Learning and Planning in Videogames via Task Decomposition

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

Michael Dann
Master of Computer Science
RMIT University, Australia

Bachelor of Science
The University of Melbourne, Australia

School of Science
College of Science, Engineering and Health
RMIT University

May 2019
Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship. I also acknowledge the support of the Defence Science Institute for the Research Higher Degree Student Grant that I received for the duration of my candidature.

Michael Dann
School of Science
RMIT University
May 26, 2019
Acknowledgments

First of all I would like to thank my supervisors, Associate Professor Fabio Zambetta and Professor John Thangarajah, who actively encouraged me to follow my research ambitions from the outset, and who gave me the freedom and confidence I needed to develop my own ideas. Throughout the highs and lows of the peer review process, they always managed to help me find perspective, and taught me a great deal about my inner thought processes along the way. I am deeply grateful for their interest in my professional development post-PhD, and for assisting financially by providing additional paid work and helping me apply for the Defence Science Institute grant.

Another person I owe a huge debt to is my very understanding wife Oanh, whom my courtship of almost derailed my candidature in the early days, but whose love and support has kept me on track ever since. She never mumbled about all the times I got up in the middle of the night to check on experiments, and valiantly handled the stress of organising the wedding while I was struggling to hit a paper deadline. We will go on a proper honeymoon soon, I swear!

Thanks to all of my friends for being understanding about my lack of free time, and for providing much-needed stress relief over videogames, bike rides, coffees, bad TV Sunday evenings and curry/beer Fridays.

Finally, I owe everything to my parents, who have shown me nothing but unconditional love and support since the day I was born. At the most challenging times along the journey, I was spurred on by the knowledge that they would still have my back even if I tried and failed.
Credits

Portions of the material in this thesis have previously appeared in the following publications:


The thesis was written in the TeXstudio editor on Ubuntu 16.04, and typeset using the \LaTeX{} document preparation system.

All trademarks are the property of their respective owners.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>11</td>
</tr>
<tr>
<td>2.1 Markov Decision Processes (MDPs)</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Reinforcement Learning (RL)</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 Tabular Action-Value Methods</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2 Deep Reinforcement Learning via DQN</td>
<td>17</td>
</tr>
<tr>
<td>2.2.3 The Deep Deterministic Policy Gradient (DDPG) Algorithm</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Addressing Reward Sparsity</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 Multi-Step Bootstrapping</td>
<td>23</td>
</tr>
<tr>
<td>2.3.2 Intrinsic Reward Schemes</td>
<td>24</td>
</tr>
<tr>
<td>2.3.3 Potential-Based Reward Shaping</td>
<td>26</td>
</tr>
<tr>
<td>2.4 Hierarchical Reinforcement Learning</td>
<td>27</td>
</tr>
<tr>
<td>2.4.1 The Options Framework</td>
<td>28</td>
</tr>
<tr>
<td>2.4.2 Autonomous Skill Acquisition</td>
<td>29</td>
</tr>
<tr>
<td>2.4.3 Acquiring Flexible Skills</td>
<td>30</td>
</tr>
<tr>
<td><strong>3 Deriving Subgoals Autonomously to Accelerate Learning in Sparse</strong></td>
<td>33</td>
</tr>
<tr>
<td><strong>Reward Domains</strong></td>
<td></td>
</tr>
<tr>
<td>3.1 Issues with Existing Intrinsic Motivation Schemes</td>
<td>35</td>
</tr>
<tr>
<td>3.1.1 Count-Based Novelty Methods</td>
<td>35</td>
</tr>
<tr>
<td>3.1.2 Prediction Error Methods</td>
<td>37</td>
</tr>
<tr>
<td>3.1.3 Visual Novelty: Strengths and Weaknesses</td>
<td>38</td>
</tr>
<tr>
<td>3.1.4 The “Risky Exploration Problem”</td>
<td>40</td>
</tr>
<tr>
<td>3.2 Our Approach</td>
<td>41</td>
</tr>
<tr>
<td>3.2.1 State Space Partitioning via a Measure</td>
<td>41</td>
</tr>
<tr>
<td>3.2.2 Exploration Effort (EE)</td>
<td>42</td>
</tr>
<tr>
<td>3.2.3 Pellet Rewards</td>
<td>46</td>
</tr>
<tr>
<td>3.3 Experiments</td>
<td>47</td>
</tr>
</tbody>
</table>
3.3.1 Configuration .................................................. 47
3.3.2 Results ........................................................... 49
3.3.3 Efficiency Compared to Existing Subgoal-Based Methods ...... 53
3.3.4 Comparison Against Non-Subgoal-Learning Agents ........... 55
3.4 Further Related Work ........................................... 56
3.5 Chapter Summary .................................................. 57
3.6 Algorithm Pseudocode ............................................ 57

4 Synoptic Vision Planning (SVP) .................................. 61
4.1 Synoptic Vision Planning (SVP) ................................ 63
  4.1.1 Assumptions .................................................. 65
  4.1.2 Training a Local Movement Skill ........................... 66
  4.1.3 Learning a High-Level Model .............................. 69
  4.1.4 Planning via Synoptic Vision .............................. 69
  4.1.5 Plan Execution and Replanning ............................ 71
  4.1.6 Reward Schemes for Training Local Movement ............ 73
  4.1.7 Encouraging Survival ...................................... 75
4.2 Experimental Configuration ..................................... 76
  4.2.1 Generating Maze-Like Navigation Tasks .................... 76
  4.2.2 Agent Configuration ....................................... 77
  4.2.3 Benchmark Agents ........................................ 79
4.3 Results and Discussion .......................................... 80
  4.3.1 Maze-Like Navigation Without Enemies .................... 80
  4.3.2 Maze-Like Navigation With Enemies ....................... 83
4.4 Related Work ..................................................... 85
4.5 Chapter Summary .................................................. 88

5 Exploration in Continuous Control Tasks via Continually Parameterised Skills ............................................. 89
5.1 Hierarchical Control via Policy Composition ................. 92
  5.1.1 Layer Organisation ......................................... 92
  5.1.2 Training the Hierarchy .................................... 93
  5.1.3 Exploring Efficiently via Continually Parameterised Skills 94
5.2 The Rocketship Domain .......................................... 95
  5.2.1 Goal-Seek Task ............................................ 96
  5.2.2 Keep-Alive Task ........................................... 97
  5.2.3 Level Configuration ....................................... 97
5.3 Applying Our Framework to Rocketship ....................... 98
  5.3.1 Skill Hierarchy ............................................ 98
  5.3.2 Training the Velocity Skill ................................ 98
List of Figures

1.1 The first screen from the Atari 2600 game, Montezuma’s Revenge. ........... 5
1.2 A schematic illustration of high-level planning via “god’s eye vision”. ........... 6

2.1 An example of pre- and post-reward states being very similar pixel-wise. .... 19

3.1 A figurative illustration of the pellet rewards mechanism. ......................... 35
3.2 Time lapses from the first screens of (a) Montezuma’s Revenge and (b) Venture, generated over 10,000 frames. The initial states for both games are provided to the left for reference. The protagonist in Venture is highlighted by the red circle. ......................................................... 39
3.3 Training scores on sparse reward games (based on a full set of lives, averaged over the last 100 episodes). Thick lines indicate average scores, thin lines represent individual training runs. ......................................................... 50
3.4 Representative states found over a training run on Montezuma’s Revenge. .... 51
3.5 Training scores on dense reward games. .................................................. 52

4.1 In platform videogames, maze-like navigation tasks commonly arise in the context of reaching a powerup, such as a “fire flower” in Infinite Mario. ........... 62
4.2 A schematic example of how SVP forms temporally abstract plans. A global plan for reaching the fire flower is projected as a sequence of local translations. 64
4.3 Under a Manhattan distance metric, $N_t^D$ corresponds to a square grid of tiles surrounding the current tile, $t$. At the start of each training episode, a random goal tile $t_g$ is assigned from within $N_t^D$. ......................................................... 67
4.4 An example from Infinite Mario where the agent will become stuck trying to reach the next waypoint, $t_w$. ......................................................... 72
4.5 The “two stage” movement in subfigure (a) can be decomposed into the two basis movements shown in subfigure (b). ......................................................... 74
4.6 An illustration of the goal placement zones used in our experiments. Note that the level structure shown here is just one sample possibility. ......................... 76
4.7 Success rate versus time taken for the basis and binary versions of SVP and the A* benchmark. ......................................................... 81
4.8 Alternative plans for the same task found by (a) the basis agent and (b) the binary agent. The goal is highlighted green, waypoints are highlighted yellow.  

4.9 The basis agent’s success rate versus time taken for different values of the replanning threshold, $k$.  

5.1 The Rocketship domain. Subfigure (a) shows how asteroid velocities are represented via radial and tangential components. Subfigure (b) shows one random spawn point in the goal-seek task.  

5.2 Average return achieved from the initial state of the goal-seek task, averaged over five runs.  

5.2 Average return achieved from the initial state of the goal-seek task, averaged over five runs.  

5.3 The average return, success rate and crash rate from a single run of the hierarchical agent on Level C. The agent began strongly manipulating the rocket’s speed at around $10^8$ frames.  

5.4 Crash rates versus training frames on the keep-alive task. Curves are averaged over five training runs.  

5.4 Crash rates versus training frames on the keep-alive task. Curves are averaged over five training runs.  

5.5 Primitive action distributions for the hierarchical and flat agents when biased to turn to the right.
List of Tables

4.1 The training parameters used for Infinite Mario. 78
4.2 Results compared across the three goal placement zones. Error bounds represent one standard deviation. 83
4.3 The performance of the hybrid agent versus the parent methods from which it was derived. 84
5.1 The asteroid settings for each level. 98
5.2 The settings used in our experiments. 100
5.3 Pre-training success and crash rates on Level A. 101
5.4 Crash rates prior to learning on Level D. 104
Abstract

Artificial intelligence (AI) methods have come a long way in tabletop games, with computer programs having now surpassed human experts in the challenging games of chess, Go and heads-up no-limit Texas hold’em. However, a significant simplifying factor in these games is that individual decisions have a relatively large impact on the state of the game. The real world, however, is granular. Human beings are continually presented with new information and are faced with making a multitude of tiny decisions every second. Viewed in these terms, feedback is often sparse, meaning that it only arrives after one has made a great number of decisions. Moreover, in many real-world problems there is a continuous range of actions to choose from, and attaining meaningful feedback from the environment often requires a strong degree of action coordination. Videogames, in which players must likewise contend with granular time scales and continuous action spaces, are in this sense a better proxy for real-world problems, and have thus become regarded by many as the new frontier in games AI.

Seemingly, the way in which human players approach granular decision-making in videogames is by decomposing complex tasks into high-level subproblems, thereby allowing them to focus on the “big picture”. For example, in Super Mario World, human players seem to look ahead in extended steps, such as climbing a vine or jumping over a pit, rather than planning one frame at a time. Currently though, this type of reasoning does not come easily to machines, leaving many open research problems related to task decomposition. This thesis focuses on three such problems in particular: (1) The challenge of learning subgoals autonomously, so as to lessen the issue of sparse feedback. (2) The challenge of combining discrete planning techniques with extended actions whose durations and effects on the environment are uncertain. (3) The questions of when and why it is beneficial to reason over high-level continuous control variables, such as the velocity of a player-controlled ship, rather than over the most low-level actions available. We address these problems via new algorithms and novel experimental design, demonstrating empirically that our algorithms are more efficient than strong baselines that do not leverage task decomposition, and yielding insight into the types of environment where task decomposition is likely to be beneficial.
CHAPTER 1

Introduction

In the field of Artificial Intelligence (AI), there is a long tradition of using games as a benchmark for comparing machine intelligence to human intelligence. Two of the biggest watershed moments in AI history are the 1997 victory of chess computer Deep Blue [Campbell et al. 2002] over Garry Kasparov, who was then reigning world champion, and the recent victories of AlphaGo [Silver et al. 2016] over the top human Go players Lee Sedol and Ke Jie. Games are a natural environment for AI research because they are designed to be challenging for humans, often testing multiple facets of intelligence simultaneously, such as learning, planning, perception and memory recall. At the same time, it is usually both cheaper and faster to experiment in simulated games than it is to deploy a physical agent into the real world.

As AI techniques have improved, they have been applied to ever more complex games. At the time of writing, the current frontier in games AI research is arguably videogames. Some of the world’s largest companies are now conducting research in this domain, including Google [Mnih et al. 2015], Facebook [Tian et al. 2017] and Microsoft [Van Seijen et al. 2017]. One reason for this interest is that, as a multi-billion dollar industry, videogames are an important domain in their own right. Developers are increasingly eager to incorporate intelligent agents into their games, such as automated game-testing agents, automated opponents for strategy games, and non-player characters that respond realistically to the player’s actions. Beyond this intrinsic interest, however, there is a more fundamental reason why videogames research is important. Historically, a major strength of computer-based agents has been their ability to perform a vast number of low-level calculations quickly. In videogames though, where players must respond to high-dimensional sensory input in real-time, often via continuous control devices such as mice and joysticks, the kind of atomic, brute force computation that computers excel at is not sufficiently scalable. In short, videogames require the player to see beyond the level of individual pixels and frames, and recognise the “big picture”. Since many large, real-world problems also
exhibit this property, videogames have rightly become regarded as more than just a fun application domain. Broadly speaking, the aim of this thesis is to develop scalable learning and planning methods for videogames, in the hope that many of the insights gained may carry over to complex real-world problems.

Recently, the rise of deep learning [LeCun et al. 2015] has seen significant progress on one aspect of this problem; namely, it has enabled machines to become much better at identifying high-level patterns in low-level input. As such, deep learning algorithms have proven particularly successful on tasks such as object recognition [Krizhevsky et al. 2012] and speech recognition [Hinton et al. 2012]. More pertinent to us, deep learning has also enabled rapid advances in games AI. To give a prominent example, one of the main historical challenges in developing strong agents for Go was accurately evaluating the board, i.e. determining which side is winning, and by how much. Until recently, most artificial Go agents relied upon expert-crafted representations to perform this task, and still calculated far less accurate evaluations than human experts. However, the recent program AlphaZero [Silver et al. 2017; 2018] – a successor to AlphaGo – learned to evaluate the board accurately from only a raw capture of the stones’ positions, using deep learning. Deep learning’s strength in handling low-level input has also been showcased in videogames, most notably by the well known Deep Q-Network (DQN) agent of Mnih et al. [2015], which learned to play many Atari 2600 games competently from only a pixel-based representation of the screen and a feed of the game score.

This recent progress notwithstanding, there remain other forms of granularity arising in videogames that modern AI techniques continue to struggle with. In particular, artificial agents often struggle when faced with a vast number of granular decisions. In this thesis, we consider two common ways in which this can manifest: via granularity in the time scale and via granularity in the action space. To be clear, what we mean by “granularity in the time scale” is that most videogames run at no less than 30 frames per second, so that each decision made by the agent persists for only a fraction of a second. By “granularity in the action space”, we refer to games that take continuous input devices.\footnote{Such games technically have granular, rather than continuous, action spaces, since computer variables are never truly continuous at the storage level.}

The challenge imposed by time scale granularity on learning agents is exemplified by the poor performance of Mnih et al.’s [2015] DQN agent in the game Montezuma’s Revenge. On the first screen of the game, the player must acquire a key in order to gain access to the remainder of the dungeon (see Figure 1.1). Acquiring the key is also the first scoring opportunity in the game and hence the first time that the player receives any explicit feedback from the environment. To a human player, it might appear that the key is not located all that far from the start location, since it takes less than 10 seconds to reach the key with perfect play. However, due to time scale granularity, even the fastest trajectory to the key spans hundreds of individual time steps. Furthermore, at several
points along the journey, the agent will die if it presses the wrong button. Therefore, a
learning-based agent that has not yet gleaned any information about the game’s objective
is very unlikely to fluke its way to the key. At the same time, the only way such an agent
can learn about the game’s objective is by reaching the key. In other words, the agent is
caught in a dependency loop.

Time scale granularity is also problematic for planning-based agents, i.e. agents that
calculate their next action by projecting future events. Due to videogames’ granular time
scales, planning agents must project hundreds of time steps into the future in order to
achieve search depths of more than just a few seconds. Moreover, if such agents are
required to respond in real-time, the time budget allowed to calculate each action will be
very limited. This challenge is even greater in games where feedback is sparse, such as
Montezuma’s Revenge, since there is limited information available to guide the direction
of the search. As we shall see, this is particularly problematic in maze-like navigation
tasks, since acting according to a heuristic progress measure will often lead the agent to
a dead-end.

Finally, in domains with continuous action spaces, the type of issues described above
are only compounded. Aside from the complexity of there being a limitless number of
actions to choose from, issues arising from sparse feedback may be amplified by the preva-
ience of useless or wasteful actions. To give an extreme example, imagine a modified
version of Montezuma’s Revenge where the protagonist is replaced by a simulated hu-
manoid robot, such that the action space consists of the set of possible torques on the
robot’s joints. In this setting, the vast majority of actions will fail to achieve any kind of
meaningful motion, making the chances of discovering the key even more remote.

