, the set of intermediate tasks was selected from a subset of the tasks used by our approach. More specifically, the set of source tasks considered by HTS-CR is $< \Phi_D(0), \Phi_D(0.25), \Phi_D(0.5), \Phi_D(0.75) >$, with the target task being $\Phi_D(1)$. Four tasks were included to ensure HTS-CR had enough time to complete the head and tail selection process and evaluate some longer curricula within the time limits. HTS-CR was modified to optimise the performance on the final task rather than the cumulative return; this does not violate the guarantee of optimality and is the more appropriate metric to be optimised for this comparison. HTS-CR was trained 10 times longer compared to the other algorithms in all of our test domains. In the plots provided in the results section, the final performance of HTS-CR is reported as a dashed line. As at every iteration, HTS-CR starts training on the final task only after having trained on the full curriculum; when plotting the performance of this algorithm, its curve will not start until later in the training. Our implementation of this baseline is a direct translation of the author's code from Java to Python.

Hardware specifications The experiments were conducted using Python on a computer with a 16 core AMD Ryzen 9 3950X CPU, 32 GB of RAM and an NVIDIA GeForce RTX 2080 Ti GPU.

4.6 Results

This section presents the results on the six testing domains. First we perform an ablation study to establish the properties of several components of the framework:

uniform sampling of the Friction-based progression, parallelism among learners, and correctness of the mapping function. We aim to verify experimentally that uniform sampling is beneficial to the agent's performance; that parallel learning does not result in a (possibly faster) convergence to a worse behaviour than with single-process learning; and that adaptive progression functions are robust to errors in the mapping function. Finally, with the last experiment we evaluate our framework by comparing the Exponential, Friction-based and Rolling Sphere progression functions to other state-of-the-art Curriculum Learning algorithms.

4.6.1 Friction-based progression

This section seeks to verify that uniform sampling in the Friction-based progression has a positive effect on the agent's performance.

In order to verify whether the uniform sampling introduced in Equation 4.1 is advantageous, we compared this formulation of the Friction-based progression, referred to as *uniform* from here onward, to the *monotonic* formulation introduced in Equation 4.5. We also developed an alternative formulation aimed at modelling the more simple case where there is a direct correlation between the speed of the object and the complexity of the environment, specified by the following equation:

$$\Pi_{fu}(t, m, s_{t-1}, p_{t-i}, p_t, i) = \vec{1} \cdot (1 - s_t) \tag{4.12}$$

Given the direct relationship between the speed of the object and performance of the agent, this formulation will be referred to as *speed* from here onward.

Experimental results We compared the three formulations described above on all our six testing domains to find the best performing one and explore each formulation's properties. The plots for these experiments can be found in Figure A.1, with Table 4.1 reporting the highest performance achieved on each domain by each formulation.

Variant	GW Maze	Point Maze	Dir. P. M.	Ant Maze	2v2 HFO	Pred. Prey
Speed	**89.4 ± 6.0**	**100 ± 0.0**	**98.5 ± 0.6**	**95.0 ± 1.5**	65.7 ± 6.1	216.8 ± 13.3
Sampling	**94.4 ± 3.9**	**99.7 ± 0.4**	**97.8 ± 1.2**	**95.1 ± 1.4**	**75.8 ± 1.4**	**249.8 ± 8.5**
Monotonic	**94.0 ± 3.6**	99.1 ± 0.7	**98.2 ± 0.7**	**94.6 ± 1.0**	68.0 ± 5.0	**238.2 ± 15.6**
Metric	Treasures out of 100	Goals reached out of 100	Goals reached out of 100	Goals reached out of 100	Goals scored out of 100	Survival time in actions

Table 4.1: Best performance on each domain. All the methods within a 95% confidence interval from the best performing method are in bold.

Our experimental results show that in the Point Mass Maze environment, there is a clear benefit when using the *speed* formulation, as it results in a significant increase in performance during the initial part of the training and also results in a much quicker convergence to the optimum. In this domain, the *sampling* and *monotonic* formulations perform similarly, with the *sampling* formulation resulting in slightly higher performance at the start of the training and converging within the confidence interval of the best performing formulation.

In both the Directional Point Maze domain and the Ant Maze domain, all three formulations perform equally both throughout the training and as far as the best performance achieved is concerned.

When analysing the results in the Grid World Maze, HFO and Predator Prey domains, the benefits of the *sampling* formulation are highlighted over the *speed* formulation. Firstly in the Grid World Maze domain, the *sampling* and *monotonic* formulations perform equally throughout the training, whereas the *speed* formulation has a lower probability of solving the maze in the initial part of the training. Moreover, the *speed* formulation converges to a considerably lower average value, although still within a 95% confidence interval to the other two formulations. Moreover, in the HFO domain, the *speed* formulation results in a lower initial performance compared to the other two. This is significant as the discrete parameter included in this domain, whether the offending team starts with the ball, is changed towards the start of the progression. In the Predator Prey domain, on the other hand, the performance of the *speed* formulation drops significantly in the later stages of the training, unlike the other two formulations. This is due to instability in the agent's performance as the progression gets to the target task, resulting in the progression quickly lowering the complexity of the environment. Not being able to train on the hardest complexity directly, in turn, hurts the testing performance of the agent on the target task, which is particularly evident when looking at Figure A.1. This is mitigated by the *monotonic* formulation by having the agent train on the target task directly whenever the progression finishes; however, the experimental results suggest that the *sampling* technique is the most effective. This pattern is also present in the HFO domain, where the *sampling* formulation performs significantly better than the *monotonic* progression.

Based on our experiments, we hypothesise that the definition of the Friction-based progression in Equation 4.12 results in the algorithm having a weakness in domains with discrete parameters that can significantly increase its difficulty. In fact, if the agent's performance drops significantly in the harder environment, the

progression would immediately decrease the complexity, and the agent would not have enough time to learn this new and more complex environment. This could, in turn, result in the progression being halted. This is supported by the fact that whenever an environment has a discrete parameter, the performance of the *speed* formulation was inferior compared to when using a uniform sampling.

Overall our experimental results highlight the benefits of the higher stability provided by the *sampling* formulation, which was always either the best performing one or within the confidence interval of the best performing formulation. The *speed* formulation seems to be best suited for less complex domains and seems to be less effective whenever there are discrete parameters in the domain. Finally, the *monotonic* formulation seems to be more stable than the *speed* formulation and would be useful if a monotonic increase in complexity was needed. As the *sampling* formulation was found to be the best performing one, from this point onward, whenever we refer to the Friction-based progression, the *sampling* formulation is implied.

4.6.2 Effects of Multiprocessing

In this experiment, we seek to verify whether the use of multiprocessing hinders our approach's performance. Further to this, we also want to verify if multiprocessing has any added benefits aside from speeding up the learning process.

In order to verify the hypothesis stated above, we tested the Exponential, Friction-based and Rolling Sphere progression on our testing domains. All progression functions were tested with and without multiprocessing in order to be able to compare their performance and draw some more general conclusions on the effects of multiprocessing on our method. The multiple processes were set up as independent workers, each with an independent curriculum, as described in Section 3.5. After a certain number of time steps, the experience gathered by all the workers was combined in a single update for the PPO algorithm. The number of workers used was 4 for the Grid World Maze domain, 8 for all MuJoCo navigation domains, and 16 for the Predator-prey domain. Because of the instability of the code used in the HFO domain, it was not practical to use multiprocessing in this case; thus, we will only be performing this test on the remaining five domains.

Experimental results Our experimental results reported in Table 4.2 (with the full plots available in Section A.2) show a clear increase in performance over the first half of the training when using multiprocessing for both the Exponential and

Friction-based progression functions on the Grid World Maze domain. This increase in performance is sustained throughout the training for the Friction-based progression; however, the best average performance achieved by the Exponential progression is achieved without multiprocessing. It is worth mentioning that this performance's confidence interval intersects the confidence interval relative to the method with multiprocessing. On the other hand, when using the Rolling Sphere progression, no multiprocessing results in a higher initial performance but lower asymptotic performance (but still within the confidence interval).

Variant	GW Maze	Point Maze	Dir. P. M.	Ant Maze	Pred. Prey
F. B. P. multi	**94.4 ± 3.9**	**99.7 ± 0.4**	**97.8 ± 1.2**	**95.0 ± 1.4**	**249.8 ± 8.5**
F. B. P. single	74.6 ± 11.5	**99.4 ± 0.5**	**97.6 ± 2.0**	**95.0 ± 1.4**	**239.6 ± 10.2**
R. S. P. multi	**93.5 ± 5.6**	**100.0 ± 0.0**	**98.8 ± 0.5**	**94.2 ± 1.1**	**250.6 ± 7.5**
R. S. P. single	**90.6 ± 6.2**	**100.0 ± 0.0**	**99.1 ± 0.3**	91.2 ± 1.2	234.2 ± 7.7
Exponential multi	**77.5 ± 10.1**	**97.5 ± 1.7**	**79.3 ± 2.9**	**92.0 ± 2.0**	**239.3 ± 7.9**
Exponential single	**88.9 ± 7.8**	75.3 ± 1.0	51.6 ± 1.5	**91.0 ± 2.5**	**224.2 ± 8.3**
Metric	Treasures out of 100	Goals reached out of 100	Goals reached out of 100	Goals reached out of 100	Survival time in actions

Table 4.2: Best performance on each domain with and without multiprocessing, divided by method. If the algorithm with and without multiprocessing perform within a 95% confidence interval, both performances are in bold, otherwise the highest of the two is in bold.

In the Point Maze and Directional Point Maze domains, using multiprocessing in conjunction with the Friction-based progression results in increased average performance, although within a confidence interval of the Friction-based progression without multiprocessing. On the other hand, in these domains multiprocessing resulted in a significant increase in the performance of the Exponential progression throughout the training. The Rolling Sphere progression tends to have a higher initial performance without multiprocessing, but tends to have a higher final performance with multiprocessing in these domains.

There is no measurable difference between the Friction-based progression with or without multiprocessing on the Ant maze domain. On the other hand, using multiprocessing with the Exponential progression results in an increase in performance over the first half of the training but results in convergence to a similar value compared to the method without multiprocessing. The Rolling Sphere progression's behaviour on this domain is the opposite: initially there is no difference in performance, but when training without multiprocessing the final performance is considerably lower.

Finally, both the Exponential and Friction-based progression converge to a higher average value using multiprocessing in the Predator Prey domain but converge just within the confidence interval of the same method without multiprocessing. The Friction-based progression without multiprocessing results in a higher average performance in the first half of the training but has a lower average performance for the rest of the training. On the other hand, the Exponential progression without multiprocessing results in a lower performance from the first third of the training onward, although usually within the confidence interval of the Exponential progression with multiprocessing. One final detail that should be mentioned is that both the Friction-based progression and the Exponential progression without multiprocessing experienced a significant drop in performance at some point in the training, which is not present in their counterpart when using multiprocessing. This coincides with the time where the progression is set to end with the Exponential progression and is likely also linked to a transition to the target task in the Friction-based progression. This is a crucial time of the progression, and in this specific domain, this drop highlights that more stability is needed. In the case of multiprocessing, this stability is provided by having different processes end the progression at different times (Friction-based) or progressing to more complex environments with different schedules. The Rolling Sphere progression without multiprocessing results in a significantly higher performance over the first third of the training, however its convergence is outside of the confidence interval of the version with multiprocessing, indicating a clear benefit of the technique in this domain.

In order to quantify the speedup experienced when using multiple parallel workers, we timed the execution time of each run and the results can be seen in table 4.3. It is worth noting that the execution time might not be a good representation of the difference in speed of each method. This is motivated by the fact that run timings are influenced by other experiments and programs running on the same machine. This could be mitigated by running experiments one at a time, however it would also result in a considerable amount of time being needed, such as more than 16 continuous days of experiments for each method in the single process Predator Prey domain. This domain is also the one that best highlights the low reliability of execution time as a metric: the Friction-based progression averages an execution time of just under 40 hours, whereas the Rolling Sphere and Exponential Progression seem to be twice as fast. This however is not a pattern found anywhere else, and given the high memory requirements of the predator-prey domain due to using a convolutional neural network, our hypothesis is that the machine was running several of

these experiments in parallel[1], and was forced to use its swap as well as its RAM during the training, greatly reducing its performance. Adding to the uncertainty of the execution time metric is the use of Precision Boost Overdrive on the AMD Ryzen 9 3950x processor; this technology boosts the processor's clock speed based on its load and temperature, resulting in an increased dependency of execution time on external factors. Taking the unreliability of the metric into account, we can still conclude that there is a significant time advantage in using multiple parallel workers over only a single one.

Variant	GW Maze	Point Maze	Dir. P. M.	Ant Maze	Pred. Prey
F. B. P. multi	2:30	9:01	15:31	2:15:26	2:19:51
F. B. P. single	5:51	35:04	1:03:26	10:12:33	39:36:07
R. S. P. multi	2:30	9:18	16:51	2:10:29	4:06:48
R. S. P. single	7:09	32:09	1:03:02	8:25:25	20:00:43
Exponential multi	2:29	9:14	17:19	3:39:03	3:45:36
Exponential single	6:52	33:48	1:02:22	11:09:39	19:16:06

Table 4.3: Average execution time on each domain with and without multiprocessing, divided by method.

Our results suggest that as well as providing a lower execution time, utilising multiprocessing does not hurt the performance on the agent, and it can, at times, result in an increase in performance. Concerning the Friction-based progression, we did not observe an instance in our testing where it was detrimental to use multiprocessing. Whenever using the Rolling Sphere progression, the performance was always higher in the initial stages without multiprocessing, but the performance later in the training was often noticeably higher when using multiprocessing. On the other hand, when using an Exponential progression, the version without multiprocessing resulted in a higher best performance in the grid world maze domain, albeit still within the confidence interval of the version with Multiprocessing. This is offset by the Exponential progression benefiting from the use of Multiprocessing in our other four testing domains.

4.6.3 Robustness of adaptive progression functions

As progression functions rely on mapping functions in order to generate a curriculum, we consider whether the Friction-Based and Rolling Sphere progression are

[1] Our code for running experiments has a list of experiments to run, and it runs an experiment whenever enough cpu cores are freed by other experiments. This results in an unpredictability on which experiment is ran alongside other experiments.

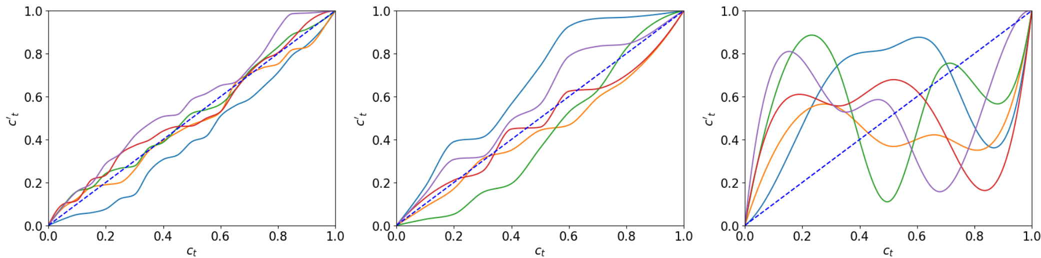

Figure 4.7: Examples of the three different types of noise: Local (left), Global (center) and Random (right). The blue dashed line represents $\vec{c'_t} = \vec{c_t}$, i.e. no noise.

resilient to a badly formulated mapping function. This section is specific to adaptive progression functions as they are generated online and therefore can adapt to noise in the progression. Since the Exponential progression is determined before execution, in the case of a badly formulated mapping, its parameters would need to be modified on a case by case basis.