The central motif explored in this thesis is that all of the above issues appear to call
for some type of task decomposition. For example, in sparse feedback tasks, it may be
possible to break the task of reaching some distant goal into subgoals. Then, even if a
particular attempt at a task yields no direct environmental feedback, the agent can infer
how much progress it made based on how many subgoals were attained. In a similar vein,
CHAPTER 1: INTRODUCTION

Figure 1.2: A schematic illustration of high-level planning via “god’s eye vision”.

videogame planning problems that are long in terms of the number of frames needed to complete them may be simplified by identifying high-level waypoints and looking ahead in temporally extended steps. For humans, it is relatively straightforward to plan like this in classical 2D platform videogames, such as Super Mario Bros., where one can leverage the “god’s eye vision” provided to visualise plans in terms of local translations (Figure 1.2). Finally, in the extreme example we gave of changing the protagonist in Montezuma’s Revenge to a humanoid robot, it is clear that the task of key discovery would be easier if the agent was first taught the subtask of learning how to walk.

Unfortunately, despite the intuitiveness of the above ideas, the ability to identify and exploit substructure in tasks does not come easily to artificial agents. For one, there is the challenge of autonomous subgoal identification, which has been studied for decades without reaching consensus on the best approach [Vezhnevets et al. 2017, Thrun and Schwartz 1995, Diney 1998, Şimşek et al. 2005, Konidaris and Barto 2009, Machado et al. 2017; 2018b]. This challenge is particularly difficult in domains, such as videogames, where the state of the environment is conveyed visually via raw pixels. Secondly, the type of high-level planning sketched in Figure 1.2 is deceptively difficult for machines to perform. The main reasons for this are that the exact run time of each step is uncertain, as is the exact state that the agent will find itself in after each step. Finally, while it is intuitively clear that continuous skills, such as walking, are needed in very complex control problems, acting over high-level control variables may entail a loss of fine-grained control, and it is not clear at which point this trade-off becomes worthwhile, nor whether acting over continuous skills offers any benefit beyond a greater chance of discovering distant rewards. Over the course of this thesis, we seek to address and understand these key issues. After listing our research questions below, we provide a slightly more in-depth summary of the problems described and the novel contributions that we make to address them.
Research Questions

To summarise, we wish to address time scale and action space granularity in videogames in the hope that our research may carry some insight into real-world control problems. We are particularly interested in tasks where feedback is sparse, as these pose a particular challenge in the face of such granularity. Our specific research questions are as follows:

**R1** In videogames with sparse rewards, what learning methods can be developed to identify subgoals autonomously and improve exploration efficiency?

**R2** Given the granular time scale in videogames, what planning techniques can be developed to leverage “god’s eye vision” and achieve a lookahead depth comparable with that of humans?

**R3** In sparse reward videogames with continuous actions, what advantages does acting hierarchically over continually parameterised skills offer over ordinary learning?

**R1**

Sparse reward videogames, such as the infamous *Montezuma’s Revenge*, pose a significant challenge for agents based on reinforcement learning [Sutton and Barto 1998]. It has long been thought that hierarchical reinforcement learning [Parr and Russell 1998, Sutton et al. 1999, Dietterich 2000], which decomposes drawn-out tasks into subtasks, may be key to exploring efficiently in such domains. In fact, it has already been established that hierarchical agents can make rapid progress in sparse reward *Atari* games, so long as they are provided with appropriate subgoals by a human expert [Kulkarni et al. 2016]. However, a major deficiency of such agents is that they do not address subgoal identification, which is a very challenging part of the problem.

In Chapter 3, we propose a new, autonomous approach for deriving subgoals that is applicable to raw visual input and is more efficient than competing methods. We propose a novel intrinsic reward scheme [Bellemare et al. 2016] for exploiting the derived subgoals, and apply it to three *Atari* games with sparse rewards to test whether it yields more efficient exploration. Our agent achieves performance comparable with strong baselines derived from density models [Bellemare et al. 2016, Ostrovski et al. 2017]. To the best of our knowledge, this makes ours the first fully-autonomous, subgoal-oriented agent to reach competitive performance on these games. In addition, our intrinsic reward scheme appears to have relatively little distortion on the native task objective, causing no discernible harm in two games where another prominent approach [Ostrovski et al. 2017] was detrimental.
In platform videogames, players are frequently tasked with solving medium-term navigation problems in order to gather items or powerups. Artificial agents must generally obtain some form of direct experience before they can solve such tasks. Learning agents require multiple training runs to gather this experience, which makes them ill-suited to solving new tasks on their first attempt. Planning agents, on the other hand, leverage a predictive model of the environment to generate simulated experience. The strongest planning agents for videogames typically exploit perfect predictive models, derived directly from the game’s code. However, they struggle to formulate long-term plans quickly, due to time scale granularity. Human players do not possess exact models, yet appear capable of planning ahead in high-level, temporally extended steps.

In Chapter 4, we hypothesise that the “god’s eye” view provided in classical 2D platform games makes this type of planning significantly easier. Motivated by this observation, we propose a planning approach for platform videogames that seeks to bridge the gap between temporally extended actions, which are uncertain in outcome and duration, and a traditional discrete search algorithm. We apply this approach to randomly generated, maze-like navigation problems in Infinite Mario [Togelius et al. 2010]. After an initial training phase where our agent learns transferable skills [Konidaris and Barto 2007] and high-level knowledge about the game, it is capable of solving new navigation tasks without further training. Moreover, our agent scales better with goal distance compared to a streamlined low-level search agent that exploits an exact model of the game’s physics.

Reinforcement learning agents for tasks with continuous action spaces often require a steady, informative reward signal to learn effectively. In videogames, however, certain tasks are much easier to specify via a sparse reward. For example, in a game involving moving obstacles, it is much easier to prescribe a negative reward for colliding with an obstacle than it is to prescribe a continual reward that guides the agent along the optimal path in all situations. However, in continuous action domains, it can be difficult to learn without frequent rewards, especially if effecting meaningful change in the environment requires a high degree of action coordination (as illustrated by our earlier example of a humanoid robot exploring the dungeon in Montezuma’s Revenge).

Intuitively, a sensible approach to this issue is to first teach the agent one or more parameterised skills [da Silva et al. 2012; 2014b;a], such as the skill of being able to walk in direction \( d \) (where \( d \) is the so-called parameter), before tackling the main task itself. To date though, we are not aware of any empirical work that compares this type of hierarchical learning versus ordinary, non-hierarchical learning. Existing work largely focuses on simplified problems, where there is clearly no need for hierarchy, or on tasks so
complex that the need for hierarchy is obvious and the low-level layer is either hardcoded or abstracted away. Therefore, it is unclear whether the only advantage offered by the hierarchical approach is that it makes reward discovery easier, or whether it yields further benefits. In Chapter 5, we investigate this question by performing a case study in a domain that lies in the grey area, i.e. where the need for hierarchy or otherwise is non-obvious. We describe a framework for learning hierarchically over parameterised skills and compare this approach against ordinary, non-hierarchical learning. Interestingly, even on tasks where the non-hierarchical agent discovers enough feedback to make significant progress, it is outperformed in the long-term by the hierarchical agent. Additional analysis suggests that the hierarchical agent’s advantage lies in exploring a wider range of task-relevant actions, even when the variance in the agents’ action spreads is identical.

**Thesis Structure**

In Chapter 2, we cover the background concepts necessary to understand our contributions. Note that related work is not covered here unless it is strictly required background; the majority of related work is covered as it arises on a per-chapter basis. Over the following Chapters 3, 4 and 5, we explain our core contributions, which correspond chronologically to our three research questions. Finally, in Chapter 6, we summarise our findings and discuss potential avenues for future work.
Taking a high-level view, all of the research questions introduced in the previous chapter concern an agent acting in an environment. In videogames research, this interaction is commonly modelled as a Markov Decision Process (MDP) [Sutton and Barto 1998]. At each time step, the agent receives information about the state of the environment, based on which it selects an action. The state of the environment then evolves according to the action chosen, and the agent receives a reward (which may be positive, zero, or negative to reflect punishment). In a videogame, the state of the environment might be conveyed in terms of the raw pixels on the screen, or alternatively via high-level details such as the player’s current health and the locations of enemies. An action might consist of a button combination on a gamepad. The agent might be rewarded for increasing the game’s score, or for making progress through a level. Whatever the specific scenario, the agent’s aim in an MDP is to maximise its return, or the sum of its future rewards.\(^1\) These concepts, which underpin all of the problems studied in this thesis, are introduced formally in Section 2.1.

The artificial intelligence techniques most applicable to a game depend on how much information the agent is assumed to know from the outset. For example, in some games, the players are provided with a predictive model of the environment. In chess, this is merely equivalent to being told the rules: Given a state (the current board) and an action (the next move), the rules dictate exactly what the next state will look like. In such games, a natural approach is to deploy a planning algorithm, i.e. a method that calculates the next move by using the model to look into the future. In videogames, however, human players are not usually privy to an exact model. While it is possible to derive an exact model artificially by exploiting access to an emulator, we are more interested in understanding intelligence than achieving high-scores, so we seek to avoid emulator exploits in this thesis and stipulate that the agent should either learn a model from experience, or not use a model at all.

\(^1\)As we explain shortly, this sum is usually weighted so that the agent prefers to receive rewards quickly.
CHAPTER 2: BACKGROUND

The setting just described, where the environment model is initially unknown, is the domain of Reinforcement Learning (RL) [Sutton and Barto 1998]. The aim in reinforcement learning is to learn a rule for choosing actions, known as a policy, based on past interactions. Typically, this is achieved by exploring in the early learning stages, then gradually exploiting actions that have proven successful in the past. To give a simple example, an RL agent for Super Mario Bros. might begin by pressing random button combinations. Then, over time, it might observe that pressing the “jump” button when Mario is beneath a coin generally leads to higher scores, while pressing the “left” button when there is an enemy immediately to Mario’s left leads to lower scores. By adjusting its behaviour accordingly, while still pressing random buttons occasionally so that it can continue to discover improvements, the agent may eventually learn a strong policy.

In Section 2.2, we cover the background needed to understand the RL algorithms used in this thesis. We begin by explaining some basic methods that have strong theoretical guarantees in the tabular case, where the number of states and actions is small enough that the agent can store all necessary information in a table (Section 2.2.1). Next, we explain some issues that arise in naively applying these algorithms to more complex problems, such as videogame learning from raw pixels. We introduce the DQN algorithm [Mnih et al. 2015], whose success in handling these issues in the Atari domain pioneered the field of deep reinforcement learning (Section 2.2.2). Finally, we cover the DDPG algorithm [Lillicrap et al. 2016], which can be viewed as a analog of DQN for tasks with continuous action spaces, such as videogames involving control stick input (Section 2.2.3).

As explained in Chapter 1, this thesis is primarily concerned with tasks where current reinforcement learning algorithms tend to struggle, with a particular emphasis on sparse reward problems [Ostrovski et al. 2017]. In Section 2.3, we describe some existing techniques for mitigating reward sparsity. First, we explain multi-step bootstrapping, which aims to improve the agent’s learning speed by propagating signals from distant rewards faster (Section 2.3.1). Next, we consider the natural idea of modifying the reward scheme to make it less sparse. We cover intrinsic reward schemes, which motivate the agent to generate novel or surprising experience (Section 2.3.2), and potential-based reward shaping, which incorporates prior knowledge into the reward scheme in a clever manner that leaves the optimal solution to the task unchanged (Section 2.3.3).

We conclude the chapter by discussing hierarchical reinforcement learning (Section 2.4), an approach that likewise holds promise in addressing reward sparsity. A key idea in hierarchical RL is that it may be easier for an agent to learn a complex task if it has previously learned a collection of related subtasks. This is a powerful concept that all of our contributions leverage in some way. However, hierarchical RL introduces its own questions, such as how to identify subgoals autonomously, and how best to represent learned subtask knowledge in order to maximise its reusability. After overviewing a popular hierarchical RL framework in Section 2.4.1, we discuss these issues in detail over Sections 2.4.2 – 2.4.3.
2.1 Markov Decision Processes (MDPs)

In keeping with the majority of videogames research, we assume that all decision-making problems studied in this thesis satisfy the Markov property. That is, the probability of observing state $s'$ and receiving reward $r$ at time $t+1$ depends only on the state and action chosen at time $t$:

$$
\Pr(s_{t+1} = s', r_{t+1} = r \mid s_0, a_0, r_1, \ldots, s_{t-1}, a_{t-1}, r_t, s_t, a_t) = \Pr(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t) \tag{2.1}
$$

Put another way, this property implies that the current state representation captures all information from the episode history that is relevant to decision-making. Decision-making processes that satisfy this property are known as Markov Decision Processes (MDPs). Formally, an MDP is defined by a tuple $(S, A, \mathcal{P}_a, \mathcal{R}_a, \gamma)$ where:

- $S$ is the state space, the set of all possible states.
- $A$ is the action space.
- $\mathcal{P}_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the state transition function, which gives the probability of transitioning to state $s'$ after executing action $a$ in state $s$.
- $\mathcal{R}_a(s, s')$ is the reward function, which specifies the immediate reward that the agent receives after executing action $a$ and transitioning from state $s$ to $s'$.
- $\gamma \in [0, 1]$ is the discount factor, which determines the extent to which the agent should prefer near-term rewards over distant ones.

An MDP may be episodic, meaning that the task ends when a terminal state is reached, or continuing, meaning that the agent continues to interact with the environment indefinitely. The return obtained by the agent, denoted by $R_t$, is defined as the discounted sum of all future rewards:

$$
R_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} \quad \text{[episodic task]} \tag{2.2}
$$

$$
R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad \text{[continuing task]} \tag{2.3}
$$

For continuing tasks, the discount factor must be strictly less than 1 for the infinite series in Equation 2.3 to converge. The sharper the discount (i.e. the closer $\gamma$ is to 0), the more the agent is compelled to obtain rewards quickly.

---

2 In videogames, raw pixel representations based on single screen captures often violate this property, since they fail to capture moving objects' velocities. For this reason, pixel-based agents usually receive a stack of the last few screens combined. However, even this representation may omit some important details, e.g. whether or not an item was collected much earlier in the episode. Regardless, it is common practice to ignore this problem, with the justification that most videogames remain “predominantly Markov”.

(May 26, 2019)
The main aim in an MDP is to find a policy, or a rule for choosing actions, that maximises the agent’s expected return. Formally, a policy is a function $\pi : S \times A \to [0, 1]$ where $\pi(s, a)$ gives the probability of the agent choosing action $a$ upon encountering state $s$. The “strength” of a policy can be gauged from its value function $V^\pi : S \to \mathbb{R}$, where $V^\pi(s)$ is defined as the return expected from state $s$ if the agent follows policy $\pi$ at all future time steps:

$$V^\pi(s) = \mathbb{E}_\pi[R_t \mid s_t = s]$$

(2.4)

A closely related function, which plays an important role in many learning algorithms, is the action-value function $Q^\pi : S \times A \to \mathbb{R}$. This function gives the return expected from state $s$ if the agent first takes action $a$, then follows policy $\pi$ from that point onward:

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_t \mid s_t = s, a_t = a]$$

(2.5)

Value functions can be used to compare policies. In particular, they induce a partial ordering over policies, whereby policy $\pi$ is considered stronger than policy $\pi'$ if the value function of the former exceeds that of the latter over the entire state space:

$$\pi \geq \pi' \iff V^\pi(s) \geq V^{\pi'}(s) \quad \forall s \in S$$

(2.6)

A policy that is stronger than or equal in strength compared to all other policies is known as an optimal policy. Though there may be more than one optimal policy, it is standard to denote them singularly as $\pi^*$. The value and action-value functions of an optimal policy are referred to as the optimal value function $V^*$ and optimal action-value function $Q^*$ respectively. An equivalent formal definition is as follows:

$$V^*(s) = \max_\pi V^\pi(s) \quad \forall s \in S$$

(2.7)

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) \quad \forall s \in S, a \in A$$

(2.8)

### 2.2 Reinforcement Learning (RL)

When an agent is first deployed into an unfamiliar environment, it may not know anything of the environment’s dynamics, nor the strategies that are likely to be rewarding. Under these conditions, the agent cannot be expected to perform optimally right away; instead, it must learn how to behave by interacting with the environment and adjusting its behaviour accordingly. This is the central problem studied in Reinforcement Learning (RL). In this section, we overview some important RL concepts and algorithms. For brevity, we restrict our scope to the background needed to understand our later experiments. For an in-depth introduction to classical RL, we point the reader to Sutton and Barto’s well-known book: Reinforcement Learning: An Introduction [Sutton and Barto 1998].
2.2.1 Tabular Action-Value Methods

Action-value methods are a family of reinforcement learning algorithms whose aim is to learn the optimal action-value function, $Q^*$. Learning this function is closely related to learning an optimal policy, because an optimal policy can be derived from $Q^*$ by always acting greedily with respect to it, i.e. by choosing actions that maximise $Q^*(s, a)$. The mathematical motivation for action-value methods stems from the following relationship:

$$Q^*(s,a) = E_{\pi^*}[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a]$$  \hspace{1cm} (2.9)

This is known as the Bellman equation for the action-value function. To understand the equation, recall that $Q^*(s,a)$ is defined as the return expected if the agent takes action $a$, then follows an optimal policy. This quantity can be decomposed into the expected immediate reward from taking action $a$, plus the expected return from acting greedily thenceforth, discounted for the one time step already taken. This type of calculation, where a function’s value at one point in time is calculated from a later value of that same function, is known as bootstrapping.

For the type of tasks we are interested in, there are so many unique states that it is infeasible to solve Equation 2.9 analytically. However, an approximation method known as policy iteration can be used to refine a solution estimate over time. Policy iteration involves updating two functions in tandem: an action-value function, and a policy derived from it. The action-value function is updated in accordance with the Bellman equation, by shifting the current action-value estimates towards more informed, bootstrapped estimates from later time steps. At the same time, the policy is updated to maintain consistency with the action-value function, i.e. to favour the actions with the greatest action-values.

In videogames, one complication with this approach is that it is not usually possible to store and update action-values individually. However, for ease of explanation, we first consider the tabular case, where the number of state/action combinations is assumed to be small enough that the action-value function can be stored as a table. In this setting, it is possible to update action-values individually, and algorithms exist that are guaranteed to learn an optimal policy, provided they are configured correctly. Below, we describe two such algorithms: SARSA and Q-Learning.

SARSA

From the perspective of a reinforcement learning agent, there are two aspects of the Bellman equation (Equation 2.9) that make the expression on the right-hand side difficult to estimate: The fact that the optimal action-value function is initially unknown, and the presence of the expectation. The SARSA\(^3\) algorithm [Rummery and Niranjan 1994]

---

\(^3\)The acronym stands for State-Action-Reward-State-Action, though this full name is rarely used.
Algorithm 1 SARSA

1: Initialise $Q(s, a)$
2: for each episode do
3: Sample $s$ from start distribution
4: Choose $a$ from $s$ using policy derived from $Q$
5: while $s$ is not terminal do
6: Take action $a$, observe $r$, $s'$
7: Choose $a'$ from $s'$ using policy derived from $Q$
8: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
9: $s \leftarrow s'$
10: $a \leftarrow a'$
11: end while
12: end for

(Algorithm 1), addresses these difficulties by updating towards a slightly different target:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \quad (2.10)$$

where $r, s'$ and $a'$ are sampled, as per lines 5 – 6 of Algorithm 1.

The logic behind this update rule is as follows:

- Rather than bootstrapping from the optimal action-value function, SARSA learns action-values for the current policy. The point is that knowing the action-values of the current policy is sufficient to improve it, so policy iteration will bring $Q$ closer to $Q^*$ over time.

- To account for the inaccuracy introduced by using sampled targets, the old action-value estimates are not replaced entirely; instead, they are adjusted towards the sampled targets using a linear scale factor of $\alpha$, which is known as the learning rate. The rationale for this approach is that if the sample-based updates are performed many times and the learning rate is sufficiently small, the net update will be close to that which would be achieved by training towards the true expectation.

In the tabular setting, SARSA is guaranteed to converge to an optimal policy provided that, in the limit, all state-action pairs are visited infinitely many times and the policy becomes strictly greedy with respect to the action-value function.

Q-Learning

In most RL algorithms, the agent does not act strictly greedily, but is slightly random in its decision-making. This is done to ensure that the agent remains capable of discovering improvements. For example, a common strategy is $\epsilon$-greedy exploration, where the agent selects a greedy action with probability $1 - \epsilon$ and tries a random action with probability $\epsilon$. 

(May 26, 2019)
Algorithm 2 Q-Learning

1: Initialise $Q(s,a)$
2: for each episode do
3: Sample $s$ from start distribution
4: while $s$ is not terminal do
5: Choose $a$ from $s$ using policy derived from $Q$
6: Take action $a$, observe $r, s'$
7: $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max_a Q(s', a') - Q(s,a)]$
8: $s \leftarrow s'$
9: end while
10: end for

Since the SARSA update rule (Equation 2.10) bootstraps the training target from the action-value of the next selected action, the target will be negatively affected if the agent subsequently explores a weak action. For example, suppose that an agent for Super Mario Bros. is learning to jump over a pit. The control logic in this game is such that if the player releases the jump button for a single frame, Mario will cease the ascent stage of a jump and begin falling. Therefore, if Mario is mid-way over the pit and the agent takes an exploratory action that releases the jump button, Mario will start falling into the pit and the action-value of the previously selected action will likely receive a large negative adjustment, even if the jump button was still held at that point.

The Q-Learning algorithm [Watkins 1989] (Algorithm 2) addresses this issue by bootstrapping from the maximum action-value at the next time step:

$$Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max_a Q(s', a') - Q(s,a)]$$ (2.11)

In the pit-jumping example, this means that any time the agent explores an action that releases the jump button, previous actions will be updated on the assumption that the jump button continued to be held (provided that this yields the maximum target value). Q-Learning is accordingly known as an off-policy algorithm, meaning that it learns action-values for one policy (the greedy policy) while executing another (a policy that incorporates exploration). Q-Learning’s off-policy updating enables it to converge to $Q^*$ even without reducing the exploration rate to zero in the limit, unlike under SARSA.

2.2.2 Deep Reinforcement Learning via DQN

The tabular reinforcement learning algorithms discussed in the last section offer strong theoretical guarantees, but are limited in their applicability. In videogames, the state space is typically so large that it is not feasible to store the action-value function in tabular form. Moreover, even if it were feasible to do so, states are so rarely revisited that sample-based updating would take a very long time to converge. For these reasons, in domains with large state spaces, the action-value function is typically represented via some form of function
approximator. Common function approximators applied in reinforcement learning include Radial Basis Functions (RBFs), Cerebellar Model Arithmetic Computers (CMACs) and Artificial Neural Networks (ANNs) [Frommberger 2010]. In recent videogames research that focuses on learning from raw visual representations, a specialised class of ANN known as Convolutional Neural Networks (CNNs) [Mnih et al. 2015] is particularly well-suited. Whatever the specific type of function approximator used, a common property of all types listed is that they are capable of generalising between states, meaning that they can infer information about new states based on similar states that have been encountered before. This generalisation power enables the agent to learn a reasonable policy without having to visit each individual state many times.

Unfortunately, estimating action-values via function approximation weakens the convergence guarantees of algorithms such as SARSA and Q-Learning, especially when non-linear approximators such as ANNs are used. Until only a few years ago, the approach of deep reinforcement learning, or combining reinforcement learning with deep neural function approximators, was viewed as being inherently unstable [Mnih et al. 2016]. However, Mnih et al. [2015] showed that introducing two simple adjustments to the Q-Learning algorithm – experience replay and using a target network – made it possible to train a CNN in a stable manner across a wide suite of Atari 2600 games. The authors named their CNN architecture a Deep Q-Network (DQN), and the adjusted version of the Q-Learning algorithm as the DQN algorithm, although the abbreviation “DQN” is now commonly used to refer to either. Since we make heavy use of experience replay and target networks in our own experiments, we now describe these techniques in some detail. We also explain the further enhancement of Double DQN [Van Hasselt et al. 2016], which was introduced soon after Mnih et al.’s [2015] work and is leveraged in Chapter 4.\(^4\)

Experience Replay

In videogames, successive states tend to be very similar to one another. Therefore, if the agent is trained on sequential experience, as per the standard Q-Learning algorithm (Algorithm 2), the action-value function will undergo repeated updates over a narrow range of inputs. In the deep learning setting this is problematic, because neural networks tend to overspecialise to recent examples and “forget” older information; an issue known as catastrophic forgetting [McCloskey and Cohen 1989].

To address this issue, the DQN algorithm uses experience replay [Lin 1993]. Rather than learning from experience sequentially, the agent stores transitions in a cache and replays samples from it during training. In the original DQN algorithm [Mnih et al. 2015], the replay sampling was uniform random and the oldest memories were overwritten when

\(^4\)Many other enhancements to DQN have been proposed since the original paper, but are excluded here for brevity and because they are not used in our experiments. Curious readers are pointed towards the Rainbow agent of Hessel et al. [2018], which combines many of these enhancements.
the cache reached capacity. This is the approach we take in our experiments. However, it is worth noting that more sophisticated approaches exist, such as prioritised experience replay [Schaul et al. 2016], where samples that previously led to large training errors are favoured for replay. We also note that experience replay is not the only way to decorrelate training samples. Another common approach is to run multiple agents in parallel and learn from their combined experience asynchronously [Mnih et al. 2016]. The idea is that the various actors will spread out across different regions of the state space and thus naturally diversify the stream of training samples. A side benefit leveraging parallelism is that it can significantly reduce the wall clock training time [Mnih et al. 2016].

**Separate Target Network**

As discussed earlier, one of the main reasons for using function approximation is to enable generalisation between states. However, a downside of state generalisation is that it can destabilise the bootstrapping process. For example, suppose that an agent for *Super Mario World* has just commenced learning, so that all action-values estimates are close to zero. Now suppose it collects a coin on the transition from state $s$ to $s'$, resulting in a reward of $+1$. As per the Q-Learning update rule, $Q(s, a)$ will be trained towards $1 + \gamma \max_a Q(s', a') \approx 1$. However, since states $s$ and $s'$ occur only one frame apart, they may bear a strong pixel-wise similarity (for example, see Figures 2.1a and 2.1b). In this case, until the agent has learned the significance of a coin, the two states will yield similar latent states within the neural network, and thus $Q(s', a)$ will likely increase too. This is problematic, because it means that increasing $Q(s, a)$ also increases one of the values from which $Q(s, a)$ is bootstrapped. If the same transition is sampled repeatedly, the positive feedback loop may cause the action-value estimates to explode.

To address this type of problem, DQN bootstraps action-values from a target network, which is just a lagging copy of the network being trained. The target network is refreshed periodically, but otherwise held fixed. In the example above, this means that $Q(s, a)$ will continue to be trained towards 1 until the target network is updated. Clearly this does not completely remove the problem, but it does slow down the rate at which the positive

![Figure 2.1](May 26, 2019)
feedback escalates. As Mnih et al. [2015] point out, another way of thinking about the target network measure is that it brings the training process closer to supervised learning, where it is easier (relatively speaking) to train deep networks in a stable manner.

Later work by Lillicrap et al. [2016] introduced the similar approach of *soft target updates*, where the weights of the target network are no longer replaced by weights of the learned network periodically, but are instead adjusted gradually as follows:

\[
\theta' \leftarrow \tau \theta + (1 - \tau) \theta'
\]  

where \( \theta' \) are the weights of the target network, \( \theta \) are the weights of the learned network, and \( \tau \ll 1 \) controls the size of the update, which is applied after every training minibatch.

### Double Q-Learning

Recall that the Q-Learning update rule (Equation 2.11) involves taking a maximum over the next state’s action-values. A consequence of this operation is that noise in the action-value estimates generally yields inflated training targets. For example, at the start of training, the outputs of a randomly initialised deep Q-network will not be precisely zero, but will lie in a small range centred near zero. For illustrative purposes, suppose that the action-values for the next state lie in the range (-0.02, +0.02). In this case, the \( \max \) operator will return an inflated next state value of +0.02.

*Double Q-Learning* [Hasselt 2010] addresses this problem by using one action-value function, \( Q_A \), to determine the greedy action, and another action-value function, \( Q_B \), to evaluate the greedy action. The point here is that \( Q_A \) is likely to select an action for which its own evaluation is inflated. However, so long as the bias of \( Q_B \) is uncorrelated with that of \( Q_A \), the evaluation provided by \( Q_B \) will be unbiased. Under this approach, the update rule for \( Q_A \) is as follows:

\[
a^* = \arg \max_a Q_A(s', a)
\]

\[
Q_A(s, a) \leftarrow Q_A(s, a) + \alpha [r + \gamma Q_B(s', a^*) - Q_A(s, a)]
\]

The update for \( Q_B \) is identical, but with labels swapped.

In Hasselt’s [2010] original work, the correlation between the functions’ biases was limited by training only one function per experience sample, with the choice of function made randomly. Unfortunately though, this approach halves the algorithm’s sample efficiency. A compromise that synergises well with DQN algorithm is *Double DQN* [Van Hasselt et al. 2016], where DQN’s live training network is used as \( Q_A \), and the target network is used as \( Q_B \). Though these networks are not completely uncorrelated, Van Hasselt et al. [2016] showed that the approach is effective in reducing action-value inflation. As the authors point out, it not a given that reducing action-value inflation will actually improve the agent’s performance, since uniform inflation has no effect on an \( \epsilon \)-greedy policy. However,
they showed empirically that Double DQN yields significant improvement over plain DQN in many Atari games.

### 2.2.3 The Deep Deterministic Policy Gradient (DDPG) Algorithm

In many classical videogames, the number of actions available is finite. For example, in Atari 2600 games, an action consists of a joystick direction (of which there are 9 discrete possibilities) plus a boolean input corresponding to the “fire” button, yielding a total of 18 possible actions. In applications of DQN to Atari, the neural network is thus configured to have 18 outputs, and the greedy action is calculated by simply iterating through the outputs. Unfortunately though, this approach is ill-suited to the tasks we consider in Chapter 5, where the action space is continuous and cannot be enumerated.

An alternative approach, which is better suited to such domains, is to define an objective for the agent to maximise that depends on the parameters of the agent’s policy. For example, the objective might be to maximise the expected game score, and the policy might be represented by a neural network that outputs a continuous action. In this type of application, the weights of the neural network would typically constitute the policy parameters. Notationally, we denote the agent’s policy as \( \mu_\theta \), with the subscript \( \theta \) denoting the policy’s parameters. The agent’s objective is denoted \( J(\theta) \). The general idea is to shift the policy parameters in the direction that most sharply increases the objective. This direction, \( \nabla_\theta J \), is known as the policy gradient [Sutton et al. 2000]. In continuous action domains, a key advantage of policy gradient methods is that they do not require enumerating the action set.

Several prominent policy gradient methods have been introduced in recent years, including Trust Region Policy Optimisation (TRPO) [Schulman et al. 2015], Asynchronous Advantage Actor-Critic (A3C) [Mnih et al. 2016] and Proximal Policy Optimisation (PPO) [Schulman et al. 2017]. However, since our primary research focus in Chapter 5 is not on the policy gradient method per se, but rather on implementing a high-level framework that uses the policy gradient method as a tool, we focus here on explaining the one particular algorithm that we use; namely, the Deep Deterministic Policy Gradient (DDPG) algorithm [Lillicrap et al. 2016].

DDPG involves many of the same concepts as DQN. Like DQN, the algorithm is off-policy; it attempts to learn a deterministic policy, \( \mu_\theta \), while generating experience from a stochastic policy, \( \beta \), to allow for exploration. These policies can be thought of as being analogous to DQN’s greedy and \( \epsilon \)-greedy policies, respectively. The agent’s aim is to maximise the expected return over the discounted state distribution under \( \beta \) (defined shortly). Mathematically, the objective function is defined as follows:

\[
J(\theta) = \int_S \rho^\beta(s)Q^{\mu_\theta}(s, \mu_\theta(s))ds
\]

(2.15)

where \( \rho^\beta(\cdot) \) denotes discounted state distribution under \( \beta \).
The intuition behind $\rho^\beta(s)$ is that it gives the visit probability density of state $s$ under policy $\beta$, discounted for the average time it takes to reach state $s$. To be precise:

- Let $\rho_0(\cdot)$ denote the initial distribution over states.
- Let $\rho^\beta(s \rightarrow s', t, \beta)$ denote the visit probability density of state $s'$ after beginning in state $s$ and following policy $\beta$ for $t$ steps.
- $\rho^\beta(s') = \int_S \sum_{t=1}^{\infty} \gamma^{t-1} \rho_0(s) \rho^\beta(s \rightarrow s', t, \beta) ds$

For the objective in Equation 2.15, Silver et al. [2014] showed that the policy gradient has the following elegant approximation:

$$\nabla_{\theta} J(\theta) \approx \int_S \rho^\beta(s) \nabla_{\theta} \mu(\theta)(s) Q^\mu(s, a) ds$$

(2.16)

$$= E_{s \sim \rho^\beta} [\nabla_{\theta} \mu(\theta)(s) \nabla_a Q^\mu(s, a)|_{a = \mu(\theta)(s)} ]$$

(2.17)

Note that calculating the sampled gradient requires the agent to maintain two functions: The deterministic policy, $\mu_\theta$, and the action-value function, $Q^\mu$. Since the former is responsible for action selection and the latter is responsible for action evaluation, they are also referred to as the actor and critic respectively. Policy gradient algorithms involving these two components are known as actor-critic methods [Sutton and Barto 1998].