In order to assess the robustness of our method, we modelled a badly formulated mapping function as noise in the creation of a curriculum. This noise takes the form of a function $\mathcal{N} : [0, 1]^n \to [0, 1]^n$ that maps the complexity vector $\vec{c_t}$ to its noisy counterpart $\vec{c'_t}$. This value will then be used in the generation of the MDPs to be added to the curriculum, resulting in the noisy curriculum C' being defined by the following two equations:

$$M'_t = \Phi(\mathcal{N}(\vec{c_t})) \tag{4.13}$$

$$C' = \langle M'_0, ..., M'_i \rangle \tag{4.14}$$

We identified two possible cases of mistakenly creating a mapping function. The first case is when the mapping function correctly identifies the domain knowledge that should be represented but encodes it incorrectly. For example, in a navigation domain, one such mapping function would correctly assume that a state closer to the goal would be at least as easy to solve as a state further away from the goal; however, the difference in complexity of these two states would be random. On the other hand, the second case is a random mapping function, modelling whenever even the domain knowledge encoded in the mapping function is incorrect. In a navigation domain, this would result in a state closer to the goal being potentially considered more complex than a state further from the goal.

The first case results in the noise function being monotonic, implying that given two values a and b, if $a > b$ then $\mathcal{N}(a) \geq \mathcal{N}(b)$. In this evaluation we defined two

possible noise functions belonging to this class: *local* and *global*. The first type of noise is useful to assess how robust the Friction Based progression is to a sharp but localised perturbation, whereas the second type of noise tests the algorithm's ability to react to smoother but greater noise. These two types of noise can be seen in Figure 4.7. These two noise functions still need to respect the constraints below with respect to each component of a complexity vector.

$$a > b \rightarrow \mathcal{N}(a) \geq \mathcal{N}(b) \quad \forall\, a, b \in [0, 1] \tag{4.15}$$

$$\mathcal{N}(0) = 0 \tag{4.16}$$

$$\mathcal{N}(1) = 1 \tag{4.17}$$

with the constraints in eq. 4.16 and 4.17 being necessary to ensure that all complexity vectors are still reachable. The algorithm used to generate $\mathcal{N}$ starts with the creation of a list $L = \{l_0, ..., l_n\}$ of n random values sampled from a truncated normal distribution bounded between 0 and 1 with a mean of 0.5. This list is then converted in a series of x and y coordinates, where for a given value of the list l_i the corresponding coordinates are $x = i; y = \sum_{k=0}^{i} l_k$. The coordinates are then normalised such that $(x_0, y_0) = (0, 0)$ and $(x_n, y_n) = (1, 1)$. To convert this series of points to a function and satisfy the constraints mentioned above, we use monotone preserving cubic interpolation (Hyman, 1983), and normalise the interpolated function with respect to y. Our noise generation process has two parameters: the number of points used and the standard deviation of the normal distribution. By setting these parameters accordingly, we generated the two types of noise previously introduced: *short* and *long* (Figure 4.7).

In order to model noise that does not respect the domain knowledge encoded in the mapping function, on the other hand, the only two constraints to be respected were those in Equations 4.16 and 4.17. This type of noise, referred to as *random*, is generated by defining a set of coordinates that were equally spaced on the x axis, and whose y coordinate was a random value between 0 and 1. In order to generate $\mathcal{N}$, we performed standard cubic interpolation and normalised the function between 0 and 1.

In order to isolate the effect of the noise on the progression, in our experimental evaluation, the set of parameters used for both progression functions was not changed.

Experimental results Our experimental result shows that in the Grid world Maze environment, the *local* and *global* noise have a limited influence on the performance of the agent for both progression functions, as their best performance falls within the confidence interval of the noise-less method. On the other hand, while the *random* noise doe not affect the Rolling Sphere progression, it negatively affects the Friction-based progression by resulting in lower performance in the initial part of the training and also a lower best performance. It should be noted that the Friction-based progression's performance was steadily increasing, suggesting that if given more time, this form of noise would converge to a similar value compared to the noise-less method, albeit slower. Moreover, in this domain, the performance of the Friction-based progression with *random* noise is still superior to RCG.

In the MuJoCo maze environments, applying either *local* or *global* noise does not significantly affect the agent's final performance, as all the confidence intervals for different types of noise and progressions overlap. Figure A.3 in Appendix A, in the Point Mass Maze and Directional Point Maze, shows how applying short noise to the Friction-based progression results in a marginal increase in the average performance in the first half of training. We hypothesise this occurs because the local perturbations caused by the short noise cause the progression to quickly increase the complexity of the environment, which could be beneficial in less complex environments, but detrimental in more complex ones. When applying *random* noise to the Friction-based progression in the same domains, a slight drop in performance can be observed in the Point Maze domain; however, no significant drop in performance can be observed in the directional Point Maze or Ant Maze domains. While no significant difference was observed for the Rolling Sphere progression in the Point Maze or Ant Maze domain for any type of noise, the *global* and *random* noise result in a lower performance (albeit with overlapping confidence intervals for the best performance) in the second half of the training in the Directional Point Maze domain (figure A.4).

In the HFO environment, the best performance achieved by the *local* noise falls within the confidence interval of the best performance for the Friction-based progression. However, the deviation between the runs is increased. On the other hand, applying *global* noise to the Friction-based progression results in a drop of 8.8% in the final average probability to score and a noticeable increase in the standard deviation between different runs. When looking at Figure A.3, the average scoring probability is always higher when no noise is present, although the *local* noise is always within the confidence interval. Applying *random* noise has an extremely

detrimental effect on this specific domain, resulting in a severe drop in performance. It should be noted that this is the only domain where this phenomenon can be observed, so it is hypothesized that this is due to this being the only domain that does not include multiprocessing. The Rolling Sphere progression on the other hand, has a more clear separation between different kinds of noise, as can be seen in Figure A.4. While *local* noise falls within the confidence interval of the best performance for the Rolling Sphere progression, the plot shows the detrimental effects of the noise on the progression. The *global* noise's performance is just below the confidence interval for the local noise throughout the training whereas the *random* noise results in a sharp decrease in performance for the Rolling Sphere progression.

Finally, in the Predator Prey domain, the performance seems unaffected by either *local* and *global* of noise until 2/3 of the training time for the Friction-based progression. Both *local* and *global* noise cause an increased confidence interval and a loss of average performance after this point. This effect is more pronounced when adding *global* noise, although the best performance achieved by both types of noise falls out of the confidence interval relative to the noise-less method. Applying *random* noise in this domain with the Friction-based progression results in a slower but steadier increase in performance, with its best value falling within the confidence interval of the other two types of noise. The Rolling Sphere progression on the other hand has a similar behaviour for *local* and *global* noise in this domain, both being within the same confidence interval for the whole training. The *random* noise on the other hand has a more noticeable effect on this domain, resulting in a lower performance throughout the training, although with the best performance achieved intersecting the confidence interval noiseless method's best performance.

Our results show that even without changing the parameters of our algorithms, the Friction-based and Rolling Sphere progressions are highly resilient to both the *local* and *global* types of noise. Moreover, the *local* noise is found to have a lesser impact on the training compared to the *global* noise. This is an expected result as whilst local perturbations in the complexity of the environment can be addressed by our algorithms, more pronounced perturbations result in the progression functions with a higher interval to not keeping up with the agent's ability to solve the environment. On the other hand, the Friction-based progression is more vulnerable to *random* noise, which is also expected, as this noise results in little correlation between how difficult an MDP is and its complexity. While the Rolling Sphere progression has shown resilience on this type of noise, when progressing multiple dimensions simultaneously its resilience is significantly impacted. This is likely due to

the partial gradient calculation, which results in noisy estimations and a less precise progression.

Variant	Alg.	GW Maze	Point Maze	Dir. P. M.	Ant Maze	2v2 HFO	Pred. Prey
Standard	FBP	**94.4 ± 3.9**	**99.7 ± 0.4**	**97.8 ± 1.2**	**95.1 ± 1.4**	**75.8 ± 1.4**	**249.8 ± 8.5**
	RSP	**93.5 ± 5.6**	**100.0 ± 0.0**	**98.8 ± 0.5**	**94.2 ± 1.1**	**76.7 ± 1.3**	**250.6 ± 7.5**
Short Noise	FBP	**92.5 ± 5.4**	**100 ± 0.0**	**98.6 ± 0.7**	**94.3 ± 1.3**	**75.2 ± 3.6**	225.7 ± 14.0
	RSP	**90.8 ± 5.7**	**100.0 ± 0.0**	**98.9 ± 0.6**	**95.3 ± 0.8**	**74.2 ± 3.3**	**242.5 ± 8.9**
Long Noise	FBP	**94.8 ± 4.3**	**99.8 ± 0.2**	**98.0 ± 1.0**	**94.6 ± 1.0**	67.0 ± 7.2	218.9 ± 14.1
	RSP	**94.9 ± 4.3**	**100.0 ± 0.0**	97.0 ± 1.3	**94.1 ± 1.9**	63.0 ± 8.0	**235.7 ± 10.2**
Random Noise	FBP	77.4 ± 10.8	97.9 ± 1.3	**99.2 ± 0.4**	**93.1 ± 1.4**	21.6 ± 9.6	181.0 ± 38.0
	RSP	**95.1 ± 4.5**	**99.4 ± 0.7**	**98.5 ± 0.7**	**94.0 ± 1.8**	31.4 ± 10.2	**222.5 ± 21.7**
Metric		Treasures out of 100	Goals reached out of 100	Goals reached out of 100	Goals reached out of 100	Goals scored out of 100	Survival time in actions

Table 4.4: Best performance on each domain. All the variants of each algorithm within a 95% confidence interval from the method without noise are in bold.

4.6.4 Comparison with the state of the art

This section seeks to verify whether the approaches derived from our framework can outperform state-of-the-art Curriculum Learning algorithms.

In this experimental evaluation, we compare the Friction-based progression, the Rolling Sphere progression and the Exponential progression to two state-of-the-art Curriculum Learning techniques on our testing domains. As the assumptions of Reverse Curriculum Generation are not met by two of our testing domains, as previously discussed, the comparison to this algorithm will take place on the remaining four. The results of this comparison are plotted in Figure,4.8 and the best performance of each algorithm on each domain is reported in Table 4.5.

Experimental results In the Grid World Maze domain, the Friction-based, Rolling Sphere progression and HTS-CR all converge to similar performance, whereas the best performance achieved by the Exponential progression and RCG falls within the confidence interval from each other. It is worth noting that the difference in the success rate of these two methods is about 10% in favour of the Exponential progression, although RCG did not have enough time to converge, suggesting that if given more time, its final performance would be higher.

In the Point Mass Maze domain, the Friction-based and Rolling Sphere progressions are the best performing methods, with the Exponential progression's best performance being just outside of their confidence interval. RCG is the third-best performing method, keeping up with the progression functions in the first third of the training but resulting in a lower best performance. Finally, HTS-CR is the worst

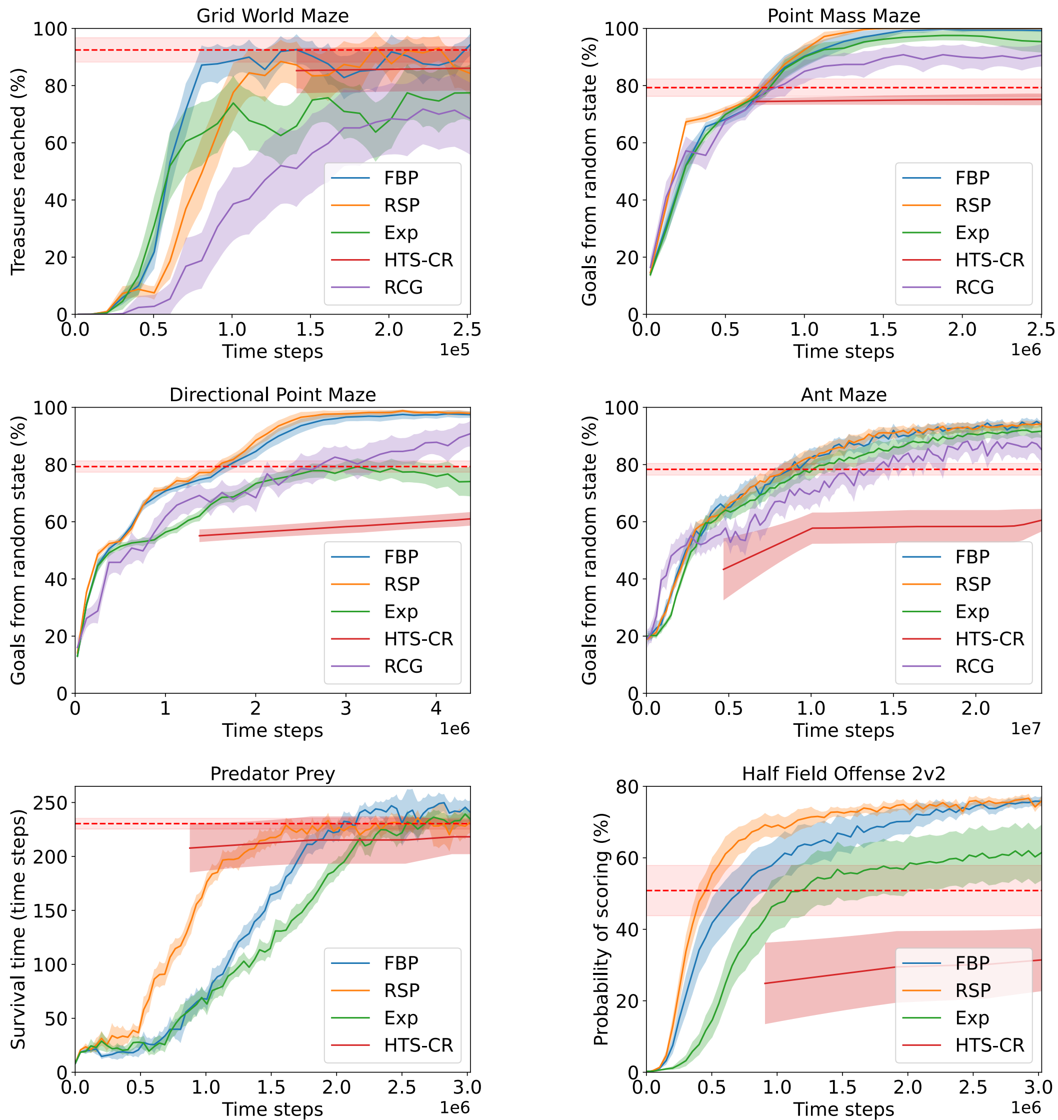

Figure 4.8: Performance of different progression functions, *Reverse Curriculum* and *HTS-CR* on six test domains, with the dashed line representing the final performance for *HTS-CR*. In these plots each time step represents an action made by the agent in the environment.

performing method in this environment, achieving an average success rate of 79.3% on the target task.