Silver et al. [2014] showed that Equation 2.17 yields the true policy gradient provided that the actor and critic are represented via a special class of function approximators. However, just as tabular Q-Learning is ill-suited to videogames, the class of functions described by Silver et al. [2014] is too restrictive for the applications we are interested in. Fortunately though, the same stability measures that enable deep function approximation in DQN are also applicable here. The DDPG algorithm is essentially just this – a combination of Silver et al.’s [2014] learning rules and the stability measures of DQN. The actor is updated via sampled estimates of the policy gradient, and the critic is updated via Q-Learning, treating $\mu(\theta)(s')$ as the greedy action in state $s'$. The parameter updating for both the actor and critic is smoothed using soft target updates, and the training samples are decorrelated via experience replay.

Since DDPG is an off-policy algorithm, the choice of exploration policy, $\beta$, is flexible. Note though that the $\epsilon$-greedy approach described earlier is ill-suited to domains with continuous action spaces, because selecting exploratory actions completely at random is often physically unrealistic and may even be dangerous. (For example, when learning to control a robot arm, it is undesirable to suddenly explore an enormous torque.) Since exploration in continuous action domains is the major focus of Chapter 5, we defer a detailed discussion of this topic until then.
2.3 Addressing Reward Sparsity

In all of the RL algorithms discussed so far, the agent’s learning is guided by the reward signal. Accordingly, if the rewards are only sparsely distributed, the agent is liable to learn slowly. In the worst case, the rewards may be so sparse that the agent cannot learn anything at all, as was the case for the original DQN agent [Mnih et al. 2015] in the game Montezuma’s Revenge. In this section, we describe some common ways of mitigating this issue. First, we explain multi-step bootstrapping (Section 2.3.1), which is designed to propagate the reward signal more quickly through time. Next, we describe two ways of modifying the reward scheme to make it more informative: intrinsic reward schemes (Section 2.3.2) and potential-based reward shaping (Section 2.3.3).

2.3.1 Multi-Step Bootstrapping

In the basic Q-Learning algorithm introduced earlier (Algorithm 2), the sampled estimate of the expected return is bootstrapped from one time step into the future:

\[ Q(s, a)_{\text{one-step target}} = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') \]  

(2.18)

One-step bootstrapping is inefficient in sparse reward videogames, because it is slow to propagate the reward signal. For example, suppose that an agent for Montezuma’s Revenge has just obtained the key for the first time. (For reference, the key is located roughly 100 time steps from the initial state under best play.) When the experience of collecting the key is processed by the algorithm, the state-action pair that preceded the key collection will be positively reinforced. However, the earlier actions that were required to bring the player close to the key will not be reinforced immediately. Instead, the state-action pairs that are two frames away from the key collection will be increased next, followed by those that are three frames away, etc.\(^5\) Therefore, even after the agent has discovered the key, learning the full path may take a long time.

An alternative approach, which propagates information faster, is to bootstrap from multiple time steps into the future. For example, a two-step target can be used:

\[ Q(s, a)_{\text{two-step target}} = r_{t+1} + \gamma r_{t+2} + \gamma^2 \max_{a'} Q(s_{t+2}, a') \]  

(2.19)

This can be extended to an arbitrary n-step target:

\[ Q(s, a)_{\text{n-step target}} = \sum_{k=0}^{n-1} \gamma^k r_{t+1+k} + \gamma^n \max_{a'} Q(s_{t+n}, a') \]  

(2.20)

The modification of Q-Learning to use an n-step target is known as n-step Q-Learning [Mnih et al. 2016]. Despite its name, it should be noted that the use of a multi-step return

\(^5\)If the action-value function is estimated by a state generalising function approximation then this is not strictly true, but the point remains that the reward signal propagates slowly through time.
means that the algorithm no longer learns a strictly greedy policy. The reason is that any
of the intermediate actions taken between time $t$ and $(t + n)$ may have been non-greedy.
In this sense, the algorithm is really a blend between Q-Learning and on-policy algorithms
such as SARSA.

In general, there is a trade-off between using one-step and multi-step return estimates.
While multi-step estimates propagate information faster, they also have greater variance
due to the additional random sampling involved. One way of balancing this trade-off is to
use the $\lambda$-return [Sutton and Barto 1998], which is a weighted mixture over all possible
bootstrap lengths. However, since this involves calculates action-values for all future
states, it is computationally expensive to use in algorithms like DQN, where calculating
the action-values for each state requires a forward pass through a deep network. A crude
alternative that can be used in episodic settings is the mixed Monte Carlo (MMC) return
[Bellemare et al. 2016], which interpolates between the one-step target and the full sampled
return for the episode (also known as the Monte Carlo return):

$$Q(s, a)_{MMC} = \eta R_t + (1 - \eta)Q(s, a)_{\text{one-step target}}$$

(2.21)

where $R_t$ is the Monte Carlo return, and $0 \leq \eta \leq 1$ determines the mixing proportion of
the Monte Carlo return.

### 2.3.2 Intrinsic Reward Schemes

Multi-step bootstrapping can help propagate feedback faster in sparse reward domains,
but is of no use if the agent has not discovered any rewards at all. At the same time, it
is apparent that human videogame players do not rely solely on explicit environmental
rewards in order to learn. This is especially apparent in adventure videogames, where most
human players are motivated not only to increase the game score, but also to explore. Since
the drive to explore originates from within the agent, it is known as a form of intrinsic
motivation [Schmidhuber 1991, Oudeyer et al. 2007]. Rewards that are inherent to the
environment, such as score increases from defeating enemies and collecting powerups,
are known as extrinsic motivation. Intrinsic motivation is a broad topic in the study
of artificial agents, so for brevity we focus on two types of scheme that are particularly
relevant to our work in Chapter 3; namely count-based novelty [Strehl and Littman 2008,
Bellemare et al. 2016], where the agent is encouraged to reach states that have rarely
been visited before, and prediction error bonuses [Burda et al. 2018a], where the agent
is encouraged to experience “surprising” events, i.e. events that do not fit well with the
agent’s current model of the world. To date, these methods have had the most empirical
success in sparse reward Atari games. However, we note that many other approaches
to intrinsic motivation exist, such as empowerment [Klyubin et al. 2005], salient events
[Barto et al. 2004], and mutual information [Still and Precup 2012].
SECTION 2.3: ADDRESSING REWARD SPARSITY

Count-Based Novelty

The basic idea behind count-based novelty methods is to maintain a visit count for each state and pay the agent an intrinsic reward for reaching rarely-visited states. Following the theoretical work of Strehl and Littman [2008], the intrinsic reward is commonly chosen to take the following form:

\[ R^+_{a}(s, s') = \frac{\beta}{\sqrt{n(s,a)}} \]  

(2.22)

where \( n(s,a) \) is the number of times action \( a \) has been chosen in state \( s \). In Strehl and Littman’s [2008] analysis, \( \beta \) is a theoretically derived constant, and it is assumed that the state-visit counts are stored in a table. For this case, the authors provide some guarantees regarding the near-optimality of the resultant policy and the time taken to train it. However, in deep reinforcement learning applications, where it is much harder to obtain theoretical guarantees, \( \beta \) is usually just treated as an adjustable reward scaling coefficient, and the state-action visit count, \( n(s,a) \), is often just replaced by a state visit count, \( n(s) \), for simplicity.

Under all the intrinsic motivation schemes considered in this thesis, the intrinsic rewards are incorporated into the MDP by simply adding them to the extrinsic reward:

\[ R_{a}(s, s') = R^e_{a}(s, s') + R^+_{a}(s, s') \]  

(2.23)

where \( R_{a} \) is the new, modified reward and \( R^e_{a} \) is the extrinsic reward.

The main challenge that arises in applying count-based bonuses to videogames is similar to that which arises with tabular learning algorithms; namely, the state space is typically so large that it is impractical to maintain individual visit counts, and even if it were possible, individual states are so rarely revisited that many counts would remain either 0 or 1 for a very long time. Just as the key to applying RL to large domains is generalising function approximation, this suggests that we need a visit count that generalises between states. That is, if state \( s \) has never been visited before, but very similar states have been visited, the visit count for state \( s \) ought to be non-zero. A visit density function that generalises in this manner is known as a pseudo-count [Bellemare et al. 2016]. The question of exactly how the pseudo-count should generalise is a non-trivial one, but since this is a major topic of research in Chapter 3, we defer a detailed discussion until then.

Prediction Error Bonuses

Whereas count-based schemes reward novel experience, prediction error bonuses are best characterised as rewarding “surprise”. To be more precise, prediction error agents attempt to learn some model of the environment, and receive intrinsic rewards for generating experience that does not conform to the model [Chentanez et al. 2005, Schmidhuber 2010, Vigorito and Barto 2010, Stadie et al. 2015]. Despite the difference in characterisations,
there is, however, a strong link between prediction error bonuses and count-based schemes: Intuitively, one would expect the prediction errors to be greatest in states that the agent has rarely visited. Similarly, just as the effectiveness of count-based schemes is mainly determined by the choice of pseudo-count model, the effectiveness of prediction error schemes is mainly determined by the prediction task chosen. Some common types of prediction task used to generate intrinsic rewards are as follows:

- **Next state prediction.** Given the current state and action, the agent attempts to predict the next state. The predictive model in this case is called a task dynamics model. In domains with high-dimensional state representations, e.g. Atari screen captures, the prediction task is often simplified by first converting the state representation to a lower-dimensional representation over features.

- **Action prediction.** Given the current state and the next state, the agent attempts to predict the action that was taken. In this case, the predictive model is called an inverse task dynamics model. An advantage of this approach is that it forces the agent to focus on elements of the representation that are under its control, e.g. the location of the player sprite, but not on extraneous details that have no bearing on action prediction, e.g. moving objects in the game background. Accordingly, the intrinsic rewards will not be affected if new extraneous details are introduced.

- **Predicting an encoding of the current state.** Given the current state, the agent attempts to predict an encoding of the state. For example, under random network distillation (RND) bonuses [Burda et al. 2018b], the agent attempts to predict the output of a randomly initialised CNN applied to the current state. As we explain further in Chapter 3 (Section 3.1.2), this type of approach makes the intrinsic rewards robust to stochasticity in the state transition function.

### 2.3.3 Potential-Based Reward Shaping

Besides their natural inclination to explore, another, perhaps more obvious reason why human players are not as troubled by reward sparsity as artificial agents is because human players usually have some idea of what they are “supposed” to do. For example, suppose that a human player of an adventure game notices a treasure chest at the edge of the screen. Suppose also that on their first attempt to reach the chest, they make it most of the way there, but die to an enemy. Based on their understanding of the objective, they can infer that their early actions were strong and their final actions were weak, even though they failed to receive explicit positive feedback by reaching the chest.

In situations where an artificial agent is given prior knowledge about the goal, e.g. “the aim is to get the powerup”, a natural idea is to try to derive a more incremental reward scheme so that the agent can learn in a similar manner to the human player in the example
above. However, modifying a task’s reward scheme is potentially dangerous, because an optimal solution to the modified task may not be an optimal solution to the original task, and vice-versa. (This issue also applies to the intrinsic reward schemes discussed earlier.) Ideally, the modified reward scheme should preserve optimal policies. Fortunately, this property is guaranteed by a particular type of reward modification, known as potential-based shaping \[\text{Ng et al. 1999}\]. Under potential-based shaping, the reward function is modified as follows:

\[
\hat{R}_a(s, s') = R_a(s, s') + \gamma \Phi(s') - \Phi(s)
\] (2.24)

where \(\hat{R}_a\) is the new reward function, \(R_a\) is the original reward function, \(\gamma\) is the MDP’s discount factor, and \(\Phi : S \rightarrow \mathbb{R}\) is a potential function that encodes the agent’s prior knowledge about the task.

Typically, \(\Phi\) is configured to be a rough estimate of the optimal value function, \(V^*\). In the special case where the functions match exactly, the optimal value function under the modified reward scheme becomes zero everywhere. Accordingly, provided the action-value function is initialised to zero, learning an optimal policy becomes relatively easy: All the agent must do is retain zero action-values for the optimal actions, while learning negative action-values for sub-optimal actions. Nonetheless, \(\Phi\) does not have to be a close estimate of \(V^*\) for potential-based shaping to be effective; this case merely highlights some of the intuition behind the approach.

Regardless of the choice of potential function, potential-based shaping is proven to preserve optimal policies in the tabular case \[\text{Ng et al. 1999}\]. In fact, in tabular Q-Learning, potential-based shaping is merely equivalent to initialising the action-values differently \[\text{Devlin and Kudenko 2012}\]. However, it is important to note that these guarantees say nothing about the speed of learning under potential-based shaping. In fact, a poor choice of potential function may actually slow learning progress.

### 2.4 Hierarchical Reinforcement Learning

In reality, the question of whether or not a particular videogame has “sparse” rewards is a matter of perspective. From the standpoint of an artificial agent that selects a new action at every single frame, the reward scheme in Montezuma’s Revenge seems extremely sparse, because the first scoring opportunity is around 100 frames from the start location. However, from the perspective of human players, who appear to view such tasks in terms of extended actions, such as “descend ladder” and “run to the end of the platform”, the first reward appears to be only a handful of decision-making steps away. In the RL literature, these kind of extended actions are commonly referred to as temporally abstract actions (to reflect the fact that their durations are non-concrete), or more succinctly as skills.

Besides helping to mitigate reward sparsity, skills can also be used to improve exploration efficiency. To illustrate this point, consider a gridworld agent exploring via
an \(\epsilon\)-greedy policy over an action space of \(A = \langle\text{up}, \text{down}, \text{left}, \text{right}\rangle\). Since \(\epsilon\)-greedy selects exploratory actions at uniform random, the \text{up} action will be sampled with the same likelihood as the \text{down} action. Likewise, the \text{left} and \text{right} actions will be selected with similar frequency. As such, the agent’s exploratory actions will tend to negate each other. An alternative way of phrasing this issue, as per Thrun [1992], is that \(\epsilon\)-greedy is an undirected exploration method. However, if the agent explores by randomly executing skills, committing to them until they have been completed, the short-term cancellation of actions will be avoided. That is, under Thrun’s [1992] terminology, skills can be used to achieve directed exploration.

Yet another application of skills, which we leverage strongly in Chapter 4, is for facilitating knowledge reuse. For example, suppose that an agent for a dungeon crawling videogame manages to complete the first level, and then finds itself in a new level with a different layout to the first but containing many of the same enemy types. While a policy for solving the first level cannot be reused in its entirety, a set of skills for defeating the enemies in the first level may help the agent to learn the second level faster.

Finally, one of the main motivations for training skills is that they can be used to decompose long tasks hierarchically into more manageable pieces. For example, the task of escaping the first room in Montezuma’s Revenge can be decomposed into a shallow plan containing two skills as “get key” → “open door”. Taking this further though, the “get key” skill can itself be expressed in terms of lower-level skills, e.g. “descend ladder” → “jump to rope” → “jump to platform” → “descend ladder” → “jump over skull” → etc. For this reason, the broad field of research concerned with acquiring and exploiting skills is known as hierarchical reinforcement learning.

Hierarchical RL has been studied for decades under the guise of several major frameworks: Hierarchical Abstract Machines (HAMs) [Parr and Russell 1998], the Options Framework [Sutton et al. 1999] and MAXQ [Dietterich 2000]. However, since the options framework is probably the best known approach, and is the only one of these frameworks that we explicitly leverage in our contributions, we do not cover HAMs or MAXQ here. For a summary of the important differences between the three frameworks, we point the reader to Barto and Mahadevan’s [2003] survey.

2.4.1 The Options Framework

An option is one concrete way of modelling a skill. Informally, options can be thought of as extended actions that are only executable in certain situations. Formally, an option is defined as a tuple \((\mathcal{I}, \pi, \beta)\) where:

- \(\mathcal{I} \subseteq S\) is the set of states the option can be initiated from.
- \(\pi : S \times A \rightarrow [0,1]\) is a policy that returns the probability of selecting action \(a\) when in state \(s\).
• $\beta: \mathcal{S} \rightarrow [0,1]$ returns the probability that the option will terminate in a given state.

If an MDP is modified to allow the agent to act over options, the decision-making process is technically no longer Markov. The reason for this is that the external environment state no longer conveys all information needed to predict future events. Since options have prolonged execution times, calculating quantities such as the expected return requires one to know which option, if any, is currently executing. Decision-making processes that are Markov except for this detail are called Semi-Markov Decision Process (SMDP).

The Q-Learning algorithm can be extended to SMDPs via multi-step bootstrapping (Section 2.3.1). The idea is to treat an option execution that lasts for $n$ steps as a single transition, and perform a corresponding $n$-step update. The resultant algorithm is known as SMDP Q-Learning [Sutton et al. 1999]. To prevent the agent from becoming locked into sub-optimal options, each option’s termination condition, $\beta$, can be overridden so that the option will terminate if the expected return from executing an alternative option becomes greater. Options with this modification are known as interrupting options. We return to this idea in Chapter 4, proposing an alternative interruption scheme for use in a planning context.

### 2.4.2 Autonomous Skill Acquisition

In order for a hierarchical reinforcement learning agent to be considered truly autonomous, it ought to be capable of identifying and learning skills without the intervention of a human expert. The problem of autonomous skill acquisition is closely related to that of subgoal detection, since many skills are effectively just a means for completing a subgoal, e.g. “get key”, “defeat enemy”, “cross river”. Autonomous skill and subgoal identification constitutes arguably the central challenge in hierarchical RL, with research dating back decades [Thrun and Schwartz 1995, Digney 1998] and continuing into the modern era of deep reinforcement learning [Vezhnevets et al. 2017, Machado et al. 2018b].

Historically, one of the most common approaches to subgoal identification has been the bottleneck method [Menache et al. 2002]. To illustrate the intuition behind this idea, suppose that an agent is situated in a gridworld consisting of two rooms separated by a doorway. Given any task that requires navigation between the two rooms, it will be necessary for the agent to enter the doorway at some stage. As such, a skill for reaching the doorway is likely to be useful. More generally, the bottleneck method can be characterised as looking for chokepoints in the state space structure. In domains where it is viable to track statistics for individual states, bottlenecks can be identified by analysing visit counts or by looking for commonalities between sample trajectories [Menache et al. 2002, Şimşek and Barto 2004, Şimşek et al. 2005]. However, in what should now be recognised as a familiar theme, methods that rely on exact states being visited multiple times cannot easily be extended to larger domains. There have recently been some attempts to extend
the bottleneck idea to the deep RL setting [Machado et al. 2017; 2018b], but as we explain further in Chapter 3, this line of research remains very much a work in progress.

Another prominent approach to autonomous skill acquisition is skill chaining [Konidaris and Barto 2009]. The basic idea here is to learn a set of skills for solving a task in reverse order. First, the agent learns a skill for solving the task upon initiation from states that are very nearly solved. Next, the agent learns skills for reaching these “nearly solved” states when initiated slightly further away from task completion. Continuing in this manner, the agent eventually learns a set of skills than can be chained together to solve the task from any initiation state. One strength of this approach is that it does not require the state space to be small or discrete. However, a downside is that it requires some way of generating training data in a back-to-front manner, which may be non-trivial in sparse reward tasks where the objective is initially unknown. For this reason, the approach is ill-suited to the problem studied in Chapter 3, where we aim to identify subgoals autonomously with minimal \textit{a priori} knowledge about the reward function, but it strongly influenced our approach in Chapter 4, where we consider sparse feedback planning tasks in which the agent is assumed to have some knowledge of the goal.

In the more recent setting of deep reinforcement learning, another style of approach for acquiring skill-like knowledge is to craft the neural network architecture in such a way that the agent is forced to reason in a temporally abstract manner [Vezhnevets et al. 2016; 2017]. We save an explanation of Vezhnevets et al.’s [2017] method until Chapter 3, since it is better contextualised after we have introduced our approach.

2.4.3 Acquiring Flexible Skills

As mentioned earlier, one of the main motivations for training skills is to facilitate knowledge reuse. To this end, certain types of skill may be more reusable if they are goal-parameterised. da Silva et al. [2012] illustrate this point via the example of an agent learning to shoot a soccer ball. Rather than training distinct skills for kicks of varying strength, it may be more efficient to train a single, flexible kick with an adjustable strength.

In this section, we discuss two common approaches to training and representing flexible skills: (1) Training a map from task parameters to policy parameters and (2) Universal value function approximation, where the goal of the skill (i.e. the strength of the shot in the example given) is passed as an explicit input to the policy.

Mapping Task Parameters to Policy Parameters

Suppose that a soccer agent has already learned how to kick a ball at 50% power and at 60% power. From this knowledge, an intelligent agent ought to be able to interpolate how to kick a ball at 55% power. Following the same intuition, one approach to acquiring flexible skills is to seek a mapping $\Theta : T \rightarrow \theta$ from task parameters $t \in T$ to policy parameters $\theta \in \mathbb{R}^N$ [da Silva et al. 2012; 2014a].
Of course, policies with a complex functional form will likely demand a more sophisticated mapping than a simple linear interpolation. To the best of our knowledge, interpolating the weights of a deep neural network is currently beyond the capability of such methods. However, by using a sophisticated, non-linear regression, da Silva et al. [2012] were successful in applying this approach to a simulated dart throwing domain, where policies were represented by Dynamic Movement Primitives (DMPs) [Schaal et al. 2005] containing around 40 parameters. Closely related work has focused on active learning of parameterised skills, where the agent is given some control over the tasks it learns the mapping from [da Silva et al. 2014b, Fabisch and Metzen 2014].

**Universal Value Function Approximators (UVFAs)**

In a standard MDP, an implicit assumption is that the agent’s objective is encoded by the reward function. However, in the example of the soccer agent above, it may seem unnatural to think of the desired kick power as being an intrinsic property of the environment. Instead, it may be more natural to think of it as an internal goal of the agent, with the reward being an intrinsic one that is dependent on the goal. From this viewpoint, action-values no longer depend on just the state and action, but also on the goal. In other words, the action-value function ought to be written as $Q(s, a, g)$, instead of just $Q(s, a)$. Value functions that are goal-parameterised in this way are known as Universal Value Function Approximators (UVFAs) [Schaul et al. 2015].

Training a UVFA for the action-value function is relatively straightforward; the only major modifications required to the learning algorithm are that the goal must be included alongside the state in the policy input, and the reward must be adapted according to the goal. Unlike the parameter mapping approach described above, this means that UVFAs are also easily compatible with deep function approximation, which makes them our method of choice when training goal-parameterised skills in Chapters 4 and 5.
In recent years, videogames have become an increasingly popular domain for artificial intelligence research. This is in large part due to the famous Deep Q-Network (DQN) agent of Mnih et al. [2015], which learned to play many Atari 2600 games to human level from only raw pixel input and a feed of the game score. Impressive as this was, DQN fared far better in reflex-driven games with frequent scoring opportunities, such as Video Pinball, than in sparse reward games where scoring opportunities are few and far between. Notoriously, it failed to learn a path to the first key in Montezuma’s Revenge (see Figure 1.1) after more than a month of experience. DQN’s weakness in sparse reward games has driven much subsequent research [Bellemare et al. 2016, Kulkarni et al. 2016, Osband et al. 2017].

1This chapter covers material previously published in [Dann et al. 2019].

Figure 1.1: The first screen from the Atari 2600 game, Montezuma’s Revenge (repeated from page 5).

The reason DQN struggles in sparse reward games is that until the agent has discovered some reward, it sees no incentive to favour one course of action over any other. It responds to this predicament in a manner typical of many learning-based agents: by simply choosing actions at random. In games with dense rewards, such as Video Pinball, random action selection is often sufficient for the agent to find rewards and start improving. However, in sparse reward games the agent can become stuck in a “chicken-egg” scenario, where it cannot improve its policy until it finds some reward, but it cannot discover any rewards until it improves its policy.

It has long been thought that hierarchical reinforcement learning [Parr and Russell 1998, Sutton et al. 1999, Dietterich 2000], which decomposes drawn-out tasks into sub-tasks, may be key to solving sparse reward problems efficiently. In fact, it has already been established that hierarchical agents can make rapid progress in sparse reward Atari games, so long as they are provided with appropriate subgoals by a human expert. For example, Kulkarni et al. [2016] demonstrated that an agent equipped with high-level way-points for the first room of Montezuma’s Revenge could learn to exit the room faster than all existing agents at the time. Roderick et al. [2018] went further by providing their agent with a factored state representation that included information such as “the player has the key”. From this, their agent learned to obtain the key then return to a door within a single life, a feat that many higher scoring agents still do not achieve (as explained in the next section). Of course, a major deficiency of using handcrafted subgoals is that it requires the manual effort of a human expert. Despite decades of research [Vezhnevets et al. 2017, Thrun and Schwartz 1995, Digney 1998, Şimşek et al. 2005, Konidaris and Barto 2009, Machado et al. 2017; 2018b], identifying subgoals from high-dimensional representations, such as Atari screen captures, remains a major open challenge.

The main contribution of this chapter is a new, autonomous method for deriving subgoals in such domains. The method operates by partitioning the state space according to a novel heuristic distance measure, which we term exploration effort. The approach can be readily applied to raw visual input, and the subgoals it identifies in Atari games resemble those identified by human experts in previous work. To utilise the subgoals found, we propose a novel intrinsic reward scheme, dubbed pellet rewards, which we so named because of its inspiration from the pellets in the game Ms. Pacman. The intuition behind pellet rewards is illustrated figuratively in Figure 3.1. Essentially, the idea is to transform sparse reward tasks into denser reward tasks by placing collectable bonuses throughout the state space that encourage exploration. We apply this approach to three Atari games with sparse rewards (Venture, Freeway and Montezuma’s Revenge), achieving performance comparable with strong baselines derived from density models [Bellemare
3.1 Issues with Existing Intrinsic Motivation Schemes

Currently, the strongest agents for sparse reward Atari games are driven by intrinsic motivation. The idea behind this approach is to provide the agent a reward bonus for generating novel or surprising experiences. As we explained in the background chapter (Section 2.3.2), two common ways of deriving such bonuses are count-based schemes, where the agent is encouraged to reach rarely visited states, and prediction error schemes, where the agent learns some kind of model and is rewarded for reaching states where the model is inaccurate. In this section, we review some notable agents from each family. Count-based agents are covered in Section 3.1.1 and prediction error agents are covered in Section 3.1.2. Throughout the discussion, we observe that despite the many differences between various bonus schemes applied to Atari, most of the successful ones reward essentially the same thing: visual novelty. That is, the size of the reward bonus is strongly correlated to how novel the observed screen is in a superficial, pixel-level sense. In Section 3.1.3, we explain why this property is helpful in some games, but not in others. Following this, we highlight a further issue with existing schemes; specifically, that they can cause the agent to become excessively risk averse (Section 3.1.4). Understanding these issues is necessary to appreciate several of the design choices made later when we introduce our approach.

3.1.1 Count-Based Novelty Methods

Count-based methods (Section 2.3.2) operate by maintaining a visit count for each state and paying the agent an intrinsic reward that diminishes with the observed state’s visit...
The bonus generally takes the following form:

\[ r^+(s) = \frac{\beta}{\sqrt{n(s)}} \]  

(3.1)

where \( n(s) \) is the state visit count and \( \beta \) is a reward scaling factor.

While simple conceptually, a major challenge in applying this approach to domains with image-based state representations is that the state space is typically too large to maintain individual visit counts. Bellemare et al. [2016] were the first to make significant progress on this problem in Atari, doing so by deriving a pseudo-count from a state density model (Section 2.3.2). In order to estimate densities without incurring an excessive computational overhead, they used the efficient Context Tree Switching (CTS) model [Veness et al. 2012]. To briefly summarise, CTS estimates the likelihood of observing a given screen by factoring the screen into pixels, calculating the probabilities of observing the individual pixel intensities (by quantising the intensities into buckets and conditioning on neighbouring pixels), then multiplying the individual pixel probabilities. Since this gives equal weight to all pixels, the novelty of a state is strongly correlated to how many “surprising” pixels it contains. In turn, this means the intrinsic rewards are sensitive to the game’s visual design, i.e. they are affected by aspects such as the size of sprites. Nonetheless, Bellemare et al.’s agent made significant progress on Montezuma’s Revenge, exploring 15 rooms of the first level and far outscoring previous agents.

Following Bellemare et al.’s work, Ostrovski et al. [2017] tried using a more sophisticated, neural density model in place of CTS. In particular, they used PixelCNN [van den Oord et al. 2016a;b], which follows the same underlying principle as CTS of multiplying conditional pixel probabilities. Across a range of experiments in Atari, they found that PixelCNN bonuses were generally more effective than those derived from CTS. Martin et al. [2017] proposed another closely-related approach, with the main difference being that the factors in their model correspond to Blob-PROST features [Liang et al. 2016] rather than individual pixels. (Briefly, the presence of a Blob-PROST feature indicates that a “blob” of contiguous, same-colour pixels exists at a particular location on the current screen.) This approach is likewise sensitive to game’s visual aesthetic. For example, in the graphically simple game of Pong there can be as few as 6 blobs present, while in Battlezone there may be up to 412 active blobs [Liang et al. 2016].

Tang et al. [2017] proposed an alternative count-based approach that, unlike the above methods, does not centre around factoring the observation. Instead, their idea was to hash the screen to a lower-dimensional space where it is feasible to track individual visit counts. As the authors note, one issue with this approach is that a naïve, pixel-based hash may not map strategically similar states to the same bucket. Therefore, instead of using an arbitrary hashing function, they use an autoencoder [Rumelhart et al. 1986] to

\[ \text{The term “strategically similar states” is our own. Throughout this chapter we use it to refer to states that are similar from a player point of view.} \]
map screens to a compressed, binary form. Unfortunately though, this approach turned out to be less successful on sparse reward Atari games than previous methods. Moreover, it is important to note that even this method is affected by extraneous visual details, such as moving objects in the game background. This is because the underlying autoencoder is trained to minimise the pixel-level reconstruction error, so its binary codes will capture visually prominent objects even if they are irrelevant to the agent’s decision making.

### 3.1.2 Prediction Error Methods

Besides count-based methods, another common way of encouraging exploration is to use the error of some trained predictive function as an intrinsic reward (Section 2.3.2). The intuition behind this idea is to reward the agent for experiencing “surprising” events.

Until very recently, this method had not been as successful in sparse reward Atari games as pseudo-count methods. Moreover, the published results that did exist were difficult to draw conclusions from. For example, Stadie et al. [2015] trained a task dynamics model (Section 2.3.2) over an autoencoder’s feature space, using the next state prediction error as an intrinsic reward. The corresponding agent failed to score at all in Montezuma’s Revenge, although it was only trained for 20 million frames (or 10% as long as Mnih et al.’s [2015] original DQN agent). For these reasons, we paid little attention to prediction error agents in our recent conference paper [Dann et al. 2019]. However, there have been some significant developments in this area since then:

- In a broad empirical study, Burda et al. [2018a] tried a similar approach to Stadie et al. [2015], except that they focused on learning from intrinsic rewards only. They ran long experiments over a wide range of domains, comparing intrinsic rewards derived from several different types of dynamics model. An interesting observation to come out of this work is that prediction error agents are drawn towards states where there is large uncertainty in the next state’s representation. For example, supposing an agent for Montezuma’s Revenge is on the threshold of entering a new room and there is some non-determinism injected into the emulator, e.g. via sticky actions [Machado et al. 2018a], the agent will not be able to predict with certainty which room it will be in next. Since the pixels will be completely different depending on which room is encountered, the agent’s pixel-wise prediction error (and hence the intrinsic rewards) will generally be large.

- In a subsequent paper, Burda et al. [2018b] noted that the stochastic transition problem can be avoided by calculating prediction errors based on the current observation, rather than the next one. All that is required of the prediction targets is that they should depend deterministically on the current state and be sufficiently challenging

---

³They also report some results with extrinsic rewards enabled, but stress that their agent was not tuned for this setting. On Montezuma’s Revenge and Venture, these results were only semi-competitive.
to model that the prediction errors do not quickly become zero. In practice, the authors found that the output of a randomly initialised convolutional neural network (CNN) constituted a suitable target. The resulting intrinsic rewards, named *random network distillation (RND) bonuses*, combined with some other modifications that we discuss later, led to a new high-score in *Montezuma’s Revenge*.

Relating this back to our broader discussion, there are two points to note: (1) At the end of their paper, Burda et al. [2018b] note that RND bonuses remain ill-suited to encouraging high-level exploration. For example, on the first screen of *Montezuma’s Revenge*, the player is faced with the high-level choice of exiting the room via the left door or the right door after collecting the key. Existing agents tend to form long-lasting preferences for one of the doors and rarely try the other option. To encourage exploration at this level, it seems necessary to reason over high-level subgoals, which only strengthens the motivation for our work in this chapter. (2) Despite the large differences in the underlying machinery of RND bonuses and the count-based methods discussed earlier, *RND bonuses remain just a proxy for rewarding visual novelty*. To see this, note that convolutional neural networks are continuous functions, so small pixel-level changes are unlikely to cause major changes in random feature space. Therefore, if the agent has learned to predict the random features accurately for one screen, it will not be particularly “surprised” by small perturbations of that screen. Conversely, new screens will typically cause a jump in the feature values, resulting in large bonuses until the agent has experienced similar screens many times.