In the Directional Point Maze domain, our results show a clear advantage to using the Friction-based and Rolling Sphere progression over all other methods, resulting in a convergence close to the optimal policy already at three-quarters of the

training. The Rolling Sphere progression has a slightly better average performance at times, although always with overlapping confidence intervals. On the other hand, RCG produces a curriculum with a much higher deviation between different runs and a significantly slower increase in performance. The Exponential progression is clearly outperformed by RCG in this domain, as whilst in the first three-quarters of the training the two methods perform similarly, RCG's performance keeps improving whereas the Exponential progression's performance falls off slightly. As the Exponential progression does not take into account the agent's performance, it will progress to a more complex environment regardless of whether the agent can solve the current one. As a lot of the environments where Curriculum Learning is useful have a sparse reward function, this could potentially result in the agent not experiencing any reward for several epochs, resulting in the deterioration of the agent's policy. Out of the six testing domains, this is the only one where this phenomenon can be clearly identified by looking at the agent's performance over time. It should also be noted that the best performance achieved by the Exponential progression in this domain is equal to the performance achieved by HTS-CR.

In the Ant maze domain, the Friction-based and Rolling Sphere progressions were the best performing methods, with the Exponential progression being within their confidence interval for most of the training. Reverse Curriculum Generation resulted in a lower performance throughout the training and an increased deviation between runs. Finally, HTS-CR's final performance was the lowest of the algorithms tested on this domain.

As mentioned in Section 4.5.2, RCG was not applicable in both the HFO and Predator Prey domains.

The results in the HFO domain are generally consistent with our other testing domains, where the Friction-based and Rolling Sphere progressions are the best performing algorithms in this domain (with the Rolling Sphere's mean performance being slightly higher), followed by the Exponential progression and finally by HTS-CR. In this domain, the final performance of HTS-CR was within the confidence interval of the best performance of the Exponential progression, although the Exponential progression resulted in a 12% increase in the average probability to score.

Finally, in the Predator Prey domain the advantages of being able to progress in multiple dimensions using the Rolling Sphere progression are highlighted. This method in fact converges to its final performance by half of the training, clearly outperforming the Friction-based progression in the time-to-threshold metric, but converging to a similar performance. The Friction-based progression was the second

best performing method, although the Exponential progression's highest survival time was within its confidence interval. On the other hand, HTS-CR's final survival time was within the confidence interval of the Exponential progression but not within the confidence interval of the Friction-based progression.

Overall our experimental results show that the Rolling Sphere and Friction-based progressions were consistently the methods with the highest average performance; moreover, the Exponential progression's performance was either on par with other state-of-the-art Curriculum Learning algorithms or greater. This clearly highlights the benefits of using any of the three progression function and also shows the potential of the framework defined in this paper as more progression functions are created.

Algorithm	GW Maze	Point Maze	Dir. P. M.	Ant Maze	2v2 HFO	Pred. Prey
F. B. Pr.	**94.4 ± 3.9**	**99.7 ± 0.4**	**97.8 ± 1.2**	**95.1 ± 1.4**	**75.8 ± 1.4**	**249.8 ± 8.5**
R. S. Pr.	**93.5 ± 5.6**	**100.0 ± 0.0**	**98.8 ± 0.5**	**94.2 ± 1.1**	**76.7 ± 1.3**	**250.6 ± 7.5**
Exp. Pr.	77.5 ± 10.1	97.5 ± 1.7	79.3 ± 2.9	**92.0 ± 2.0**	62.0 ± 7.9	**239.3 ± 7.9**
HTS-CR	**92.4 ± 4.3**	79.3 ± 3.0	79.3 ± 2.0	78.3 ± 2.0	50.8 ± 7.0	230.4 ± 5.2
Rev. Curr.	66.8 ± 13.3	90.8 ± 3.7	90.8 ± 4.5	88.2 ± 4.0	n/a	n/a
Metric	Treasures out of 100	Goals reached out of 100	Goals reached out of 100	Goals reached out of 100	Goals scored out of 100	Survival time in actions

Table 4.5: Best performance on each domain. All the methods within a 95% confidence interval from the best performing method are in bold.

Chapter 5

Automatic mapping functions

The algorithms presented in this thesis so far focus on automating the creation of a curriculum assuming some domain knowledge is available in order to build a mapping function. However, this might not always be the case, as some environments might contain parameters that do not have a direct and obvious correlation with the complexity of the environment. Moreover, it should be noted that up until this point mapping functions were assumed to be static: complexity was in fact defined as being related to the learning time of each MDP. This definition does not take into account that complexity is not only dependent on the environment itself, but also on the agent's internal parameters. In order to account for this, in this chapter we change the intuition behind complexity from "how complex a task would be for a randomly initialised agent" to "how complex a task is for the current agent at this point in time". This requires a degree of automation in the generation of mapping functions, as it would be unrealistic for a domain expert to foresee the agent's ability at different points during its training. However, automating the generation of the mapping function also presents the opportunity to better tailor a curriculum to each individual agent by leveraging the information on its ability. As up until now our methods for generating a curriculum have always been learning algorithm and function approximator agnostic, our aim is to preserve these properties when automating the generation of mapping functions. Therefore, we model the problem of selecting a task of appropriate complexity as a modified version of the Dynamic Multi-Armed Bandit problem (Whittle, 1988). Each task is modelled as an arm and the aim of the problem is to pull the arm (select the task) as close to the desired complexity as possible.

This chapter starts by specifying the problem it is trying to solve, to then in-

troduce the literature that is relevant to our intended approach. Following this, a new variation of the Dynamic Multi-Armed Bandit problem is introduced, and three algorithms from the literature are adapted to solve this problem. We then discuss how to generate mapping functions with algorithms that solve this new formulation. Following which we perform an initial evaluation in order to choose which of the three algorithms to fine tune for the generation of the mapping function. Finally an evaluation is conducted against a manually generated mapping function.

5.1 Problem Specification

In Chapter 3 we define a mapping function as a function $\Phi_D : [0,1]^n \rightarrow M$ that maps a complexity vector between 0 and 1 to an MDP of the corresponding complexity. As this section will look to generate mapping functions automatically and to condensate the complexity of an MDP over a single dimension, we define the complexity value c_t as being the first and only component of a one dimensional complexity vector $\vec{c_t}$. As such, given the set of potential intermediate tasks M, the problem of automatically generating a mapping function entails selecting the task in M whose complexity is closest to the desired value. Posing the problem in these terms requires a new definition of complexity: as mentioned in the introduction to this chapter our previous definition did not take into account the agent's individual ability. We therefore define complexity as a function of the agent's performance on a given MDP. In particular, given an agent with inner state θ and a performance function P, let $p_\theta^- = \min_{m \in M} \bar{P}(\theta, m)$ be the lowest average performance the agent (given its state) achieves on an MDP in the set M, and let $p_\theta^+ = \max_{m \in M} \bar{P}(\theta, m)$ be the highest. The complexity of an MDP can then be defined as:

$$Comp(\theta, m) = 1 - \frac{\bar{P}(\theta, m) - p_\theta^-}{p_\theta^+ - p_\theta^-} \tag{5.1}$$

Assuming that the set of MDPs M stays stationary throughout the training, the problem of generating a mapping function is specified by a sequence of agent states $[\theta_0, ..., \theta_T]$ and a sequence of complexity values $[c_0, ..., c_T]$. Given both sequences, a mapping function generates a sequence $[m_0, ..., m_T]$ of MDPs in M whose error can be defined as:

$$Error = \sum_{t=0}^{T} (Comp(\theta_t, m_t) - c_t)^2 \tag{5.2}$$

Algorithm 2 Automatic mapping function

Input: Set of MDPs M, *DVB algorithm*, *RL algorithm*, progression function Π, performance function p

1: t := 1
2: m_t := random element of M
3: **while** learning **do**
4:     *RL algorithm* executes an episode on m_t
5:     Calculate performance p_t based on the observed states and rewards
6:     Update *DVB algorithm* based on $m_t \rightarrow p_t$
7:     $c_t := \Pi(t, p_t)$
8:     $\mathbb{V}_t := p_\theta^- + (p_\theta^+ - p_\theta^-) \cdot c_t$
9:     m_{t+1} := Result of *DVB algorithm* with target = $\mathbb{V}_t$
10:     t := t + 1
11: **end while**

It is the aim of the mapping function to minimise the error defined in the equation above by selecting the appropriate arm given the agent's state and required complexity.

While this definition might suggest that the agent's state needs to be known in order to determine the complexity of the environment, we assume that that is not the case. Our assumption is that the agent's performance can be observed, however it is up to the algorithm generating a mapping function to both estimate $\bar{P}(\theta, m)$ as well as p_θ^- and p_θ^+ if needed.

The way an algorithm that automates the generation of a mapping function interacts with our framework can be seen in Algorithm 2. In particular, the similarity between Algorithm 2 and the Contextual Multi-Armed Bandit problem (Lu et al., 2010) is what prompted us to use the Multi-Armed Bandit framework to address our problem. In order to further demonstrate this similarity and provide the necessary background information on the wider Multi-Armed Bandit framework, the next section will give a brief introduction to the area.

5.2 Background

This section will first introduce the Multi-Armed Bandit problem and some of the methods that are used in order to approach it. It then discusses the special case of the problem when each arm's payoff changes over time. While this case is more complex, it is also more widely applicable. Finally we will discuss the problem of Contextual Bandits, where the decision of which arm to pull is influenced by what

context is observed. Later in this chapter we will introduce our specific version of Contextual bandits, where the desired complexity value is modelled as a context

5.2.1 Multi-Armed Bandits

The Multi-Armed Bandits (MAB) problem was introduced by Robbins (1952), and it is often used to study the exploration and exploitation dilemma. The problem revolves around a slot machine (or bandit) with multiple arms $A = [a_0, ..., a_K]$, where each round an arm is pulled a reward is received, and no prior information on each arm's reward is available. Each arm has its own fixed probability distribution over rewards $D = [d_0, ..., d_K]$ (unknown to the player), and rewards are independent and identically distributed. The goal of the player is to maximise the reward gathered over all rounds.

The most simple approach to solving a MAB problem is to employ an ϵ greedy strategy: playing the arm with the highest mean reward with probability $1 - \epsilon$, and a random arm with probability $\epsilon \in (0, 1)$. While this strategy always has a degree of exploration, and is therefore guaranteed to eventually identify the arm with the highest average reward, it doesn't take into account the exploration disparity between arms.

This weakness is addressed by the UCB method (Auer et al., 2002a), with the intuition of estimating an upper confidence interval for each arm's expected reward. This is achieved by using the Chernoff-Hoeffding bound, and it provides an implicit way to balance exploration and exploitation. At each round an arm a is chosen based on the following equation:

$$\max_{a \in A} \bar{x}_a + \sqrt{\frac{2 \; ln \; n}{n_a}} \tag{5.3}$$

where $\bar{x}_a$ is the mean reward of arm a, n is the total number of rounds, and n_a is the number of rounds arm a was pulled. It is evident that this method always explores different arms, therefore not being negatively impacted by an incorrect estimate of an arm's value. However, unlike ϵ greedy, its exploration adapts to an arm's value and reduces over time without having to change the algorithm's parameters.

Another popular method for solving the MAB problem is Thompson sampling, that originates from Thompson (1933). The idea behind this algorithm is to model each arm's underlying reward distribution. At each round, all distributions are sampled, and the arm corresponding to the highest sample is played. Once a reward is received the distribution is then updated. While the idea behind this algorithm

is straightforward, an assumption needs to be made on the nature of the underlying distribution from which the rewards are sampled. This is why Thompson Sampling is a popular algorithm when rewards are binary (Bernoulli trials), as a binomial distribution can be assumed. It is also worth mentioning that a prior for each arm also needs to be provided.

These assumptions are relaxed by BESA (Baransi et al., 2014), an algorithm based on sub-sampling. Given two arms a and b, that were played in n_a and n_b rounds respectively, this algorithm compares their value by averaging $min(n_a, n_b)$ samples without replacement from the arm's past rewards. This approach is based on the intuition that the more an arm is played, the more its potential is known, and it boasts state-of-the-art performance with no prior assumptions on the problem, and without needing parameter tuning.

5.2.2 Dynamic Multi-Armed Bandits

While the Multi-Armed Bandit problem is an interesting case-study for the exploration and exploitation dilemma, its assumption of a static reward distribution cannot always be satisfied. The Dynamic Multi-Armed Bandit (DMAB) problem, first introduced by Whittle (1988), extends the classic MAB problem by allowing the reward distribution of each arm to change over time. More formally, given a family of probability distributions $D(\mu)$ with expectation $\mu \in [0; 1]$, the reward for playing arm i at time t is a random sample from the distribution $D(\mu_i(t))$. The change in reward distributions is modelled in different ways, for example Garivier and Moulines (2008) assumes the distributions are static but are changed a set amount of times, whereas Gupta et al. (2011) models the expectation for each arm i, μ_i, as following Brownian Motion. In our case we make the latter assumption.

Whenever solving the Dynamic MAB problem, the aim is to minimise the regret, defined as:

$$Regret = \sum_{t=0}^{T} r_t^* - r_t \tag{5.4}$$

where r_t^* is the reward gathered by playing the optimal arm at time t and r_t is the reward obtained by the algorithm sampling an arm at time t.

Algorithms to solve the DMAB problem are usually adapted versions of existing MAB algorithms. An example is Sliding-Window UCB (SWUCB), introduced by Garivier and Moulines (2008), which is a version of UCB that only uses the last τ

Algorithm 3 Dynamic Thompson Sampling

Input: Threshold C, Number of arms n

1: $t := 0$
2: $\alpha^a_{-1} := 2 \quad \forall a \in \mathbb{N}_{<n}$
3: $\beta^a_{-1} := 2 \quad \forall a \in \mathbb{N}_{<n}$
4: **loop**
5: For each arm a sample $\hat{\mu}^a_t := Beta(\alpha^a_{t-1}, \beta^a_{t-1})$
6: Pull arm k such that $\hat{\mu}^k_t := max\{\hat{\mu}^1_t, ..., \hat{\mu}^n_t\}$ and receive reward r_t
7: $\alpha^k_t := \alpha^k_{t-1} + r_t$
8: $\beta^k_t := \beta^k_{t-1} + (1 - r_t)$
9: **if** $\alpha^k_t + \beta^k_t > C$ **then**
10: $\alpha^k_t := \alpha^k_t \cdot \frac{C}{C+1}$
11: $\beta^k_t := \beta^k_t \cdot \frac{C}{C+1}$
12: **end if**
13: $t := t + 1$
14: **end loop**

rounds whenever estimating the arm with the highest expected reward. Other variations of UCB for Dynamic Multi-Armed Bandits exist, such as UCB$_f$ (Slivkins and Upfal, 2008), that accounts for possible variance in each arm's reward distribution, or Discounted UCB (Kocsis and Szepesvári, 2006) that calculates the discounted mean for each arm, increasing the impact of recent samples.

Finally, Dynamic Thompson Sampling (DTS) (Gupta et al., 2011) is an extension to the Thompson Sampling algorithm mentioned in the previous section, and it aims to minimise the regret by estimating the reward distribution for each arm as accurately as possible. For each arm a, DTS stores two values: α^a and β^a. These are the parameters of a Beta distribution and are updated according to the reward received from pulling each arm. The difference between standard Thompson Sampling and DTS is that α^a and β^a are bounded by a constant C such that $\alpha^a + \beta^a \leq C$. This constraint, combined with the update rule used for α^a and β^a (lines 7 to 11 in Algorithm 3) results in an exponential weighting in favour of more recent samples, as proven by Gupta et al. (2011). The way DTS selects the arm that should be played next is by sampling the Beta distribution for each arm, and then choosing the arm with the highest sample.