### 3.1.3 Visual Novelty: Strengths and Weaknesses

In the game that has become emblematic of the sparse reward challenge – *Montezuma’s Revenge* – several types of visual novelty bonus have proven effective [Bellemare et al. 2016, Ostrovski et al. 2017, Martin et al. 2017, Burda et al. 2018b]. However, it is worth reflecting on why this is so. In each room of the dungeon, the pixels that represent stationary objects such as walls and ladders remain unchanging, while the few moving enemies that exist follow pre-determined paths (either straight lines or oscillations back and forth). For this reason, the vast majority of visual novelty is driven by the protagonist’s movement. Figure 3.2a shows a time lapse of the first room, generated under a uniform random policy. The oscillating skull generates a uniform blur at the bottom of the screen, indicating that no one skull position is especially novel. The protagonist, on the other hand, spends a disproportionate amount of time on the central ladder, a lesser amount of time at the bottom-right ladder, and barely any time near the top-left and top-right doors. The reason visual novelty bonuses work here is because they encourage the agent to even out this distribution, or in other words to explore the room.

While this explains the success of visual novelty schemes in *Montezuma’s Revenge*, it also hints at scenarios where the approach is likely to struggle: (1) When minor visual
SECTION 3.1: ISSUES WITH EXISTING INTRINSIC MOTIVATION SCHEMES

Figure 3.2: Time lapses from the first screens of (a) Montezuma’s Revenge and (b) Venture, generated over 10,000 frames. The initial states for both games are provided to the left for reference. The protagonist in Venture is highlighted by the red circle.

Differences between states convey important strategic differences. (2) When large visual differences arise from unimportant changes in the game state.

Minor visual difference $\Rightarrow$ important strategic difference

A clear example of a small visual detail conveying important information occurs on the first screen of the game Venture, where the protagonist is represented by a tiny purple dot (see inside the red circle in Figure 3.2b, which was inserted by us). Since the protagonist accounts for so few pixels, visual novelty bonuses will barely increase even if the agent navigates the protagonist to a rarely-explored area of the map.\(^4\)

A similar issue arises immediately after the player has collected the first key in Montezuma’s Revenge. In this case, visual novelty fails to capture the fact that collecting the key fundamentally changes the game state, such that states on the return leg from the key are fundamentally different from those on the approach. Since the protagonist will have generally spent more frames on the route to the key than at the location of the key, the agent sees returning to the spawn location as being relatively “boring” compared to staying near the bottom-left pedestal. Fortunately for such agents though, the problem can be shortcut by deliberately suiciding. Providing the player has lives in hand, dying causes the protagonist to return immediately to the spawn location with the key still in possession, from which point it is easy to obtain an extrinsic reward by opening a door.\(^5\)

However, reinforcement learning agents will only find this solution if episodes are deemed not to terminate upon life loss. Roderick et al. [2018] cover this issue in depth and show

---

\(^4\)Experts may point out that visual novelty bonuses are nonetheless helpful in Venture [Ostrovski et al. 2017, Martin et al. 2017]. However, the real reason for this is that the screen undergoes a large visual change when the player enters one of the four rooms (corresponding to the open shapes in the four quadrants of Figure 3.2b), which encourages the agent to explore them.

\(^5\)Videos of Bellemare et al.’s [2016] agent and Ostrovski et al.’s [2017] agent taking this shortcut are available at https://youtu.be/0yI2wJ6F8r0 (skip to 0:51) and http://youtu.be/232tOUPKPoQ.
that an agent based on the intrinsic reward scheme of Bellemare et al. [2016] fails to exit the first room when the game is modified so that the player only receives a single life.

**Unimportant strategic difference \(\Rightarrow\) large visual difference**

The converse to the aforementioned issue in *Venture*, where the protagonist accounts for very few pixels, is that the first screen also contains three much larger enemies. As such, the agent will receive relatively large visual novelty bonuses if the enemies move to rare positions. Since the agent has no control over the enemy movements, this merely adds noise to the learning process. The same problem occurs in the game *Freeway*, where the player-controlled chicken accounts for few pixels compared to the moving vehicles on the highway. Martin et al.’s [2017] agent struggled in this game as a result, becoming “overawed” by the novelty bonuses arising from the continually changing traffic.

### 3.1.4 The “Risky Exploration Problem”

In videogames where the player has a certain number of lives, the question arises of what should constitute the end of an episode: (A) When the player loses a life, or (B) When the player has lost all lives. In most work on *Atari* games to date, including the original DQN paper [Mnih et al. 2015], choice (A) was made. This setting usually yields better performance, as it helps the agent learn to avoid life loss. However, terminating episodes upon life loss was heavily detrimental to Bellemare et al.’s [2016] agent in *Montezuma’s Revenge*, where it achieved far higher scores under setting (B). This phenomenon is unlikely to be unique to Bellemare et al.’s agent, although it is difficult to verify this claim because ignoring life loss has since become the default setting for exploration-oriented agents.

Besides the fact that terminating episodes upon life loss prevents the agent from learning suicide-based shortcuts, as described above, another issue is that the intrinsic rewards paid under both types of scheme we have covered so far (count-based and prediction error) are always non-negative. Accordingly, staying alive indefinitely at a heavily-visited location is generally preferable to terminating the episode. On tasks that require the agent to risk life loss in order to learn, such as jumping over the first skull in *Montezuma’s Revenge*, terminating episodes upon life loss may thus have an anti-exploratory effect. For the remainder of this chapter, we refer to this issue as the “risky exploration problem”. Addressing the risky exploration problem is not as simple as recentring the intrinsic rewards so that they are sometimes negative, since this may teach the agent to suicide from negative reward states. Terminating episodes after all lives are lost does not entirely remove the problem either, since the agent will still become risk averse on its final life. Moreover, the player may only receive a single life. Burda et al. [2018b] propose a more general approach to this issue, whereby the agent maintains separate Q-values for the intrinsic and extrinsic components of the reward. The intrinsic Q-values ignore episode termination altogether, with the intuition that all one really loses upon episode
termination in a videogame (from a pure exploration perspective) is the time invested to reach that point. However, the agent may still become excessively risk averse under this approach, because in most videogames the respawn point will be a heavily explored state and thus episode termination may still incur a significant loss of intrinsic reward.

3.2 Our Approach

In this section, we introduce a new, autonomous method for identifying subgoals in domains with high-dimensional state representations. Following this, we propose an intrinsic reward scheme for exploiting the derived subgoals. Similarly to hash-based approach of Tang et al. [2017], our subgoal identification method aims to partition the state space into regions of strategically similar states. However, our approach is designed to yield far fewer partitions than their method, such that the partitions can be treated as abstract subgoals. It is also designed to give more emphasis to important visual details, such as the tiny purple dot in Venture, compared to less important details, such as distant enemy sprites. To achieve this, we introduce a heuristic distance measure called exploration effort. We overview the partitioning algorithm in Section 3.2.1, then provide a detailed explanation of exploration effort in Section 3.2.2. Finally, in Section 3.2.3, we introduce pellet rewards, an intrinsic reward scheme that aims to mitigate the risky exploration problem.

3.2.1 State Space Partitioning via a Measure

The idea of deriving subgoals by partitioning the state space is an old and well-known one [Dietterich 2000, Hengst 2002, Menache et al. 2002, Mannor et al. 2004, Şimşek et al. 2005]. However, classical approaches typically rely on the state space being so small that it is possible to derive partitions by performing an analysis of the task structure, or by examining individual states’ visit counts. Since such analysis is not practical in videogames, we take a simpler approach: We assume knowledge of some distance measure over states, \( d : \mathcal{S} \times \mathcal{S} \to \mathbb{R}^+ \cup \{0\} \), and use it to group nearby states together. Shortly, we propose a specific heuristic distance measure that is designed to yield partitions containing strategically similar states. For now though, merely note that there are multiple ways by which one can derive partitions from a distance measure. One such method is sketched as follows:

**Partition via a fixed radius:** Create a node at the initial state and define a fixed partition radius. As soon as the agent encounters a state outside this radius, add another node at that point. Keep adding nodes whenever the current state is outside all existing nodes’ radii. The nodes can be thought of as “representative states” for the set of partitions. The partition, \( p \), to which a state belongs is the one whose representative state, \( s_p \in \mathcal{R} \), is closest.
Unfortunately, preliminary experiments revealed the efficacy of this algorithm to be highly sensitive to the partition radius. Without careful tuning, it frequently yielded too few or far too many partitions. Therefore, we opted for the following alternative, which fixes the number of partitions that exist at any given time and does not require a partition radius:

**Partition via a schedule:** Create a node at the initial state then act according to some policy for a number of time steps. During this period, keep track of the farthest state discovered from the set of existing nodes. Periodically add that state to the set of nodes and restart the process. Again, treat the nodes as representative states for partitions.

A more formal description of the latter method can be found within the pseudocode for our full approach, presented at the end of this chapter (Algorithm 3).

### 3.2.2 Exploration Effort (EE)

Under the approach outlined so far, different partitionings of the state space may promote exploration to differing degrees. For example, consider the game *Freeway*, where the aim is to navigate a chicken to the other side of a busy road. If one were to partition the state space via a visual dissimilarity measure, the resulting partitions would most likely contain states with similar traffic configurations, as traffic accounts for most of the visual variety in *Freeway*. In that case, rewarding the agent for reaching rarely-visited partitions would be unlikely to help, as the player has no control over the traffic. On the other hand, if the state space were partitioned according to the chicken’s position, the agent would be incentivised to reach the rarer, middle-of-the-road and top-of-the-road positions.

Based on this reasoning, it seems we require a distance measure that regards states like top-of-the-road and bottom-of-the-road positions in *Freeway* as being far apart. Indeed, from an exploration perspective, there is an important sense in which such states truly are “far apart”: During the early stages of training, a decaying $\epsilon$-greedy policy is unlikely to take the chicken from the bottom of the road to the top, because the policy will be near-uniform random at this stage and thus unlikely to oversample *up* actions sufficiently. This suggests that we can derive a suitable distance measure from the amount of action over/undersampling required to transition between states.

To be clear, we do not claim that this an appropriate heuristic for all problems. For example, in a “combination safe” task where only one sequence of actions will successfully open the safe, the heuristic described does not provide a meaningful measure of progress. However, a broad problem class to which the heuristic does seem well-suited is domains where the agent is controlling a physical entity. Similar intuition underlies the use of autocorrelated noise in continuous control tasks [Wawrzynski 2015].

42
Formal Definition of EE

Continuing the above line of thought, we seek a function $E : S \times S \to \mathbb{R}^n$ that takes two states as input and returns, for each of the $n$ actions available, a measure of how much that action must be oversampled (relative to the agent’s current policy, $\pi$) in order to transition from the former state to the latter. To this end, we begin by defining an $n$-dimensional auxiliary reward scheme. At each time step, the auxiliary reward vector is:

$$\hat{r}^\pi(s,a) = \kappa \langle \hat{r}^\pi_1(s,a), \hat{r}^\pi_2(s,a), \ldots, \hat{r}^\pi_n(s,a) \rangle \quad (3.2)$$

where $\kappa > 0$ is a reward scaling factor and

$$\hat{r}^\pi_i(s,a) = \begin{cases} 1 - \pi(s,a_i), & \text{if } a = a_i \\ -\pi(s,a_i), & \text{if } a \neq a_i \end{cases} \quad (3.3)$$

The form of the reward is deliberately chosen so that the expected reward vector under the agent’s current policy is zero. However, if an action is over/undersampled for a period of time, the sum of rewards in the corresponding dimension will be positive/negative. To clarify this point, we now provide a worked example:

**Example 1.** Let $A = \{\text{up, down, left, right}\}$, and suppose that an agent transitioned from $s_0$ to $s_4$ by pressing up, right, right, down under a uniform random policy. Let $\gamma = 0.99$ and $\kappa = 1$. Then:

$$\hat{r}^\pi(s_0,a_0) = \langle \frac{3}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4} \rangle$$
$$\hat{r}^\pi(s_1,a_1) = \langle -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, \frac{3}{4} \rangle$$
$$\hat{r}^\pi(s_2,a_2) = \langle -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, \frac{3}{4} \rangle$$
$$\hat{r}^\pi(s_3,a_3) = \langle -\frac{1}{4}, \frac{3}{4}, -\frac{1}{4}, -\frac{1}{4} \rangle$$

Thus the sampled (Monte Carlo) auxiliary return is:

$$\hat{r}^\pi(s_0,a_0) + \gamma \hat{r}^\pi(s_1,a_1) + \gamma^2 \hat{r}^\pi(s_2,a_2) + \gamma^3 \hat{r}^\pi(s_3,a_3)$$
$$= \langle 0.015, -0.015, -0.985, 0.985 \rangle$$

This reflects the fact that the left action was undersampled while right was oversampled. △

Continuing in this vein, we define the *exploration effort function* as follows:

**Definition 1.** The exploration effort from $s$ to $s'$ (under time limit $m$) is the expected auxiliary return when the agent transitions from $s$ to $s'$ within $m$ steps, via policy $\pi$:

$$E^\pi_m(s,s') = \mathbb{E}_\pi \left[ \sum_{k=0}^{T-1} \gamma^k \hat{r}^\pi(s_k,a_k) \mid s_0 = s, s_T = s', T < m \right] \quad (3.4)$$

△

---

$^6$We assume in this work that the action space is discrete.

43

(May 26, 2019)
In the absence of prior knowledge, the auxiliary rewards in Equation 3.3 have zero expectation under $\pi$. However, knowledge of $s'$ means that the expectation in Equation 3.4 may be non-zero. Returning to the example in Freeway, if we knew that the chicken was at the bottom of the road in $s$, then at the top of the road in $s'$, we would expect the interleaving auxiliary return in the up dimension to be positive (assuming $\pi$ does not already favour the up button).

To train an estimator for the exploration effort function, we sample state pairs from the agent’s replay memory that are less than $m$ time steps apart, as well as the interleaving auxiliary rewards. In addition to calculating Monte Carlo targets, as per Example 1, we also calculate one-step targets as follows:

$$E_{\pi,m}(s_t, s') \text{one-step target} = \hat{r}(s_t, a_t) + \gamma E_{\pi,m-1}(s_{t+1}, s')$$

This allows us to train towards a mixed Monte Carlo target [Bellemare et al. 2016], using proportion $\eta$ of the Monte Carlo target and proportion $(1 - \eta)$ of the one-step target. Using a mixed return helps mitigate the variance of the Monte Carlo estimates, while also propagating distant rewards faster than pure one-step updating. To avoid maintaining separate estimators for $E_{\pi,m}$ and $E_{\pi,m-1}$, we bootstrap one-step targets from $E_{\pi,m}$, with the justification that the functions should be very similar for large enough $m$.

### Deriving a Distance Measure from EE

In certain situations, the exploration effort between strategically similar states may be large. For example, suppose that an agent is playing Montezuma’s Revenge via a uniform random policy and that the protagonist was positioned against a wall in $s$, then later against the same wall in $s'$. Given this knowledge, it is likely the policy oversampled actions that ran the protagonist into the wall versus those that would have escaped it. Therefore, simply treating the magnitude of the exploration effort vector as distance is inappropriate. In the worst case, it could even cause the state partitioning algorithm to generate duplicate representative states.

Fortunately, we can address this issue via a mathematical method. Let $\hat{s} \in S$ be an arbitrary reference point. Then, define the distance between $s$ and $s'$ relative to $\hat{s}$ as:

$$d_{\pi,m}(s, s') = \max(||E_{\pi,m}(\hat{s}, s) - E_{\pi,m}(\hat{s}, s')||, ||E_{\pi,m}(s, \hat{s}) - E_{\pi,m}(s', \hat{s})||)$$

To paraphrase, this measure finds the displacement of both $s$ and $s'$ from the reference point, treating the magnitude of the difference as distance. Observe that the distance from a state to itself is always zero, and the two-way maximum ensures that the measure is invariant to the order of the arguments. Taking this one step further, we define the distance between $s$ and $s'$ relative to a reference set, $\hat{S} \subseteq S$, as:

$$d_{\pi,m}(s, s') = \max_{\hat{s} \in \hat{S}} d_{\pi,m}(\hat{s}, s')$$
In our implementation we use this measure, with \( \hat{S} \) equal to the set of representative states, \( \mathcal{R} \). For clarity, we now provide a detailed worked example of this calculation:

**Example 2.** Let \( \mathcal{A} = \{ \text{up, down, left, right} \} \) and suppose that there are currently two state space partitions, with \( \mathcal{R} = \{ s_{p_1}, s_{p_2} \} \). Further suppose that the exploration effort measures between all relevant state pairs are as follows:

\[
\begin{align*}
\mathcal{E}_m^\pi(s_{p_1}, s) &= (1, 0, -2, 1), & \mathcal{E}_m^\pi(s_{p_1}, s') &= (1, 1, -3, 0) \\
\mathcal{E}_m^\pi(s, s_{p_1}) &= (-1, 1, 3, 0), & \mathcal{E}_m^\pi(s', s_{p_1}) &= (-1, -1, 2, 1) \\
\mathcal{E}_m^\pi(s_{p_2}, s) &= (-4, -2, 1, 1), & \mathcal{E}_m^\pi(s_{p_2}, s') &= (-4, -1, 1, 1) \\
\mathcal{E}_m^\pi(s, s_{p_2}) &= (3, 3, 0, -1), & \mathcal{E}_m^\pi(s', s_{p_2}) &= (3, 2, -2, 1)
\end{align*}
\]

Then the distances between \( s \) and \( s' \) relative to \( s_{p_1} \) and \( s_{p_2} \) are:

\[
d_{\pi,m}^{s_{p_1}}(s, s') = \max(\|\mathcal{E}_m^\pi(s_{p_1}, s) - \mathcal{E}_m^\pi(s_{p_1}, s')\|, \|\mathcal{E}_m^\pi(s, s_{p_1}) - \mathcal{E}_m^\pi(s', s_{p_1})\|)
\]

\[
= \max((1, 0, -2, 1) - (1, 1, -3, 0), \|(-1, 1, 3, 0) - (-1, -1, 2, 1)\|)
\]

\[
= \sqrt{6} \text{ (using the L2 norm)}
\]

\[
d_{\pi,m}^{s_{p_2}}(s, s') = \max(\|\mathcal{E}_m^\pi(s_{p_2}, s) - \mathcal{E}_m^\pi(s_{p_2}, s')\|, \|\mathcal{E}_m^\pi(s, s_{p_2}) - \mathcal{E}_m^\pi(s', s_{p_2})\|)
\]

\[
= \max((3, 3, 0, -1) - (3, 2, -2, 1), ||(0, 1, 2, 2)||)
\]

\[
= 3
\]

Finally, distance between \( s \) and \( s' \) relative to the reference set \( \mathcal{R} \) is:

\[
d_{\pi,m}^\mathcal{R}(s, s') = \max_{s' \in \mathcal{R}} d_{\pi,m}^s(s, s') = \max(d_{\pi,m}^{s_{p_1}}(s, s'), d_{\pi,m}^{s_{p_2}}(s, s')) = 3
\]

\[\triangle\]

**A Note on Possible Alternative Formulations of EE**

In essence, the exploration effort function is designed to measure how hard it is for the agent to transition from one state to another under the current exploration policy. Given this intuition, it might seem that a statistical measure, such as the Kullback-Leibler (KL) Divergence [Kullback and Leibler 1951], would be a more natural way of calculating the difference between the sampled action distribution and the expected distribution. Initially, we did in fact try this approach. However, a problem with the idea is that it fails to take states’ temporal separation into account. For example, suppose that \( s \) and \( s' \) are successive states and that due to the appearance of a bullet in \( s' \), it is clear that the agent pressed the “fire” button in \( s \). In this case, the KL divergence between the sampled distribution (where
the probability of the “fire” action is 1) and the expected distribution may be extremely large, even though the states are clearly close in an exploration sense. To address this, we considered multiplying the KL divergence by the number of time steps separating the states. However, this value is unbounded, which makes it problematic to use as a training target. Two big advantages of formulating EE as a return are that (A) it is bounded for $\gamma < 1$ and (B) we can calculate one-step Bellman updates, which significantly reduces the variance of the training targets.

3.2.3 Pellet Rewards

The approach presented thus far does in fact yield partitions for sparse reward Atari games that resemble intuitive subgoals. (Skip to Figure 3.4 for a preview.) However, we have not yet shown how these partitions can be exploited, nor demonstrated that such an approach actually aids exploration. To enable us to address these points, in this section we propose a partition-based intrinsic reward scheme, dubbed pellet rewards. Our approach takes its name and inspiration from the collectable pellets in Ms. Pacman that, from an exploration viewpoint, have a number of desirable effects:

1. They compel the player to perform an approximate depth-first search of the environment, as it is generally more efficient to continue forward rather than to retreat over a trail of consumed pellets. However, the player is in no way disincentivised from backtracking upon hitting a dead-end.

2. The player receives no incentive to remain in the same state (unlike under schemes where the agent receives a bonus after every time step).

3. The player’s incentive to avoid death depends on how many pellets have already been collected. Once the player has collected all low-risk pellets, there is no reason not to attempt high-risk pellets.

We mimic the pellet mechanic by paying the agent a novelty bonus on its first transition (per episode) into a partition. Visited partitions become “collected”, meaning no further novelty bonus is paid for reaching those partitions until the next episode. Similarly to previous schemes [Bellemare et al. 2016, Martin et al. 2017, Ostrovski et al. 2017], the bonus takes the form $\beta / \sqrt{n_p}$, where $\beta$ is a constant scale factor and $n_p$ is partition’s visit count. However, instead of counting how many frames the agent has spent in the partition, we count how many times the corresponding pellet has been collected.

One downside of creating partitions on a schedule is that sometimes the algorithm creates seemingly redundant partitions in regions that have already been thoroughly explored. To avoid generating destabilising novelty bonuses in such cases, we calculate $n_p$ as an inferred visit count. That is, we calculate a moving average of the partition’s visit rate, $r_p$, and set $n_p = r_p \times totalEpisodeCount$. This ensures that new partitions that
are frequently visited quickly attain large counts. Further, to ensure that visit counts are non-decreasing, we lower bound $n_p$ to its previous value. Asymptotically, this means the bonuses for pellets with non-zero collection rates decay to zero, ensuring that they do not have a large, ongoing distortion on the task objective.

### 3.3 Experiments

To test our algorithm’s ability to identify strategically meaningful subgoals, and to gauge the effectiveness of the resultant pellet rewards in aiding efficient exploration, we applied our approach to a selection of *Atari* games. In this section, we describe the configuration of our agent (Section 3.3.1), present the results of our experiments (Section 3.3.2) and then, based on our findings, position the efficiency and overall strength of our agent against previous methods (Sections 3.3.3 and 3.3.4).

#### 3.3.1 Configuration

In this section, we focus on explaining non-standard or otherwise pertinent aspects of our configuration for *Atari* games that warrant some comment. All other settings can be found in our source code\(^7\).

**Environment.** We used version 0.6 of the *Arcade Learning Environment (ALE)* [Bellemare et al. 2013], injecting stochasticity via *sticky actions* [Machado et al. 2018a] with a stickiness of 0.25. To expose the agent to “risky exploration” scenarios (as explained in Section 3.1.4), we considered episodes to be terminated upon life loss.

**Architecture.** Since the exploration effort function takes two screens as input and acts as a kind of similarity measure, we approximated it via a *Siamese* architecture [Bromley et al. 1994]. First, the screens are passed through two parallel encoders, whose weights are shared. The structure of the encoders is identical to the combined preprocessing and convolutional section of Mnih et al.’s [2015] network, except that for computational efficiency we reduce the number of convolutional filters from (32, 64, 64) to (16, 16, 16) and input only a single screen rather than a four frame history. The encoders feed into a layer that concatenates the mean of the encoder outputs with their difference. The intuition behind this is that exploration effort between states should mostly depend on the states’ differences (e.g. differences in the position of the protagonist in *Montezuma’s Revenge*). However, keeping only the difference in encodings would throw away some information, so we concatenate it with the encoders’ mean output. The concatenated layer feeds into a fully connected layer of 128 units, followed by an output layer with one node per action.

\(^7\)https://bitbucket.org/mchldann/aaai2019
Our Q-network architecture matches that of Mnih et al. [2015], except that we additionally feed a representation of each pellet’s collection status to the fully connected hidden layer. This is essential, as otherwise the agent has no way of knowing whether or not it should move on from a pellet. Rather than using a binary representation, we note that low-value pellets ought to have less effect on Q-values than high-value pellets. Also, if the agent has settled on a successful trajectory, but then a new pellet somehow spawns well away in an essentially irrelevant part of the state space, then this should not destabilise the agent’s policy. To achieve both of these properties, we represent uncollected pellets via zeroes and collected pellets via their reward values. Compared to using a binary representation, we found that this greatly improved stability late in training.

Exploration Effort Training Parameters. The auxiliary reward scale factor, $\kappa$, was set to 1. The time separation constant, $m$, was set to 100. Both the EE and Q-function were trained via mixed Monte Carlo updates with $\eta = 0.1$. To further mitigate the large variance of the EE targets, the Monte Carlo deltas were clipped to 0.5. Prior to commencing Q-learning, the EE function was trained for 8 million frames (2 million samples) on experience generated via a uniform random policy. From that point on, it was trained in tandem with the Q-function.

Partition / Pellet Configuration. Pellet rewards were calculated using partition visits at sample time rather than at the time of insertion into the replay memory. The pellet reward scale factor, $\beta$, was set to 1. The time between partition additions was initially set to 80,000 frames, then increased by 20% with each addition. Experience collection for Q-learning commenced once there were 5 partitions in existence.

Improved Baseline Exploration. The usual $\epsilon$-greedy decay schedule, whereby $\epsilon$ is annealed to a small value over the first 4 million frames [Mnih et al. 2015], risks “dooming” agents that have discovered few rewards by the end of the schedule. In essence, favouring the greedy action 90+\% of the time before the agent has learned anything is too committal and may harm the agent’s chances of ever learning. To mitigate this issue, we fix $\epsilon = 1$ for the remainder of the episode whenever the agent exceeds 500 actions without receiving a reward (either a pellet or an extrinsic reward). Our rationale is that DQN’s default discount of $\gamma = 0.99$ effectively limits the agent’s planning horizon to around 500 actions (as $0.99^{500} \approx 0.01$). Assuming there is some way to achieve a positive return within this time frame, rewardless trajectories of greater length indicate that the agent may not have explored the current region sufficiently. As we shall see in the next section, this tweak yields a more competitive baseline and helps contextualise past agents’ results.
3.3.2 Results

To quantify the impact of introducing pellet rewards, we applied our approach to five Atari games, benchmarking against a configuration that was identical in every respect except that pellet rewards were turned off. Three of the games chosen had sparse rewards (Venture, Freeway and Montezuma’s Revenge) while two had dense rewards (Battlezone and Robot Tank). The reason for including the latter games was not to see whether pellet rewards would be beneficial, but rather to see if they would be detrimental in games where standard DQN already performs well. In previous work [Ostrovski et al. 2017], count-based novelty bonuses had an adverse affect in the two games chosen.

Results for Sparse Reward Games

Training curves for both agents are provided in Figure 3.3. In each graph, the thick lines represent average scores, while the thin lines correspond to individual training runs, of which we conducted five per agent. To make it easier to see the effect of pellet rewards in isolation, we have plotted all training curves from the point where Q-learning commenced. However, it should be noted that the pellet rewards agent was given an additional 8 million frames to pre-train the exploration effort function, plus a further 460,000 frames to generate the first 5 partitions. For Freeway, the graph’s time scale has been shrunk to emphasise the early learning phase. After 30 million frames, the agents’ average scores were virtually identical (pellet rewards: 33.4, baseline: 33.3).

In all three sparse reward games, our subgoal identification method found meaningful subgoals. In Venture, it placed subgoals in all the main rooms. In Freeway, it generated representative states resembling intermediate waypoints for the chicken crossing the road. For Montezuma’s Revenge, the first 20 representative states found over a particular run are shown in Figure 3.4. Like the human experts in Kulkarni et al.’s [2016] work, our method generated subgoals corresponding to the bottom-right ladder (#2 and #6), bottom-left ladder (#8), top-right door (#5), top-left door (#3) and middle-ladder (#4 and #7).

The resultant pellet rewards clearly aided extrinsic reward discovery across all three games. In Venture, all runs of the pellet agent reached an average of ≈1000 points by 15 million frames, while the baseline agent’s training curves were slower and more staggered, indicating that it relied more on luck to discover extrinsic rewards. By 30 million frames, one of the baseline runs remained stuck on zero score. Results in Freeway were similar, with the baseline exhibiting staggered starts while the pellet agent progressed so reliably that its five runs are virtually indistinguishable in Figure 3.3b.

On Montezuma’s Revenge, none of the baseline runs made progress within the time given. By contrast, the pellet rewards agent learned to reach the key, open the right-hand door, descend a ladder to reach a sword then climb back up the ladder to kill an enemy with the sword for a total of 2,500 points, achieving this consistently in 4 out of 5 runs.
Figure 3.3: Training scores on sparse reward games (based on a full set of lives, averaged over the last 100 episodes). Thick lines indicate average scores, thin lines represent individual training runs.
by the end of training. Moreover, despite episodes being terminated upon life loss, the pellet rewards agent did not become excessively afraid of the enemy skull. While the agent never learned how to reach the key then return to a door within its first life, we did witness it perform both these acts within a single life when the skull had been removed by colliding with it previously. This suggests that the agent was not discouraged from returning to “visually boring states” after reaching a dead-end. Our results under episode termination with life loss are in stark contrast with those of Bellemare et al. [2016], whose A3C+ agent averaged only 143 points under this setting, even after significantly more training time. To contextualise this score, note that the player receives 400 points in total for exiting the first room (100 for reaching the key, plus 300 for opening a door).

One surprising result is the performance of the baseline in Venture, as baselines in

\[\text{by the end of training}^8.\] Moreover, despite episodes being terminated upon life loss, the pellet rewards agent did not become excessively afraid of the enemy skull. While the agent never learned how to reach the key then return to a door within its first life, we did witness it perform both these acts within a single life when the skull had been removed by colliding with it previously. This suggests that the agent was not discouraged from returning to “visually boring states” after reaching a dead-end. Our results under episode termination with life loss are in stark contrast with those of Bellemare et al. [2016], whose A3C+ agent averaged only 143 points under this setting, even after significantly more training time.\textsuperscript{9} To contextualise this score, note that the player receives 400 points in total for exiting the first room (100 for reaching the key, plus 300 for opening a door).

One surprising result is the performance of the baseline in Venture, as baselines in

8On the other training run, the agent formed a hard-to-unlearn preference for exiting the first room via the left door, from where it is difficult to score. Bellemare et al.’s [2016] agent also experienced this issue; see Figure 2 in their work, noting the minimum score line that persists until around 60 million frames.

9Their A3C+ agent was trained for 290 million frames; ours was trained for 30 million frames. They do not report results for their DQN-based agent with episode termination upon life loss, but imply that it was also harmed by this setting.
previous work average less than 100 points in this game [Bellemare et al. 2016, Martin et al. 2017, Ostrovski et al. 2017]. The fact that our baseline began improving well after 4 million frames (i.e. after $\epsilon$ had fully decayed) suggests that premature annealing of $\epsilon$ was a significant problem for previous agents.

Results for Dense Reward Games

On the dense reward games we tested (Battlezone and Robot Tank), pellet rewards had no discernible impact (see Figure 3.5). To be clear, this is not a bad outcome. In dense reward games, it is expected that agents will improve without the need for exploration bonuses. Given this fact, a potential issue with exploration bonuses is that they may distract the agent and thus decrease the extrinsic return. From this standpoint, one advantage of pellet rewards is that the total bonus paid to the agent is relatively small.

Figure 3.5: Training scores on dense reward games.
Under DQN-style reward clipping, most extrinsic rewards have a value of 1. In contrast, after only 100 visits, a pellet’s value decreases from 1 to \((1 \div \sqrt{100}) = 0.1\). Moreover, our partition creation schedule limits the number of pellets in existence. (In our experiments, there are only 24 partitions created by the end of training.) Consequently, in dense reward games, the return contribution of the pellets is typically dwarfed by that of the extrinsic rewards. On the other hand, the PixelCNN bonuses of Ostrovski et al. [2017] were heavily detrimental in Battlezone and Robot Tank, reducing scores by 30–50% over the course of training\(^{10}\). For this to have occurred, the return contribution of the PixelCNN bonuses must have been significantly larger than that of our scheme.

### 3.3.3 Efficiency Compared to Existing Subgoal-Based Methods

Prior to our work, there have been other attempts to derive and exploit subgoals in Atari, although these have generally been inefficient in one or more of the following ways:

- The subgoal identification algorithm is sample inefficient.
- The subgoal identification algorithm identifies some seemingly useful subgoals, but many others that are redundant.
- The agent exploits subgoals to learn “skills” (Section 2.4) in tandem with the task, but this simultaneous learning does not synergise well.

Reflecting on our experimental results, in this section we position our approach against previous methods in each of these regards.

#### Sample Efficiency of Subgoal Identification

As far as we are aware, the only previous agent to have achieved a competitive score on Montezuma’s Revenge by leveraging autonomously derived subgoals is Vezhnevets et al.’s [2017] feudal networks (FuN) agent. Briefly, a feudal network is a composite neural network architecture that contains two modules: a worker and a manager. The worker receives an intrinsic reward for following subgoals set by the manager, while the manager learns to set subgoals with the aim of maximising the extrinsic return. Similarly to our agent, the FuN agent learned seemingly natural subgoals for the first screen of Montezuma’s Revenge and eventually reached an average score of around 2,000 points. However, it took roughly 500 million frames to reach this mark, while our agent took only \(\approx 25\) million frames, making ours more sample efficient by over an order of magnitude.