5.2.3 Contextual Bandits

Contextual Bandits are the last variation of the classic Multi-Armed Bandit we will discuss, however they are also the most relevant to the problem of automating the generation of a mapping function. Each round a context is observed, based on this context the algorithm picks an arm and receives a reward. The interesting property of this problem is that the reward is dependent on the context, and no assumption is made on the relationship between the two. An example of a problem that can be solved by using Contextual Bandits is user recommendation (ads, products, videos, etc.), where the context is the user's profile.

If the number of contexts is small, known algorithms for the MAB problems can be used, where one copy of the algorithm can be used for each context (Slivkins, 2019). Another case to be considered is when the number of contexts satisfies the Lipschitz-continuity with a known Lipschitz constant (Hazan and Megiddo, 2007). In this case, the context space can be discretised and the same method described above can be used (Slivkins, 2019) (this intuition will be used in the next section).

5.3 Dynamic Value Bandits

After briefly discussing the relevant Multi-Armed Bandit literature, it is evident that the problem of generating a mapping function closely resembles the Contextual Bandits problem. As seen in Algorithm 2, each round a complexity value is observed, an MDP is selected, and the agent's performance is observed. The complexity value could be equated to the context, the MDP to an arm, and the reward could be the proximity of the selected MDP to the desired complexity. However, while no assumption can be made on the relationship between context and reward in the Contextual Bandits framework, the problem of generating a mapping function has a clear relationship between the two. In order to model this problem we define a novel (to the best of our knowledge) formulation of the Multi-Armed Bandit problem: Dynamic Value Bandits (DVB).

When solving a DVB problem, the agent is provided a target value $\mathbb{V}$ at each time step $t \in [0, T]$. The agent then selects an arm (i_t) whose value is believed to be the closest to $\mathbb{V}_t$, and receives the value of the arm v_{t,i_t}. In this problem, the agent seeks to minimise the regret, that can be defined as:

$$Regret = \sum_{t=0}^{T} (\mathbb{V}_t - v_{t,i_t})^2 \quad (5.5)$$

We assume that the value of each arm changes over time, however, we make no assumption on the process that modifies the arm's value.

In principle, this new definition of regret could be optimised within the standard DMAB framework, where the reward for pulling arm i at time t could be modified to be $-(\mathbb{V}_t - v_{t,i_t})^2$. This however would make it impossible for the algorithm to know the required value of $\mathbb{V}_t$, which would be modelled as an additional factor that changes the reward of pulling an arm. In the case where $\mathbb{V}_t$ is stationary this is not a concern, in fact any upper bound of any algorithm's regret in DMAB would be upheld. However, in the worst case scenario, that is if $\mathbb{V}_t$ was randomly sampled, this strategy would be equivalent to always pulling a random arm. Simply modifying the reward is therefore not the best strategy when no assumptions can be made on the process that generates $\mathbb{V}_t$. As such, we first try to define a possible way algorithms designed for DMAB can be applied to DVB without modifying the underlying algorithm. Later in this section we will modify existing algorithms from the literature in order to specifically optimise the regret metric defined in Equation 5.5.

5.3.1 Using out of the box DMAB methods

In order to define a coherent and principled way to apply algorithms that solve the DMAB problem to DVB, we first define a new variable regret metric. We then prove that a linear combination of these metrics can be created in order to derive Equation 5.5. We finally use this theoretical result to motivate our approach which will be discussed later in this section.

Let v be a real number in the interval $[0, 1]$ and v_{t,i_t} be the expected payoff of the arm played at time t, then the new family of regret functions is defined as:

$$Reg_v = \sum_{t=0}^{T} (v - v_{t,i_t})^2 \quad (5.6)$$

Minimising this regret is equivalent to trying to minimise the difference between v and the value of the arm played. Assuming that v_t^* is the highest value achievable by an arm at time t, minimising $Reg_{v_t^*}$ is equivalent to minimising Equation 5.4. The only difference between the two is in fact that one represents the squared error

whilst the other the absolute error. As such, we consider DMAB to be a special case of DVB. However, the aim behind defining this family of regret functions is to define a set of coefficients $[c_0, ..., c_N]$ that can be used to linearly combine the set of regret functions $[Reg_{v_0}, ..., Reg_{v_N}]$ in order to derive Equation 5.5.

Theorem *Given a DVB problem with a stationary target value $\mathbb{V}_t = \tilde{V}\ \forall t \in [0, T]$ and assuming that the following three constraints are satisfied:*

$$\sum_{j=0}^{N} (c_j v_j) = \tilde{V} \tag{5.7}$$

$$\sum_{j=0}^{N} (c_j v_j^2) = \tilde{V}^2 \tag{5.8}$$

$$\sum_{j=0}^{N} (c_j) = 1 \tag{5.9}$$

Then its regret can be expressed by the following equation:

$$\sum_{j=0}^{N} c_j Reg_{v_j} \tag{5.10}$$

Proof As the regret for a DVB problem is defined by Equation 5.5, we aim to prove the equality between Equation 5.10 and Equation 5.5 assuming the three constraints mentioned in the theorem are valid. We therefore need to prove that the following equation holds:

$$\sum_{j=0}^{N} c_j Reg_{v_j} = \sum_{t=0}^{T} (\tilde{V} - v_{t,i_t})^2$$

We start by substituting Equation 5.6 instead of Reg_{v_j} to then expand the square.

$$\sum_{j=0}^{N} \sum_{t=0}^{T} c_j (v_j - v_{t,i_t})^2 = \sum_{t=0}^{T} \tilde{V}^2 + v_{t,i_t}^2 - 2\tilde{V} v_{t,i_t}$$

$$\sum_{j=0}^{N} \sum_{t=0}^{T} c_j (v_j^2 + v_{t,i_t}^2 - 2 v_j v_{t,i_t}) = \sum_{t=0}^{T} \tilde{V}^2 + v_{t,i_t}^2 - 2\tilde{V} v_{t,i_t}$$

$$\sum_{j=0}^{N} \sum_{t=0}^{T} c_j v_j^2 + c_j v_{t,i_t}^2 - 2 c_j v_j v_{t,i_t} = \sum_{t=0}^{T} \tilde{V}^2 + v_{t,i_t}^2 - 2\tilde{V} v_{t,i_t}$$

We then utilise the commutative property of summations.

$$\sum_{t=0}^{T}(\sum_{j=0}^{N}(c_j v_j^2) + v_{t,i_t}^2 \sum_{j=0}^{N}(c_j) - 2v_{t,i_t} \sum_{j=0}^{N}(c_j v_j)) = \sum_{t=0}^{T} \tilde{V}^2 + v_{t,i_t}^2 - 2\tilde{V} v_{t,i_t}$$

Finally we substitute the constraints in Equations 5.7, 5.8 and 5.9 where appropriate, therefore proving the equality and the theorem.

$$\sum_{t=0}^{T} \tilde{V}^2 + v_{t,i_t}^2 \cdot 1 - 2v_{t,i_t}\tilde{V} = \sum_{t=0}^{T} \tilde{V}^2 + v_{t,i_t}^2 - 2\tilde{V} v_{t,i_t}$$

■

Following this theorem, we develop an approach to adapt any algorithm that solves a DMAB problem to a DVB problem. The idea behind our method is to have a set of N algorithms trying to minimise the regret defined as $[Reg_0, ..., Reg_N]$. We then use the constraints in Equations 5.7, 5.8 and 5.9 in order to calculate a set of coefficients $[c_0, ..., c_j]$. In order to find the set of coefficients, we solve a minimisation problem with constraints, where the target function is the sum of the absolute values of the coefficients. While Equation 5.9 requires all the coefficients to sum to 1, the three constraints don't restrict the sign of the coefficients. Minimising the sum of the absolute value of the coefficients will in turn minimise any negative component that might be needed in order to respect the constraints with the given values of v.

In an actual DVB problem the target value changes at every time step, as such a new set of coefficients needs to be calculated. In practice, the coefficients are pre-calculated up to a desired precision and stored in a hash map. This is to decrease the time complexity of the algorithm. Once a set of coefficients has been defined, a method to choose which arm needs to be played next needs to be defined. The most straightforward and general way to choose the next arm is to clip the coefficients between 0 and 1 and ensure they sum to one. They can then be used as a probability distribution over methods, specifying the probability of each algorithm's output to be chosen. Otherwise, if there is a way to combine the output of different algorithms it can be explicitly defined on a case-by-case basis. This step is where the theorem is not fully respected, and it is also the reason why this method is inspired by the theorem but not strictly theoretically accurate.

Algorithm 4 Dynamic Value Best Empirically Sampled Average

Input: Discount factor γ

1: $t := 0$
2: $V_i := \{\} \quad \forall i \in \{1, 2\}$
3: $P_i := \{\} \quad \forall i \in \{1, 2\}$
4: **loop**
5: Receive target value $\mathbb{V}_t$
6: $n := \min_i |V_i|$
7: $\bar{S}_i :=$ sample and avg. n elements *with replacement* from V_i with prob. $\frac{P_i}{|P_i|}$
8: Play arm a with lowest $(\bar{S}_i - \mathbb{V}_t)^2$, or lowest $|V_i|$ if tied
9: Observe arm's value v_a
10: $V_a := V_a \cup \{v_a\}$
11: $P_i := P_i \cdot \gamma \quad \forall i \in \{1, 2\}$
12: $P_a := P_a \cup \{1\}$
13: $t := t + 1$
14: **end loop**

5.3.2 Custom DVB algorithms

While it might be useful to discretise the context space and combine the output of different instances as described in the last section, we also adapted three algorithms (SWUCB, BESA and DTS) to solve the full DVB problem. This is in an effort to increase the sample efficiency (therefore decreasing regret), compared to our previous method.

5.3.2.1 Dynamic Value Thompson Sampling

The algorithm whose adaptation was most natural and straightforward was Dynamic Thompson Sampling. This is because the algorithm is already equipped to address a change in the reward distribution. Moreover, most of the algorithm is centred around estimating the value of each arm as accurately as possible, requiring a minor modification of only one line (line 6 in Algorithm 3) in order for the conversion to take place. As the aim of DVB is to play the arm with the value closest to a desired value, Dynamic Value Thompson Sampling (DVTS) produces one sample per arm, however it only pulls the arm whose sample is closest to the desired value.

5.3.2.2 Dynamic Value BESA

While adapting Thompson Sampling to solve the DVB problem only required a small adjustment, BESA required a more significant deviation from the original method.

In particular two challenges need to be separately addressed: being able to specify the target value and dynamic reward distributions. The former is the easiest to address, as BESA is a sampling based algorithm it again requires a single line to be modified. In particular instead of selecting the sample with the highest value, the sample with the value closest to the target is selected. On the other hand, addressing the issue of dynamic reward distributions required a way to give more weight to more recent samples, while not changing the nature of the algorithm altogether. This was achieved by introducing a vector P used to derive the probability of sampling each value from the history. Whenever a new sample is observed, all values in each P vector are multiplied by the discount factor, and a one is added to the corresponding P vector. Assuming an arm has been played in rounds $[r_1, .., r_n]$ and that the current round is t, then the vector P will be $[\gamma^{t-r_1}, ..., \gamma^{t-r_n}]$. In order to utilise this vector in the sub-sampling, we sample with replacement (unlike the original algorithm). If this was not the case and two arms had the same number of samples, then sampling without replacement would be equivalent to not sampling. Each sample s from the history has a probability $\frac{P_s}{|P|}$ of being sampled; coupled with the way P is calculated, it results in recent samples being exponentially more likely to be selected compared to less recent ones.

The novel proposed algorithm, named Dynamic Value BESA (DVBESA), can be seen in Algorithm 4. Like its original counterpart, the algorithm can only compare the value of two arms, and if multiple arms are present, a single elimination tournament is played between them.

5.3.2.3 Value Sliding Window UCB

The last algorithm we adapt to the DVB problem is UCB. In order to address the dynamic nature of reward distributions, UCB has several variants, three of which were mentioned in Section 5.2.2. Out of these variants we choose SWUCB, as it is able to adapt better than UCB_f to a change in reward distribution, given that UCB_f considers the average of all rewards for each arm without giving more weight to recent samples. Furthermore, as SWUCB and Discounted UCB were shown to have a similar performance (Garivier and Moulines, 2008), we chose the former as its parameter selection is more simple.

In order to encourage the algorithm to choose values close to the target value $\mathbb{V}_t$, instead of choosing the arm with the highest mean we choose the algorithm where the mean is closest to the target value. As the target value has no influence on the

confidence bound, the second term was not modified. Intuitively, out of the arms whose mean value is lower than the target value, the arm with the highest mean plus confidence interval will be prioritised. On the other hand out of the arms with a mean value greater than the target value, the one with the lowest mean minus the confidence interval will be prioritised. Out of the two arms that were prioritised, the arm chosen will be the one whose confidence interval extends the furthest from the target value. Given a window size τ, this selection policy is expressed by the following equations:

$$\max_{a \in A} -|\mathbb{V}_t - \frac{1}{N_t(\tau, i)} \sum_{s=t-\tau}^{t} v_s 1_{\{a_s = i\}}| + \sqrt{\frac{log(\min(t, \tau))}{N_t(\tau, i)}} \tag{5.11}$$

$$N_t(\tau, i) = \sum_{s=t-\tau}^{t} 1_{\{I_s = i\}} \tag{5.12}$$

5.4 DVB generated mapping functions

In Chapter 3 we define a mapping function as a function $\Phi_D : [0,1]^n \to \mathcal{M}$ that maps a complexity vector with components between 0 and 1 to an MDP of the corresponding complexity. In order to solve this problem over one dimension we first define a fixed and finite set of MDPs, $\mathcal{M}$, that can be either randomly sampled or provided by an expert. We then model each MDP as an arm in a DVB problem, and aim to select the MDP of the appropriate value by translating complexity to target value using Equation 5.1. This however poses a challenge, in particular $p_\theta^+ = \max_{m \in \mathcal{M}} \bar{P}(\theta, m)$ and $p_\theta^- = \min_{m \in \mathcal{M}} \bar{P}(\theta, m)$ need to be estimated. As each method estimates the value of arms in a different way, this estimate might be natural for some methods while complex for others.

For Dynamic Value Thompson Sampling, after each round of sampling p_θ^- is estimated to be the sample with the minimum value and p_θ^+ the sample with the highest value. On the other hand, for DVBESA, each sample's latest empirical average is saved to memory, calculated in Algorithm 4, line 7. Finally VSWUCB estimates p_θ^- and p_θ^+ to be respectively the minimum and maximum value of the following expression:

$$\frac{1}{N_t(\tau, i)} \sum_{s=t-\tau}^{t} v_s 1_{\{a_s = i\}} \tag{5.13}$$

While the problem of estimating the maximum and minimum value of the agent's performance could seem like a trivial extension of existing methods, in a practical scenario with a progression function, most of the sampled MDPs should be of a similar complexity, resulting in a potentially inaccurate estimate of either bound.

5.5 Initial Evaluation

This section reports the initial evaluation performed on the different methods we introduced in this chapter. We will first test the performance of the methods on the DMAB problem, to then report their performance on the DVB problem with a randomly selected value. We then introduce an adversarial setting for the DVB problem, where the adversary selects the target value with the aim of maximising the regret. Finally we introduce a model of the agent's learning based on real data gathered by training the agent on the Grid World Maze environment. This preliminary evaluation serves to select one of the three methods introduced in Section 5.3.2, based on its performance and robustness, in order to be applied to generate a mapping function for a learning agent.

5.5.1 DMAB and DVB testing environments

DMAB The first setting in which all algorithms were tested was a DMAB environment inspired from the existing literature in the field (Gupta et al., 2011). The environment consists of 10 arms which were randomly initialised to random values $[v_{0,1}, .., v_{0,10}]$ in the interval $[0, 1]$. Each arm's value corresponds to the probability of success of the Bernoulli trial associated with each arm. Every round each arm's value changes over time according to Brownian motion with cutoff boundary, defined by the following equation:

$$v_{t+1,i} = (v_{t,i} + N(0, \sigma^2))^{\top 1}_{\perp 0} \;\; \forall \; i \in \{1, ..., 10\} \tag{5.14}$$

where N(a, b) is a normal distribution with mean a and standard deviation b, $a^{\top b}$ is equivalent to the minimum value between a and b and $a_{\perp b}$ is equivalent to the maximum value between a and b. In this setting, the regret is defined by Equation 5.4, where the value of the arm (rather than the result of the Bernoulli trial) is considered.

DVB The second problem where each algorithm was tested is an adaptation of the previously mentioned environment to DVB. In particular, each arm's value changes over time according to Equation 5.14. At each round t, the target value $\mathbb{V}_t$ is set to be the value of a random arm $v_{t,i} i \in \{1, ..., 10\}$. The regret for this problem is defined by Equation 5.5, and it is going to be the metric of comparison between methods on this environment.

Adversarial DVB While the DVB testing environment defined in the last paragraph will provide some insight on each algorithm's ability to adapt to changes in each arm's value, the target value being chosen as a random arm might reduce the complexity of the problem. If a random sampling is performed the algorithms will have the opportunity to track the value of each arm more easily, compared to other possible target value selection policies. For this reason we introduce the Adversarial DVB environment, where the adversary's aim is to select the target value that will maximise the agent's regret.

At each round t the adversary selects an arm i, the target value is then set to be $\mathbb{V}_t = v_{t,i}$. The algorithm selects the arm a deemed to be the closest in value to $\mathbb{V}_t$, and receives the output of a Bernoulli trial with probability of success $v_{t,a}$. The algorithm's regret for the round is then calculated as $(v_{t,i} - v_{t,a})^2$ and it is provided to the adversary as reward for selecting arm i. While the algorithm is in a DVB environment, the adversary is essentially solving a DMAB problem with the reward being the algorithm's regret. This adversarial setting will test each algorithm's robustness to the order in which the target values are selected, which is a key factor in assessing which algorithm should be used to generate a mapping function for a learning agent.

5.5.2 Data driven environment

In the existing literature on the DMAB problem, the assumptions on the process that changes the value of the arms are either minimal (Auer et al., 2002b) or of a more general nature (such as Brownian Motion (Gupta et al., 2011)). We followed the latter approach when creating the testing environments mentioned in the last section, however in the practical problem of generating a mapping function the underlying cause of a change in the value of each arm is known. It is in fact the agent's learning that changes its performance on each MDP, and its expected value as a result. For this reason we generate a novel testing environment, aimed at

assessing each algorithm's ability to generate an accurate mapping function.

The data for this model was collected by training an agent on the Grid World Maze environment 200 times using the Friction-based progression. After each one of the 125 epochs, its probability of reaching the goal from each state was assessed 100 times. This extensive assessment of the agent's performance enabled us to track the way its ability evolves over time, and therefore to use the data as an approximation of a learning agent's behaviour when learning with a curriculum. This has the advantage of being both a lot faster and more repeatable than training an agent on an environment, and allows us to isolate the variable of the agent's learning performance and focus on assessing the accuracy of the mapping function produced by each algorithm over 200 separate runs.

At the start of each round, a progression function specifies the complexity of the environment. The algorithm then converts the complexity value to a target value (as described in Section 5.4) and returns the arm believed to be closest to the target. The algorithm then observes the result of a Bernoulli trial with the probability of success being the probability of the agent reaching the goal from that state based on the data. The error of the algorithm's choice is then calculated, and the progression function is updated with the new performance (if needed). Each arm's probability of success is initialised according to the agent's ability in the first epoch, and every 80 rounds it is modified to match the agent's performance in the following epoch.

The way each algorithm's accuracy is assessed for this environment is the error calculated using the following formula:

$$Error = |Comp(\theta_t, m_t) - c_t| - \min_{m \in M} |Comp(\theta_t, m) - c_t| \tag{5.15}$$

Note that this definition of error differs from the one in Equation 5.2. This variation is necessary in order to prevent the complexity value chosen by the progression function influencing the error term. To better illustrate this, assume there is only one task with complexity 0, and all other tasks are of complexity 0.5 and above. If the complexity required by the progression function is 0.4 and a task of complexity 0.5 is chosen, Equation 5.2 would result in an error of 0.1. However if the complexity required was 0.3, choosing the same task would result in an error of 0.2. As in our example the task which is closest to the desired complexity is the one chosen (of complexity 0.5), the only difference between the two error terms resides in the progression function. On the other hand, calculating the error with Equation 5.15 would result in both errors being equal to zero (as the task chosen was the one closest

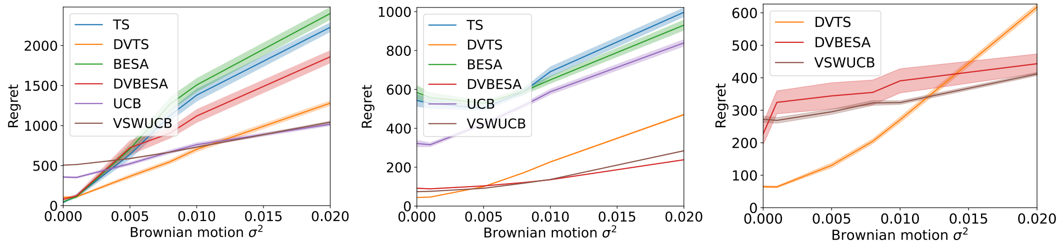

Figure 5.1: The figure on the left shows the regret of different methods on the DMAB problem. The figure in the center, on the other hand, reports the regret on the DVB environment. Finally, the figure on the right reports the regret on the adversarial DVB setting against the most effective adversary. Note that the higher the regret, the worse the performance.

to the desired complexity). While this metric does not reflect the actual proximity of the chosen tasks to the desired complexity, it is the appropriate one for comparing different algorithm's performance without any influence from external factors.

5.5.3 Experimental results

This section will present the experimental results of each method on all of the environments mentioned above. The goal of this evaluation is to determine the most robust and highest performing algorithm for the automatic generation of mapping functions.

5.5.3.1 DMAB

Our evaluation begins by comparing each algorithm's performance on the DMAB environment. This is useful to ascertain their ability to adapt to a change in the arm's value over time. Like in Gupta et al. (2011), we test each algorithm's performance on different settings of the environment by changing the standard deviation (σ^2) of the Brownian motion (see Equation 5.14). In particular, we test the performance of each algorithm on $\sigma^2 \in \{0, 0.001, 0.005, 0.008, 0.01, 0.02\}$, by averaging its regret over 100 random seeds. Besides evaluating the three methods we defined, we also evaluate their static counterpart in order to highlight the benefits of giving more weight to recent samples.

When observing Figure 5.1 (left), all algorithms except for UCB and VSWUCB have a similar performance on the standard Multi-Armed Bandit problem (when $\sigma^2 = 0$), however, when increasing the standard deviation of the Brownian motion, Thompson Sampling and BESA have the highest regret out of all methods. Surpris-

ingly, UCB is amongst the best performing algorithms in this setting, despite not accounting for dynamic reward distributions. Moreover, DVBESA's regret increases at a pace not dissimilar to the one of standard BESA. DTS, on the other hand, is clearly the best performer for values of σ^2 below 0.01, however it is outperformed by VSWUCB and UCB on higher values of σ^2.

5.5.3.2 DVB

In order to assess the suitability of each method to be used in the generation of a mapping function we also test each algorithm on the DVB problem, where the static version of each algorithm is adapted using the method defined in Section 5.3.1 with 101 evenly spaced values of v_j.

In Figure 5.1 (center), out of the algorithms using interpolation (TS, BESA and UCB), the best performing one was UCB, with Thompson Sampling and BESA's confidence intervals intersecting for most values of σ^2. DVBESA and VSWUCB's performance was similar across all values of σ^2, whereas VDTS was the best performing algorithm for value of σ^2 lower than 0.005, however for larger values of σ^2 the other methods had a clear advantage. This result is surprising, given DVBESA's comparably high regret in the DMAB setting.

5.5.3.3 Adversarial DVB

The Adversarial setting, as mentioned previously, aims to test the robustness of different methods to how the target value is selected. In particular, the DVB version of each algorithm was put against each DMAB algorithm, and the worst performance was chosen to generate Figure 5.1. The full results are reported in a table format in Section A.4.

While in the DVB problem the best performing algorithms were VDBESA and SWUCB, the results change as the value is not randomly sampled from all arms. VDTS, in fact, is the most resilient algorithm for values of σ^2 up to 0.01, after which SWUCB was the best performing method. Moreover, VDTS was the algorithm with the lowest difference between the adversarial setting and whenever no adversary was present. This highlgihts the robustness of the method, especially when compared against DVBESA, which has a very wide confidence interval, suggesting that it is either able to quickly adapt to the adversary's strategy or its regret is considerably higher compared to other methods.

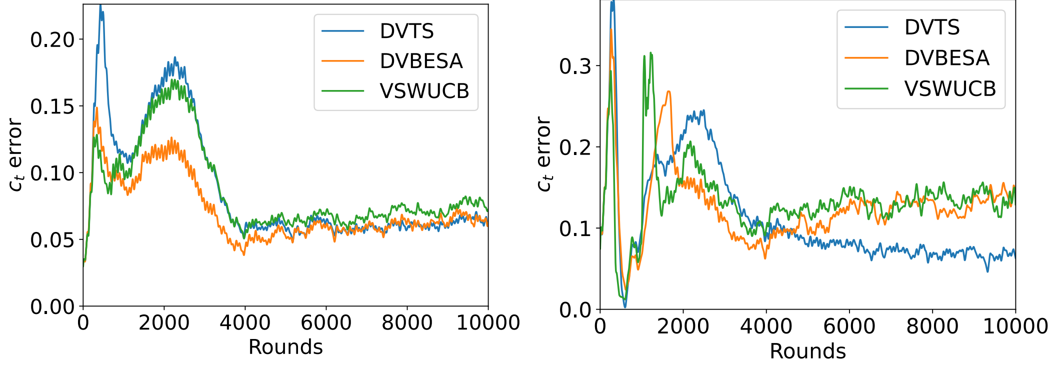

Figure 5.2: Average error for each algorithm when the complexity is randomly sampled (left), or specified by the Rolling Sphere progression (right). Each line is the average of 200 runs smoothed with a Savitzky–Golay filter.

5.5.4 Data driven environment

The final environment used for our preliminary testing is the data driven model described in Section 5.5.2, which simulates how a learning agent's performance evolves over time. As can be seen in Figure 5.2, two distinct settings for this environment were considered: one where the target complexity was randomly sampled between 0 and 1, and one where the target complexity was specified by the Rolling Sphere progression. Unlike the previous environments, where the performance of each algorithm was evaluated based on their regret, the metric of comparison between different methods was the one defined in Equation 5.15.

When the complexity is randomly sampled, DVBESA is the best performing algorithm as can be seen in Figure 5.2 (left). This is due to its better performance from 1000 to 4000 rounds; in the same time interval DVTS and VSWUCB both have a spike in their error. Moreover DVTS has an initial spike in error not shared by the other two methods. It is worth noting that after 5000 rounds, DVTS's performance is on par with DVBESA, while VSWUCB's error is marginally higher in the later rounds.

The results change when the complexity is controlled by the Rolling Sphere progression. This is the setting we are more interested in, as it is representative of the use case for this method. This setting is also more of a challenge as estimating the environment's complexity through Equation 5.1 also involves estimating the highest (p_θ^+) and lowest (p_θ^-) value out of all tasks for the current agent's state θ. While in a setting with a random target complexity it is straightforward for methods to keep an updated estimate of both values, if the target complexity is steadily increasing

over time, these estimates will be increasingly outdated and will potentially limit the performance of methods in this setting. This results in an increased error for DVBESA and VSWUCB especially after the 4000th round, whereas DVTS's error decreases steadily. However, DVTS is the worst performing algorithm from round 2000 to round 3500, with DVBESA outperforming it until the 4000th round. It is worth noting that both VDBESA and VSWMAB have a spike in error at around 1000 and 1500 rounds respectively, which VDTS does not share.

5.5.5 Initial evaluation summary

The goal of this extensive evaluation was to select an algorithm to be fine tuned to the problem of generating mapping functions. Our initial results on the DMAB problem highlighted the benefits of VSWUCB for high values of σ^2 and DVTS for lower values of σ^2. Whenever testing on the DVB problem, both DVBESA and VSWUCB performed similarly, with DVTS being the best performing algorithm only for $\sigma^2 \in [0, 0.005]$. However, whenever introducing an adversary that selects the target value, DVTS was clearly the best performing algorithm until $\sigma^2 > 0.01$, with its performance never deviating significantly from its performance with a randomly selected target value. DVBESA on the other hand was the algorithm that was most affected by the introduction of an adversary, with its 95% confidence interval being quite wide considering it was derived from 100 runs. It is evident that the best algorithm in this case is dependant on the value of σ^2: in general VSWUCB's performance seemed to be more reliable compared to DVBESA both with and without an adversary. On the other hand, the results seem to suggest that DVTS is the best algorithm for low values of σ^2.

Whenever looking at a testing environment which is closer to the intended use case, the DVTS algorithm seems to fall behind whenever the complexity is randomly selected, but significantly outperform the other two algorithms from 4000 rounds onward when the complexity is specified by the Rolling Sphere progression. This seems to imply that initially the values of the arms change frequently, but later they stabilise, resulting in the accuracy and robustness of DVTS being shown.

As a result of this evaluation, DVTS was chosen as the algorithm to be optimised in order to generate a mapping function for learning agents.

5.6 Fine-tuning VDTS

As VDTS has been shown to be the best suited algorithm to generating a mapping function, we use our knowledge of the domain in order to fine-tune the algorithm. Furthermore we aim to identify the flaws in the algorithm based on our previous experimental evaluation and try to address them within the new method.

Based on the experimental results on the DVB problem, it is evident that this algorithm's weakness is reward distributions that change quickly. As such, we keep track of the change in value of any given task over time, by performing an exponential moving average on the first derivative of the values for each arm, and encourage the algorithm to choose tasks with a higher rate of change in order to better update their distribution. Furthermore a useful bias whenever generating a mapping function (and even in the DVB problem) is to prioritise arms that have not been pulled in a while; this could in fact help with keeping an up to date estimate of each task. As DVTS is an algorithm that is based around choosing the sample with the smallest distance to the target value, there is no clear way to enforce these priority rules, as such we introduce a zone $[c-m, c+m]$ around the target complexity c of magnitude m. Any MDP whose complexity falls in this zone will be labelled as a candidate, and a weighted random sampling will take place amongst the candidates in order to choose the appropriate MDP. This will allow us to define a weight in order to bias the selection of the MDP using the principles outlined above.

Another factor that is specific to generating a mapping function that needs to be accounted for is that the target task is known. This is in fact an assumption made by most Curriculum Learning algorithms, as the performance of the agent on the target task is what Curriculum Learning ultimately wishes to optimise. As such, we are not interested in learning tasks in the curriculum space that are more complex than the target task; given a set of parameters for the agent, θ, and a target task t, let the agent's performance on the target task be p_θ^t, then the complexity of a task with a known target task becomes:

$$Comp(a, m) = 1 - \frac{\bar{P}(\theta, m) - p_\theta^t}{p_\theta^+ - p_\theta^t} \tag{5.16}$$

Another issue to consider when trying to generate a mapping function is the number of MDPs to choose from can be quite large, and sometimes there are MDPs that have a strong similarity. In order to improve the initial performance of this algorithm, we employ the strategy of an automatically defining a similarity metric

Algorithm 5 Indirect Thompson Sampling

Input: Threshold C, Number of tasks n, Threshold τ

1: $t := 0$
2: $\alpha^a_{-1}, \beta^a_{-1}, \tilde{\alpha}^a_{-1}, \tilde{\beta}^a_{-1} := 1 \quad \forall a \in \mathbb{N}_{<n}$
3: **loop**
4: For each arm a sample $\mu^a_t := Beta(\alpha^a_{t-1}, \beta^a_{t-1})$
5: For each arm a let $c^a_t := \frac{\mu^a_t - \mu^1_t}{\max_{x\in[1,n]} \mu^x_t - \mu^1_t}$
6: For each arm a sample $\tilde{\mu}^a_t := Beta(\tilde{\alpha}^a_{t-1}, \tilde{\beta}^a_{t-1})$
7: For each arm a let $\tilde{c}^a_t := \frac{\tilde{\mu}^a_t - \tilde{\mu}^1_t}{\max_{x\in[1,n]} \tilde{\mu}^x_t - \tilde{\mu}^1_t}$
8: For each arm a $\hat{c}^a_t := \frac{\alpha^a_{t-1}+\beta^a_{t-1}}{C} \cdot c^a_t + (1 - \frac{\alpha^a_{t-1}+\beta^a_{t-1}}{C}) \cdot \tilde{c}^a_t$
9: Get complexity value c_t from the progression function
10: Define set of candidates $\mathbb{C}$ containing each arm a satisfying $|\hat{c}^a_t - c_t| < \tau$
11: Pull arm k by sampling from $\mathbb{C}$ and receive reward r_t.
12: Update α^k_t, β^k_t like in Dynamic Thompson Sampling.
13: Normalise $\tilde{\alpha}^k_t$, $\tilde{\beta}^k_t$ such that their sum is $\leq C - 1$
14: Update $\tilde{\alpha}^k_t$, $\tilde{\beta}^k_t$ like in Dynamic Thompson Sampling.
15: **for** $a \in \mathbb{N}_{<n} - k$ **do**
16: $sim := JensenShannon(Beta(\alpha^k_{t-1}, \beta^k_{t-1}), Beta(\alpha^a_{t-1}, \beta^a_{t-1}))$
17: $\tilde{\alpha}^a_t := \tilde{\alpha}^a_{t-1} + sim \cdot r_t$
18: $\tilde{\beta}^a_t := \tilde{\beta}^a_{t-1} + sim \cdot (1 - r_t)$
19: **end for**
20: $t := t + 1$
21: **end loop**

between MDPs based on the similarity between their estimated reward distributions. The intuition is that in the initial stages of the training, this similarity metric can be used to update the value of multiple tasks with similar expectations based on the results from only one. The weight of this estimation will then be annihilated as more information is available on each individual task. In order to put this idea in practice, we introduce two different distributions for each task: a *strict* distribution and a *lenient* distribution. Strict distributions, as the name suggests, only get updated when a value is observed for the MDP relative to the distribution. On the other hand, lenient distributions are updated any time a value is observed, and the amount they are updated by depends on the Jensen-Shannon distance[1] between the strict distributions relative to the MDPs.

Combining the techniques mentioned above with Dynamic Thompson Sampling

[1] Square root of the Jensen-Shannon divergence (Lin, 1991), used to measure the distance between two probability distributions.

results in Algorithm 5, named Indirect Thompson Sampling, as the value of $\tilde{\alpha}$ and $\tilde{\beta}$ (lenient distribution) are mainly derived indirectly from similar MDPS. Initially both the strict and lenient distributions are initialised, then the complexity according to both distributions is calculated according to Equation 5.16. Line 10 defines the previously mentioned set of candidates, whereas line 11 selects the appropriate arm. The method of selection is a weighted sampling based on the last time an arm was pulled and the momentum of the arm's values (exponential moving average of the first derivative of the values of the arm). Finally all the distributions are updated.

It is worth noting that in order to limit the influence of other arms on the value of the lenient distributions, they are not normalised such that $\tilde{\alpha}_t^a + \tilde{\beta}_t^a \leq C$ unless a is the arm that was last pulled. This results in a much higher weight being given to samples relative to that arm for lenient distributions. Another detail that was omitted for simplicity is that in line 6, whenever the sampling takes place, $\tilde{\alpha}_{t-1}^a$ and $\tilde{\beta}_{t-1}^a$ are normalised such as their sum is equal to their strict counterpart. This accounts for the fact that the lenient distributions are not always normalised, and that updating the distribution based on similar tasks does not actually change the confidence in the value of that task. This normalisation ensures that if a limited amount of data is available for a certain task its lenient distribution does not have an unjustifiably (based on the amount of data available) small variance.

5.7 Experimental Evaluation

In order to validate our method we perform an evaluation of VDTS and ITS on our testing domains. This evaluation has the aim to test the effectiveness of our approach as well as whether our modification of VDTS to generate mapping functions was more effective than their manually generated counterpart. The manual mapping functions were the ones described in Section 3.8.2, which were also used for all other experiments in this thesis. Details on the principles used to manually generate mapping functions can be found in Section 3.4.

The results on the Grid World Maze environment show an initial rise in performance when using both automatically generated mapping functions, however the mean performance of the manually defined mapping function surpasses both automatically defined ones later in the training. While the confidence intervals intersect for much of the training, by looking at the performance of the three methods in this environment we can estimate that the manually defined mapping function should be the best performing method, followed by ITS and then by VDTS.

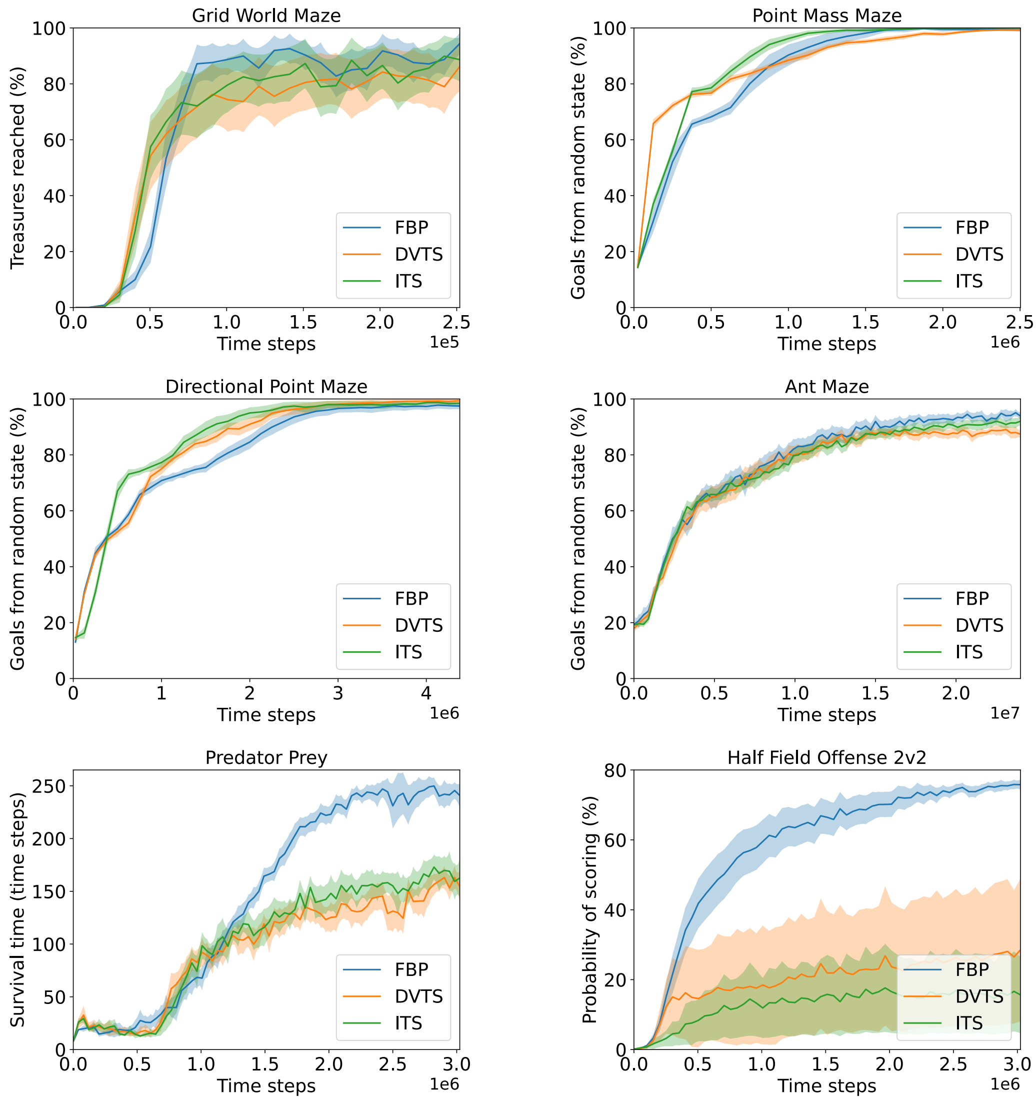

Figure 5.3: Performance of a manually defined mapping function against VDTS and ITS.

The Point Mass Maze domain provides a more clear distinction between the methods, with ITS being the bet performing method for all but the first fifth of the training. While VDTS has a better initial performance, its performance lags behind in the later stages, being outperformed by both ITS and a manually defined mapping function. Interestingly the automatic mapping functions are able to perform better than a manually defined one in this environment, which is surprising given only the x and y coordinates are modified by the mapping function.

In the Directional Point Maze domain, both automatic mapping functions out-

perform a manually defined one, with ITS outperforming VDTS in the middle stages of the agent's training but converging to a similar value. Both automatic mapping functions performing well in this task can be attributed to the manual mapping function initialising the directional model in a random heading, whereas the automatic mapping functions have control over the initial heading of the agent, which is translated to an increase in performance.

The Ant Maze domain sees all three mapping functions perform in a similar way until three quarters of the training, when VDTS's performance converges, resulting in a best performance outside of the confidence interval of ITS and a manually defined Mapping. ITS, on the other hand performs better than DVTS, but marginally worse than a manually defined mapping functions in the later stages of the training.

The Predator Prey domain, on the other hand, sees both automatically generated mapping functions outperformed by the manually defined one. In this domain, ITS's mean performance is higher compared to DVTS, however they are still within the same confidence interval throughout the training.

Finally, the HFO domain sees similar results compared to the Predator Prey domain, with DVTS being outperformed by the manually defined mapping function, and ITS being within the same confidence interval as DVTS for the whole training. Interestingly this is the only domain where the mean performance of DVTS outperforms ITS's mean performance, however given the amount of overlapping between the two confidence intervals (and their magnitude), this is not an accurate estimate of which algorithm performs best in this domain.

Our experimental results show that there is indeed a benefit in using ITS over VDTS, as the former outperforms or matches the latter in all the environments but one (HFO), in which the magnitude of the confidence intervals prevents us from making a prediction on which method outperforms the other. From this experimental evaluation we conclude that the performance of automatic mapping functions created with no previous domain knowledge is domain dependent. In particular, our approach performs well on navigation domains, where it can match or even outperform manually created mapping functions. On the other hand, our approach is outperformed by manually defined mapping functions on both Predator Prey and Half Field Offense. We hypothesize that this behaviour is caused by the environments being much harder to learn whenever the selected task is randomly sampled compared to our navigation domains. As in ITS and VDTS all priors are initialised to $\alpha = \beta = 1$, the initial stage of the curriculum will be a random sampling in order to ascertain which tasks are more complex and which tasks are less complex. This

however could negatively impact the agent's learning from the offset, potentially resulting in the agent never scoring in HFO, and therefore all tasks being of maximum complexity, or the predator always catching up to the agent. Moreover, the task's parameters randomly changing at the end of each episode might not give the agent enough time to learn a less complex parameter configuration even if given the chance to train in one. This hypothesis is supported by our experiments on the robustness of the adaptive progression functions conducted in Chapter 4. Figures A.3 and A.4 are relevant to this problem, more specifically the effect of random noise in the performance of both progression functions in the Predator Prey and HFO domains. HFO was in fact the only domain where the best performance of both progression functions with random noise was more than halved, similarly Predator Prey was the domain where random noise has the second highest impact.

Chapter 6

Conclusion and Outlooks

This chapter concludes the thesis by first outlining the contributions made to the field of Curriculum Learning in Reinforcement Learning, to then discuss the limitations of the work presented. Finally, a way in which the limitations could be addressed will be discussed in the section dedicated to future work.

This thesis empirically proved the research hypothesis stated in the introduction:

High-level domain knowledge can be used to support the generation of an informed curriculum that can train an RL agent more efficiently than other state-of-the art Curriculum Learning methods.

in particular with the experimental evaluation in Chapter 4. Moreover it answered the three research questions it set out to explore:

1. *How can high-level domain knowledge be leveraged in the creation of a curriculum?*
2. *How can the complexity of the environment be selected for a given learning agent?*
3. *How can the complexity of an environment be assessed for a given learning agent?*

6.1 Contributions

Defined a novel Curriculum Learning framework Chapter 3 defined a novel framework for curriculum generation based on two components: progression functions that specify the environment's complexity, and mapping functions that generate environments of a specific complexity. In doing so, it addressed research questions

1 and *2*. Various aspects of the framework were explored in Chapter 3, moreover Chapters 4 and 5 deal with the automation of the components of this framework.

Defined a way to exploit high-level domain knowledge Chapter 3 also defined a way to allow for simple yet significant knowledge about the environment to be used in the generation of a curriculum, therefore directly answering research question *1*. This happens through mapping functions, whose manual generation via domain knowledge is addressed in Chapter 3.

Determined how long should be spent on each task The framework defined in Chapter 3 also answers an open question in the Curriculum Learning literature: how long should be spent on each task. This is specified by progression functions within the framework defined in this thesis. We addressed this in Chapter 3, in particular Section 3.6 when discussing curriculum granularity. Granularity in fact deals with how often the complexity of the environment is modified; our experimental results strongly suggest that changing the task up to every episode is the most beneficial. This contribution addressed research question *2*.

Built an online curriculum tailored to the agent Chapter 4 defined several methods to automatically build a curriculum tailored to the agent's ability, therefore addressing research question *2*. In particular, the Friction-based and Rolling Sphere progression functions were introduced. The Rolling Sphere progression was also used to progress each dimension separately by performing a line search over each dimension in order to estimate the gradient.

Automated the generation of a mapping function Our framework is not limited to a manually defined mapping function, in fact Chapter 5 automates the creation of mapping functions. A new algorithm for the generation of mapping functions is defined, and our empirical results show that an automatic mapping function can even outperform a manually defined one. In order to automatically define a mapping function, the complexity of different tasks was assessed for different agents, therefore answering research question *3*.

Defined a new variation of the Multi-Armed Bandit problem In order to automate the generation of a mapping function, Chapter 5 defined a new variation of the Multi-Armed Bandit (MAB) problem. This variation, named Dynamic Value

Bandits, requires the agent to pull the arm whose expected payoff is closest to an observed value. Thompson Sampling, UCB and BESA were adapted to solve the DVMAB problem, and an extensive evaluation was performed over several testing domains. This contribution was vital in the automation of the generation of mapping functions, therefore addressing research question *3*.

6.2 Limitations

The most evident limitations posed by the framework in Chapter 3 are the need for expert knowledge in order to generate a mapping function, and the need to manually define how the complexity should be changed over time. These limitations are addressed in Chapters 4 and 5, however the methods used to automatically generate mapping and progression functions also have some limitations.

The first adaptive progression function defined, the Friction-based progression, can only progress all dimensions of complexity simultaneously, a limitation that is addressed by the rolling Sphere progression. However, the method used to progress each dimension of complexity independently can only progress one dimension at a time. Furthermore, as this approach is based on a gradient estimation it is susceptible to noise in the mapping function.

On the other hand, the way we automatically generate mapping functions has an inherent limitation stemming from the Multi-Armed Bandit framework: a fixed set of tasks (or arms) needs to be provided at the start of the training. This is necessary for each algorithm to keep track of the value of each arm. Another limitation, that in certain cases could be an advantage, is the fact that we do not perform any generalisation over the parameter vector, for example by using a neural network to estimate the complexity. This was a conscious choice and it has two motivations. Firstly, similar parameter vectors can have a very different complexity which might not be easily captured by a function approximator. Moreover using a function approximator would impose a limit on the set of MDPs that can be part of the curriculum, whereas in our case any MDP can be used provided a valid transfer method is used whenever needed. Finally, a limitation of our method of automating the generation of mapping functions is that it under-performs in certain environments, requiring further experimentation in order to identify any possible areas of improvement.

6.3 Future work

As previously mentioned, the field of Curriculum Learning for Reinforcement Learning is a relatively young field of research, having been formalised by Narvekar et al. (2016). As such, many open questions are still unanswered and many different directions could be taken by future work in this field. This section will conclude the thesis by mentioning some ideas for future work that are related to the work undertaken in this thesis.

One possible research avenue for future work would be to define a method to progress each dimension of complexity simultaneously. This would provide a more fine-grained control over the progression itself and would also be potentially more resilient to noise in the mapping function. In order to address this problem, existing policy search methods could be used, with the aim to keep the agent in environments where its performance is within given thresholds.

Another opportunity for future work would be to address both the second and third limitation mentioned in the last section by using a model to estimate the complexity of the environment. This would allow for a random set of MDPs to be chosen whenever needed, and the way MDPs are generated could be further expanded upon. In our current research MDPs are randomly sampled from the parameter vector, although in principle any generation method could be used. We performed preliminary experiments in trying to use the agent's value function to determine the complexity of different MDPs, however this would impose restrictions in the set of MDPs that can be added to the curriculum. Moreover, this method is dependent on the quality of the agent's initial value function, as a bad initialisation could result in the agent being stuck in more complex environments.

As mentioned in the previous section, our algorithm for the generation of mapping functions under-perform in some domains, as such future work could address this limitation. A way this could be achieved is by limiting the amount of domain knowledge necessary for the generation of the mapping function to only estimating the priors for each task's beta distribution in the ITS algorithm. This could result in the problem of random exploration at the start of the training being overcome, while relaxing the assumptions made on the generation of mapping functions, and still being able to adapt the mapping function to each individual agent.

Furthermore, an interesting open question to answer would be whether it is possible to define a universal agent that acts as a progression function, specifying the desired complexity over time. Many challenges arise from this research idea, one

being how to structure the reward function, another being how to train an agent capable of accounting for temporal dependencies and changing environments. While the idea of defining the generation of a curriculum as an MDP was already explored by Narvekar et al. (2017), having an agent specify the complexity of the environment could lead to a more streamlined algorithm.

Finally, a theoretical analysis of the DVB problem could be undertaken, in particular estimating an upper bound to the regret for each one of the newly defined algorithms, and more algorithms for solving this problem could be defined.

Appendix A

Additional Results

A.1 Friction Based progression variants

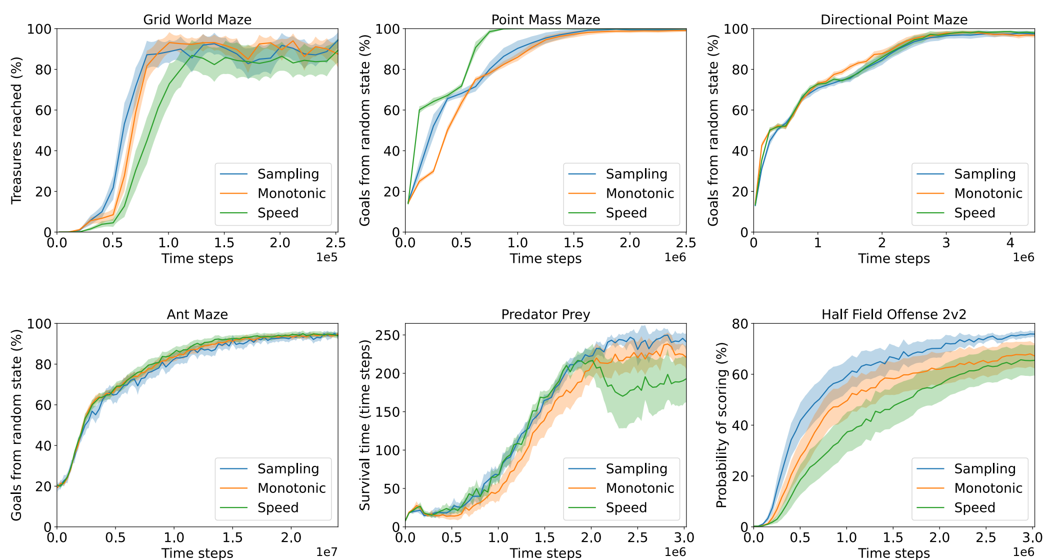

Figure A.1: Performance of the three formulations of the Friction-based progression on six test domains.

A.2 Effects of Multiprocessing

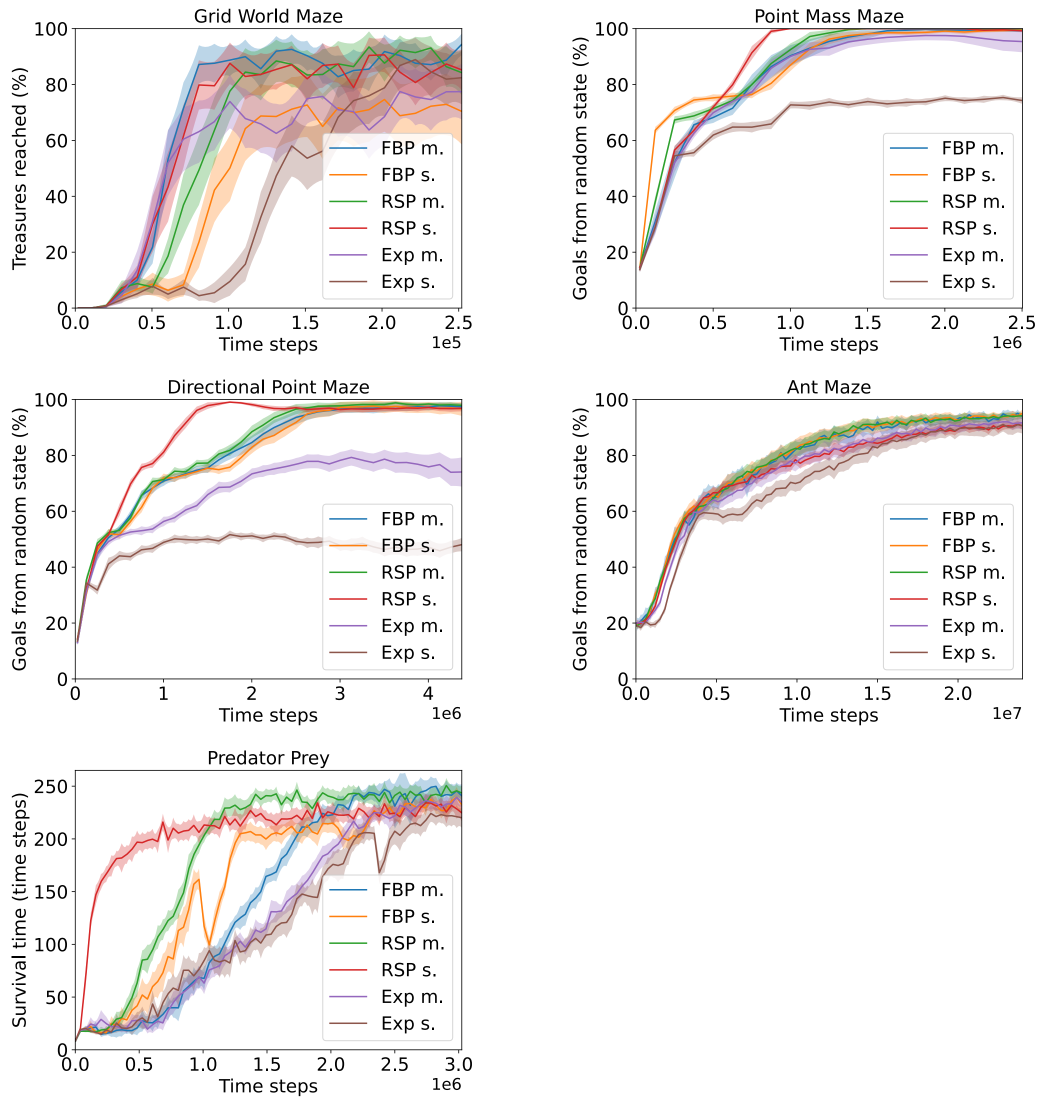

Figure A.2: Performance of the Friction-based progression, Rolling Sphere progression and Exponential progression on five test domains with (m.) and without (s.) multiprocessing.

A.3 Robustness of adaptive progression functions

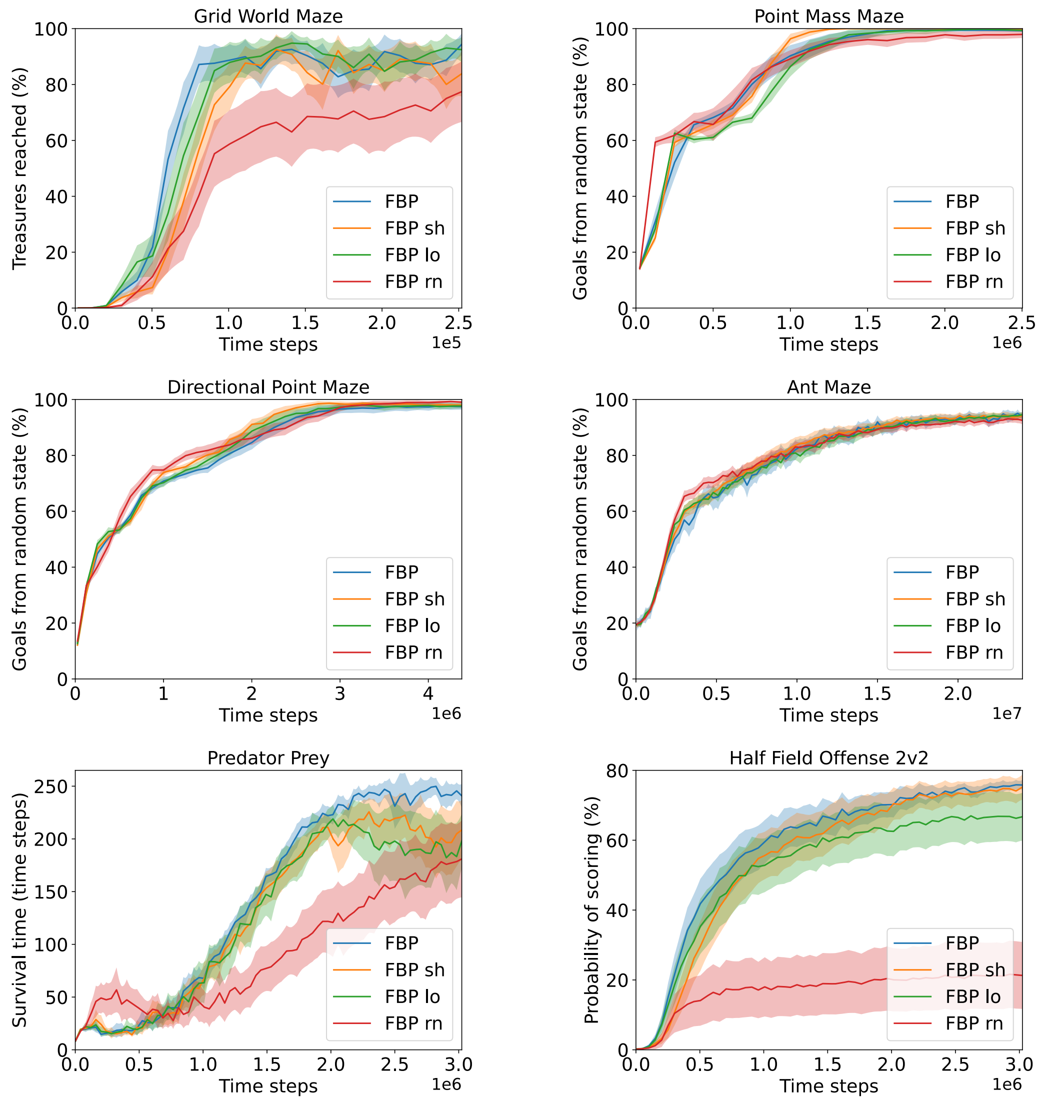

Figure A.3: Performance of the Friction-based progression with local (sh), global (lo) and random (rn) noise on six test domains.

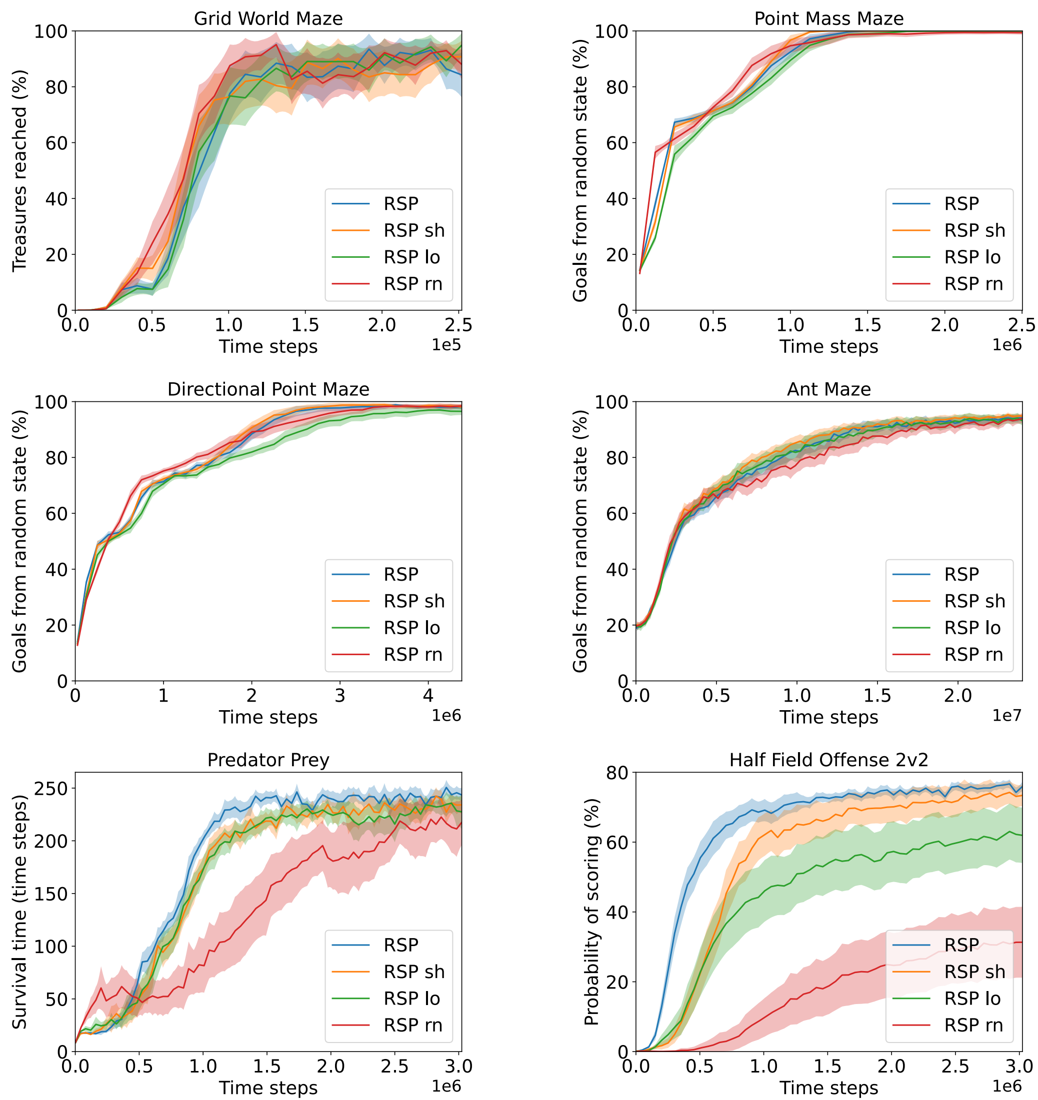

Figure A.4: Performance of the Rolling Sphere progression with local (sh), global (lo) and random (rn) noise on six test domains.

A.4 Adversarial DVB

This section reports the full results for the Adversarial DVB environment. The algorithm on the left is the adversary, whereas the algorithm on the top of the table is the one trying to match the target value. The highest regret, which was the one used for the plots, was highlighted in bold.

	VDTS	VDBESA	SWUCB
VDTS	55.18 ± 2.28	109.57 ± 6.90	99.59 ± 2.81
VDBESA	**64.78 ± 2.87**	**227.29 ± 33.81**	**272.35 ± 9.59**
SWUCB	45.58 ± 1.94	101.63 ± 4.30	80.06 ± 2.38
No adv.	44.21 ± 1.79	91.61 ± 3.06	74.09 ± 2.39

Table A.1: Results for adversarial DVB with $\sigma^2 = 0$

	VDTS	VDBESA	SWUCB
VDTS	58.06 ± 1.74	108.78 ± 3.74	99.39 ± 2.34
VDBESA	**63.98 ± 2.67**	**324.31 ± 35.27**	**268.86 ± 8.72**
SWUCB	48.16 ± 1.49	105.88 ± 3.08	81.06 ± 2.10
No adv.	46.24 ± 1.63	88.74 ± 2.59	75.82 ± 1.89

Table A.2: Results for adversarial DVB with $\sigma^2 = 0.001$

	VDTS	VDBESA	SWUCB
VDTS	113.46 ± 2.26	125.14 ± 4.11	106.51 ± 1.83
VDBESA	**130.44 ± 7.03**	**344.22 ± 40.42**	**295.26 ± 7.85**
SWUCB	106.35 ± 2.57	123.81 ± 4.23	100.33 ± 1.86
No adv.	99.92 ± 2.13	103.80 ± 1.74	91.60 ± 1.65

Table A.3: Results for adversarial DVB with $\sigma^2 = 0.005$

	VDTS	VDBESA	SWUCB
VDTS	192.21 ± 3.77	137.81 ± 2.88	129.30 ± 1.89
VDBESA	**204.51 ± 7.06**	**354.89 ± 38.15**	**322.51 ± 7.39**
SWUCB	195.37 ± 4.09	144.72 ± 2.33	135.89 ± 2.03
No adv.	171.55 ± 3.40	121.17 ± 1.69	117.66 ± 1.57

Table A.4: Results for adversarial DVB with $\sigma^2 = 0.008$

	VDTS	VDBESA	SWUCB
VDTS	256.44 ± 5.04	149.13 ± 1.92	150.16 ± 2.01
VDBESA	**270.57 ± 8.34**	**390.47 ± 37.23**	**323.52 ± 5.61**
SWUCB	267.64 ± 5.19	164.15 ± 2.86	165.49 ± 2.19
No adv.	226.99 ± 3.89	135.58 ± 1.69	136.95 ± 1.67

Table A.5: Results for adversarial DVB with $\sigma^2 = 0.01$

	VDTS	VDBESA	SWUCB
VDTS	538.68 ± 10.22	254.74 ± 2.48	306.59 ± 3.07
VDBESA	519.80 ± 10.22	**443.21 ± 30.07**	**411.58 ± 4.98**
SWUCB	**617.77 ± 9.10**	303.93 ± 4.30	371.21 ± 4.03
No adv.	470.29 ± 5.80	238.26 ± 2.22	284.54 ± 2.98

Table A.6: Results for adversarial DVB with $\sigma^2 = 0.02$

Bibliography

Eugene Allgower and Kurt Georg. Simplicial and continuation methods for approximating fixed points and solutions to systems of equations. *Siam review*, 22(1): 28–85, 1980.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.

Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine learning*, 23(2-3):279–303, 1996.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002a.

Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002b.

Akram Baransi, Odalric-Ambrym Maillard, and Shie Mannor. Sub-sampling for multi-armed bandits. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 115–131. Springer, 2014.

Jonathan Baxter. A bayesian/information theoretic model of learning to learn via multiple task sampling. *Machine learning*, 28(1):7–39, 1997.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

Justin A Boyan and Andrew W Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in neural information processing systems*, pages 369–376, 1995.

Tim Brys, Anna Harutyunyan, Matthew E Taylor, and Ann Nowé. Policy transfer using reward shaping. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 181–188, 2015.

Richard Caruana. Multitask learning: A knowledge-based source of inductive bias. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 41–48. Morgan Kaufmann, 1993.

Jonathan H Connell and Sridhar Mahadevan. Rapid task learning for real robots. In *Robot Learning*, pages 105–139. Springer, 1993.

Felipe Leno Da Silva and Anna Helena Reali Costa. Object-oriented curriculum generation for reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2018.

Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

Jeffrey L Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48(1):71–99, 1993.

Meng Fang, Tianyi Zhou, Yali Du, Lei Han, and Zhengyou Zhang. Curriculum-guided hindsight experience replay. *Advances in Neural Information Processing Systems*, 32:12623–12634, 2019.

Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. *In Proceedings of the 1st Annual Conference on Robot Learning*, 2017.

Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In *International Conference on Machine Learning (ICML)*, 2018.

F Foglino, C Coletto Christakou, R Luna Gutierrez, and M Leonetti. Curriculum learning for cumulative return maximization. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. IJCAI, 2019a.

Francesco Foglino, Christiano Coletto Christakou, and Matteo Leonetti. An optimization framework for task sequencing in curriculum learning. In *2019 Joint IEEE 9th International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*, pages 207–214. IEEE, 2019b.

Aurélien Garivier and Eric Moulines. On upper-confidence bound policies for non-stationary bandit problems. *arXiv preprint arXiv:0805.3415*, 2008.

Liang Ge, Jing Gao, Hung Ngo, Kang Li, and Aidong Zhang. On handling negative transfer and imbalanced distributions in multiple source transfer learning. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 7(4):254–271, 2014.

Fred Glover and Manuel Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998.

David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine Learning*, 3(2):95–99, 1988.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

Alex Graves, Marc G Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *international conference on machine learning*, pages 1311–1320. PMLR, 2017.

Neha Gupta, Ole-Christoffer Granmo, and Ashok Agrawala. Thompson sampling for dynamic multi-armed bandits. In *2011 10th International Conference on Machine Learning and Applications and Workshops*, volume 1, pages 484–489. IEEE, 2011.

Matthew Hausknecht, Prannoy Mupparaju, Sandeep Subramanian, Shivaram Kalyanakrishnan, and Peter Stone. Half field offense: An environment for multiagent learning and ad hoc teamwork. In *AAMAS Adaptive Learning Agents (ALA) Workshop*. sn, 2016.

Elad Hazan and Nimrod Megiddo. Online learning with prior knowledge. In *International Conference on Computational Learning Theory*, pages 499–513. Springer, 2007.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

James M Hyman. Accurate monotonicity preserving cubic interpolation. *SIAM Journal on Scientific and Statistical Computing*, 4(4):645–654, 1983.

Tommi Jaakkola, Satinder Singh, and Michael Jordan. Reinforcement learning algorithm for partially observable markov decision problems. *Advances in neural information processing systems*, 7, 1994.

Lu Jiang, Deyu Meng, Qian Zhao, Shiguang Shan, and Alexander G Hauptmann. Self-paced curriculum learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

Andrej Karpathy and Michiel Van De Panne. Curriculum learning for motor skills. In *Canadian Conference on Artificial Intelligence*, pages 325–330. Springer, 2012.

Faisal Khan, Bilge Mutlu, and Jerry Zhu. How do humans teach: On curriculum learning and teaching dimension. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL `https://proceedings.neurips.cc/paper/2011/file/f9028faec74be6ec9b852b0a542e2f39-Paper.pdf`.

Pascal Klink, Hany Abdulsamad, Boris Belousov, and Jan Peters. Self-paced contextual reinforcement learning. In *Conference on Robot Learning*, pages 513–529, 2020a.

Pascal Klink, Carlo D' Eramo, Jan R Peters, and Joni Pajarinen. Self-paced deep reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9216–9227. Curran Associates, Inc., 2020b. URL `https://proceedings.neurips.cc/paper/2020/file/68a9750337a418a86fe06c1991a1d64c-Paper.pdf`.

Levente Kocsis and Csaba Szepesvári. Discounted ucb. In *2nd PASCAL Challenges Workshop*, volume 2, 2006.

M Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. *Advances in neural information processing systems*, 23:1189–1197, 2010.

Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

Tyler Lu, Dávid Pál, and Martin Pál. Contextual multi-armed bandits. In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics*, pages 485–492. JMLR Workshop and Conference Proceedings, 2010.

Binyamin Manela and Armin Biess. Curriculum learning with hindsight experience replay for sequential object manipulation tasks. *Neural Networks*, 145:260–270, 2022.

Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. Teacher-student curriculum learning. *IEEE transactions on neural networks and learning systems*, 2017.

Bruce D McCandliss, Julie A Fiez, Athanassios Protopapas, Mary Conway, and James L McClelland. Success and failure in teaching the [r]-[l] contrast to japanese adults: Tests of a hebbian model of plasticity and stabilization in spoken language perception. *Cognitive, Affective, & Behavioral Neuroscience*, 2(2):89–108, 2002.

Nicola Milano and Stefano Nolfi. Automated curriculum learning for embodied agents a neuroevolutionary approach. *Scientific reports*, 11(1):1–14, 2021.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous

methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 566–574. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

Sanmit Narvekar, Jivko Sinapov, and Peter Stone. Autonomous task sequencing for customized curriculum design in reinforcement learning. In *(IJCAI), The 2017 International Joint Conference on Artificial Intelligence*, 2017.

Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *arXiv preprint arXiv:2003.04960*, 2020.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287, 1999.

Peng Si Ow and Thomas E Morton. Filtered beam search in scheduling. *The International Journal Of Production Research*, 26(1):35–62, 1988.

Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

Christos H Papadimitriou and John N Tsitsiklis. The complexity of markov decision processes. *Mathematics of operations research*, 12(3):441–450, 1987.

Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. In *International Conference on Machine Learning*, pages 2817–2826. PMLR, 2017.

Sebastien Racaniere, Andrew K Lampinen, Adam Santoro, David P Reichert, Vlad Firoiu, and Timothy P Lillicrap. Automated curricula through setter-solver interactions. *arXiv preprint arXiv:1909.12892*, 2019.

Zhipeng Ren, Daoyi Dong, Huaxiong Li, and Chunlin Chen. Self-paced prioritized curriculum learning with coverage penalty in deep reinforcement learning. *IEEE transactions on neural networks and learning systems*, 29(6):2216–2226, 2018.

Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom van de Wiele, Vlad Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing solving sparse reward tasks from scratch. In *Proceedings of the 35th International Conference on Machine Learning*, pages 4344–4353, 2018.

Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

Manuel S Santos and John Rust. Convergence properties of policy iteration. *SIAM Journal on Control and Optimization*, 42(6):2094–2115, 2004.

Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International conference on machine learning*, pages 1312–1320. PMLR, 2015.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2016.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Alexander A Sherstov and Peter Stone. Function approximation via tile coding: Automating parameter choice. In *International Symposium on Abstraction, Reformulation, and Approximation*, pages 194–205. Springer, 2005.

Felipe Leno Da Silva and Anna Helena Reali Costa. Object-oriented curriculum generation for reinforcement learning. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1026–1034. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

David Silver. Lecture 2: Markov decision processes. *UCL. Retrieved from www0. cs. ucl. ac. uk/staff/d. silver/web/Teaching_files/MDP. pdf*, 2015.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Burrhus Frederic Skinner. How to teach animals. *Scientific American*, 185(6):26–29, 1951.

Aleksandrs Slivkins. Introduction to multi-armed bandits. *arXiv preprint arXiv:1904.07272*, 2019.

Aleksandrs Slivkins and Eli Upfal. Adapting to a changing environment: the brownian restless bandits. In *COLT*, pages 343–354, 2008.

Petru Soviany, Radu Tudor Ionescu, Paolo Rota, and Nicu Sebe. Curriculum learning: A survey. *arXiv preprint arXiv:2101.10382*, 2021.

Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. *arXiv preprint arXiv:1703.05407*, 2017.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Maxwell Svetlik, Matteo Leonetti, Jivko Sinapov, Rishi Shah, Nick Walker, and Peter Stone. Automatic curriculum graph generation for reinforcement learning agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.

Matthew E Taylor, Shimon Whiteson, and Peter Stone. Transfer via inter-task mappings in policy search reinforcement learning. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 37. ACM, 2007.

Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

John N Tsitsiklis and Benjamin Van Roy. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081, 1997.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

Lev Semenovich Vygotsky. *Mind in society: The development of higher psychological processes*. Harvard university press, 1980.

Xin Wang, Yudong Chen, and Wenwu Zhu. A survey on curriculum learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

Peter Whittle. Restless bandits: Activity allocation in a changing world. *Journal of applied probability*, 25(A):287–298, 1988.

Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation, 2017.