\(^{10}\)See Figure 17 in their appendix, noting that the agents with mixed Monte Carlo returns are most comparable to ours.
Selectivity of Subgoals Identified

One other method that has recently shown promise in identifying subgoals in Atari is that of eigenoptions [Machado et al. 2017; 2018b]. Recall from the background that one of the most common approaches to autonomous subgoal identification is the bottleneck method (Section 2.4.2). One way of characterising bottlenecks is in terms of diffusive information flow, or the way in which the agent’s state occupancy flows from an initial state to downstream states over time. For example, in a gridworld containing two rooms separated by a door, suppose that the agent starts at the centre of one room and executes a uniform random policy for a short period of time. Now suppose that this experiment is repeated many times, and a heat map is generated based on the agent’s final location. Roughly speaking, the heat map will indicate a “hot spot” in the initial room, centred on the agent’s start location, and a cool zone in the second room, since the agent is unlikely to navigate through the door by chance. The cut-off between hot and cool zones evidences the fact that the doorway is a bottleneck. This type of approach was applied to Atari by Machado et al. [2017], and later extended to work with raw pixel representations [Machado et al. 2018b] by using a specific type of diffusion model known as the successor representation [Dayan 1993]. Both agents were capable of learning options autonomously on the first screen of Montezuma’s Revenge, although to the best of our knowledge they did not perform well on the actual game. Furthermore, Machado et al.’s [2018b] agent identified hundreds of options for the first room alone. While our agent did identify some seemingly redundant subgoals (e.g. see subgoals #9 and #10 in Figure 3.4), it was far more selective than the eigenoption method overall.

Explicit Versus Implicit Skill Training

Under classical approaches to hierarchical RL, such as the options framework (Section 2.4.1) and MAXQ [Dietterich 2000], the agent explicitly learns “skills”, or skill-like subpolicies. This is also true of several recent subgoal-learning methods applied to Atari [Vezhnevets et al. 2017, Machado et al. 2017; 2018b]. Unfortunately, one downside of this approach is that it potentially hampers sample efficiency, because learning subpolicies may not synergise well with learning the actual task. With this in mind, we opted for an implicit kind of subpolicy learning instead: By feeding a representation of the pellets’ collection statuses to the Q-network’s hidden layer, we are effectively treating the Q-network as a universal value function approximator (UVFA). As explained in Section 2.4.3, UVFAs can be thought of as encapsulating multiple subpolicies.

To illustrate the relationship between our approach and explicit skill training, suppose that we take the representative states from Figure 3.4 and edit the pellet representation so that all pellets except for the fifth one are seen as having been collected by the agent. Provided the agent has learned how to reach this pellet, it will be inclined to head towards
the top-right door (where the fifth pellet is located). While this is not quite the same thing as having a skill for reaching the top-right door, it is clearly related, and it has the advantage that the subpolicy is naturally learned in tandem with the task itself.

### 3.3.4 Comparison Against Non-Subgoal-Learning Agents

In terms of maximising the game score, agents based on per-frame visual novelty bonuses are currently state-of-the-art. However, as Burda et al. [2018b] note, these methods do not address the challenge of achieving high-level exploration, which is one of the key motivations for subgoal-based methods. For this reason, comparing headline scores across agent types is not strictly fair; subgoal-learning methods are designed not only to maximise the game score, but also to learn high-level information about the task structure. Nonetheless, we believe that comparing our agent’s results against those of non-subgoal-learning agents is useful for context. To this end, a case-by-case comparison is provided below:

- Until recently, Bellemare et al.’s [2016] agent based on the CTS density model held the record mean score on Montezuma’s Revenge of 3439 points after 100 million training frames. However, the same agent reached only $\approx 350$ points in Venture, which is far inferior to our agent’s score of $\approx 1200$ points after only 30 million frames.

- Ostrovski et al.’s [2017] agent, based on the PixelCNN density model, attains comparable scores to our approach on the three sparse reward games tested ($\approx 1000$–1200 on Venture, $\approx 30$ on Freeway and $\approx 1500$–3000 on Montezuma’s Revenge).

- As noted earlier in this chapter, Burda et al.’s [2018b] agent based on RND bonuses now holds record high-scores on Montezuma’s Revenge and Venture. In Montezuma’s Revenge, their strongest agent achieved a mean score of 10,070 by the end of training, while a separate configuration that was slightly weaker overall managed to finish the first level on one occasion, scoring 17,500 points. However, there are a couple of qualifying remarks that must be made about this agent: Firstly, it made use of 1,024 parallel actors, which generated roughly 16 billion frames of experience per training run. Secondly, their agent was tailored towards games with sparse rewards by changing the discount from its usual value of 0.99 to 0.999. Ablative analysis indicates that this change had a large impact. Extracting midway scores from their training curves is difficult, but a rough comparison with the agents above leaves us sceptical about whether RND bonuses are actually more effective than CTS or PixelCNN bonuses. Their 30 million frame score in Venture appears similar to ours, while in Montezuma’s Revenge their score appears marginally better.
• Tang et al.’s [2017] agent, which hashes the state space in order to track individual counts, is one of the most similar agents to ours.\footnote{We class their method as a non-subgoal-learning approach though, because the hashed state space is too granular for the partitions to be thought of as encoding high-level objectives.} However, even when using hand-designed features to perform the hashing, their agent’s 50 million frame scores (238 on Montezuma’s Revenge and 616 on Venture) are significantly worse than our agent’s 30 million frame scores.

As this comparison shows, our agent’s scores are not state-of-the-art, but they are competitive. We stress that the most notable aspect of our results is that they were achieved via high-level bonuses derived from autonomously identified subgoals, an approach that may aid in achieving high-level exploration in the future. Finally, it is worth noting that the agents cited above do not consider episodes terminated upon life loss. As far as we are aware, our results in Montezuma’s Revenge are state-of-the-art under this setting.

\section*{3.4 Further Related Work}

At the time of writing, exploration in sparse reward environments has become a very hot topic in deep reinforcement learning. Numerous new papers are appearing at conferences and in pre-print every month, so attempting to cover all related work is nearly impossible. Throughout this chapter, we have focused on methods that are similar in style and scope to ours. However, there are several other approaches worth mentioning that have not fit into our discussion thus far:

• Prior to Bellemare et al.’s [2016] breakthrough work on pseudo-counts, there were several early attempts to improve DQN’s exploration [Osband et al. 2016, Machado et al. 2015, Stadie et al. 2015]. However, while these methods had some success in games with an intermediate level of reward sparsity (e.g. Frostbite and Hero), to the best of our knowledge there were no agents prior to Bellemare et al.’s that had notable success on the most challenging games, such as Montezuma’s Revenge. A recent extension to the uncertainty-based approach of Osband et al. [2016] draws some interesting links between RND bonuses and Bayesian inference [Osband et al. 2018]. An agent based on this new work made some progress on Montezuma’s Revenge, albeit less than agents driven by explicit exploration bonuses.

• From an engineering standpoint, it is of course easier to tackle reward sparsity by guiding the agent towards a known solution. In this vein, Pohlen et al. [2018] used demonstrations from a human expert to teach an agent how to finish the first level of Montezuma’s Revenge. Impressively, Aytar et al. [2018] achieved the same feat by merely having the agent watch YouTube videos. Taking the level of engineering to an even higher level, the recent Go-Explore [Ecoffet et al. 2019] managed to reach the...
159th level of *Montezuma’s Revenge*, far surpassing any previously reported results. However, this method remains controversial amongst RL purists since, in addition to relying on human advice, it fundamentally relies on access to an simulator in order to disable environment stochasticity and save/reload states. While humans do essentially exploit “reloading states” in some learning situations (e.g. when practising a golf swing at a driving range), they are clearly not reliant on this method.

- On the scale of domain knowledge exploitation, other approaches fall somewhere between standard DQN [Mnih et al. 2015] and the heavily engineered methods cited above. For example, Choi et al. [2018] make the assumption that the agent is controlling a protagonist within a “contingent region” of the screen. They train their agent to recognise this location, and use the location to derive count-based novelty bonuses. Stanton and Clune’s [2018] *Deep Curiosity Search* makes a similar assumption, but takes the protagonist’s position directly from the emulator’s RAM state rather than learning it. While our approach is motivated by the same underlying intuition as these agents, it is important to note that it does not *rely* on the game being 2D or involving a player-controlled protagonist. Finally, it is worth noting that Stanton and Clune’s [2018] approach involves a very similar bonus scheme to ours, whereby the agent is rewarded for achieving “intra-life novelty”, i.e. reaching locations that have yet to be visited in the current episode. However, our pellet rewards also decay in value after each time they are collected, which means that they encourage both intra- and inter-life novelty.

### 3.5 Chapter Summary

In this paper, we proposed a new approach to the important problem of identifying subgoals autonomously in high-dimensional state spaces. Our experiments in the *Atari* domain showed that our method was capable of identifying meaningful subgoals from raw pixels. We proposed a novel intrinsic reward scheme for exploiting the subgoals and used it to train an agent that was competitive with strong baselines on three sparse reward games. In addition, our approach helped mitigate the “risky exploration” problem, and performed gracefully on two games where a previous intrinsic reward scheme was detrimental.

### 3.6 Algorithm Pseudocode

A sketch of our full approach is provided in Algorithm 3. For comprehensive implementation details and a complete list of parameter settings, please refer to our source code (https://bitbucket.org/mchldann/aaai2019).
Algorithm 3 Q-learning with Pellet Rewards

1: var: current set of rep. states, $\mathcal{R} = \{s_{p1}, s_{p2}, \ldots, s_{pn}\}$
2: var: time elapsed since last partition added, $t_{\text{partition}}$
3: var: time gap between partition additions, $T_{\text{add}}$
4: var: current candidate for the next rep. state, $\hat{s}_{p_{n+1}}$
5: var: max dist. veered since last partition addition, $D_{\text{max}}$
6: var: the set of partitions visited in the episode so far, $v$
7: var: Monte Carlo mixing coefficient for Q-learning, $\eta_Q$
8: var: Monte Carlo mixing coefficient for EE, $\eta_E$

9: procedure MainLoop()
10: \hspace{1em} Reset()
11: \hspace{1em} // Add representative state for first partition
12: \hspace{2em} $s_{p1} \leftarrow s$
13: \hspace{2em} $\mathcal{R} \leftarrow \{s_{p1}\}$
14: \hspace{2em} $t_{\text{partition}} \leftarrow 0$
15: \hspace{1em} for each episode do
16: \hspace{2em} \hspace{1em} while $s$ is not terminal do
17: \hspace{3em} $\pi \leftarrow \epsilon$-greedy policy derived from $Q$
18: \hspace{3em} Select $a \sim \pi(s, \cdot)$
19: \hspace{3em} Calculate aux. reward $\hat{r}^\pi$ as per Eq. 3.3
20: \hspace{3em} Take action $a$, observe $r, s'$
21: \hspace{2em} // Determine the current partition
22: \hspace{3em} $s_{p_c} \leftarrow \arg\min_{s_p \in \mathcal{R}} d(s', s_p)$
23: \hspace{2em} // Update the set of visited partitions
24: \hspace{3em} $v' \leftarrow v \cup s_{p_c}$
25: \hspace{2em} // Update the best candidate according to the distance measure
26: \hspace{2em} // defined by Equation 3.6
27: \hspace{3em} if $d(s', s_{p_c}) > D_{\text{max}}$ then
28: \hspace{4em} $\hat{s}_{p_{n+1}} \leftarrow s'$
29: \hspace{4em} $D_{\text{max}} \leftarrow d(s', s_{p_c})$
30: \hspace{2em} end if
31: \hspace{2em} Store transition info $\{s, v, a, \hat{r}^\pi, r, s', v'\}$ in the replay memory
32: \hspace{2em} // Add a new rep. state every $T_{\text{add}}$ steps
33: \hspace{2em} $t_{\text{partition}} \leftarrow t_{\text{partition}} + 1$
34: \hspace{3em} if $t_{\text{partition}} > T_{\text{add}}$ then
35: \hspace{4em} $\mathcal{R}.add(\hat{s}_{p_{n+1}})$
36: \hspace{4em} $D_{\text{max}} \leftarrow 0$
37: \hspace{3em} $t_{\text{partition}} \leftarrow 0$
38: \hspace{2em} end if

(May 26, 2019)
### QLearn()

```plaintext
QLearn()
EELearn()
s ← s′
v ← v′
end while
Update all partitions’ visit counts based on v
Reset()
end for
end procedure
```

### Reset()

```plaintext
Reset()
Reset the game and set s equal to the initial state
v ← {}
end procedure
```

### QLearn()

```plaintext
QLearn()
Sample random minibatch of transitions \{s, v, a, r, s′, v′\} from replay memory
r+ ← 0
if v ≠ v′ then
    r+ ← pellet reward for visited partition (i.e. the single partition in v′ \ v)
end if
targ_{one-step} ← r + r+ + γ \max_a Q(s′, v′, a)
Calculate extrinsic and intrinsic returns, R and R+, via the remaining history in the replay memory

\[
targ_{MC} ← R + R^+
\]
\[
targ_{mixed} ← (1 - η_Q)targ_{one-step} + η_Qtarg_{MC}
\]
Update Q(s, v, a) towards targ_{mixed}
end procedure
```

### EELearn()

```plaintext
EELearn()
Sample a minibatch of state pairs and interleaving auxiliary rewards
\{s_t, s_{t+k}, \{\hat{r}^π_t, \ldots, \hat{r}^π_{t+k-1}\}\} from the replay memory with k < m

targ_{one-step} ← \hat{r}^π_t + γ E^π_m(s_{t+1}, s_{t+k-1})

targ_{MC} ← \sum_{i=0}^{k-1} γ^i \hat{r}^π_{t+i}

targ_{mixed} ← (1 - η_E)targ_{one-step} + η Etarg_{MC}
Update E^π_m(s_t, s_{t+k-1}) towards targ_{mixed}
end procedure
```
One of the most impressive aspects of Mnih et al.’s [2015] Deep Q-Network (DQN) agent was its generality. The suite of Atari games they considered encompassed a wide range of genres, including sports games, adventure games, platformers and shooters, yet their agent learned to play reasonably well in most situations from only raw visual input and a feed of the game score. As discussed in the previous chapter, a notable exception to this rule was in tasks with sparse rewards, which commonly manifested as navigation problems. Examples include obtaining the first key in Montezuma’s Revenge and reaching the first item in Venture. Navigation problems are themselves an important and much-studied domain, and although the novelty-driven agents of the previous chapter can eventually solve these examples from raw pixel input, it is evident from the amount of training time they require that they tackle such tasks in a different manner to human players.

Of course, a clear advantage possessed by human players is the amount of pre-existing domain knowledge they bring to these games. Upon seeing the first screen in Montezuma’s Revenge, most human players will realise that they are dealing with a navigation task where the aim is to reach the key. This realisation hinges on several facets of human intelligence: object recognition, a semantic understanding of keys and doors, and the ability to intuit that one is “supposed” to acquire the key because Montezuma’s Revenge is an adventure game. Nonetheless, pre-existing knowledge does not entirely account for human players’ edge in sparse reward navigation tasks. While injecting task understanding might increase learning speed (e.g. via reward shaping [Ng et al. 1999]), all learning-based agents fundamentally rely on task repetition. Human players, on the other hand, can often solve previously unseen tasks on their first attempt.

This chapter covers material previously published in [Dann et al. 2017a;b; 2018b].
This difference between learning-based artificial agents and human players is well-illustrated by games that use random level generation, such as *Infinite Mario* (Figure 4.1). *Infinite Mario* is a fan-made game that incorporates elements of the classic *Nintendo* games *Super Mario World* and *Super Mario Bros. 3*. The game’s levels are “flat” in structure, meaning that the player must predominantly travel in one direction to reach the level’s end. For level-completing agents, this means that sparse rewards are not a major problem, as they can learn to travel greedily towards the goal. Nonetheless, maze-like navigation tasks often arise via secondary objectives, such as the collection of powerups. Unlike the navigation tasks faced in *Montezuma’s Revenge*, however, those encountered in *Infinite Mario* are different each episode, owing to the game’s random level generation. On one episode, the player might be faced with the scenario in Figure 4.1a, while on the next they might be faced with Figure 4.1b, then Figure 4.1c, and so on. Since human players with limited prior experience in the game can often solve such tasks at their first attempt, it is clear that they do not rely on task repetition alone. Instead, they appear to leverage predictive models of the environment. For example, players of *Infinite Mario* learn that the left and right buttons cause Mario to run, while pressing the jump button launches Mario into the air. This enables them to visualise solutions to previously unseen navigation tasks, essentially substituting simulated experience for real experience.

Unfortunately, mirroring the approach just described entails multiple challenges from an algorithmic standpoint. First, there is the problem of model acquisition. Exploiting access to the game’s internal model (e.g. via programmatic access to a hardware emulator) is essentially “cheating”, since this access is unavailable to human players. Moreover, in some domains there is simply no way to access the internal model. Therefore, to ensure extensibility and to truly match human players, the agent must be capable of learning an approximate model from experience. A second issue is that videogame agents must respond in real-time, but maze-like navigation tasks such as those in Figure 4.1 may take hundreds of frames to complete. If a solution is beyond the agent’s maximum search depth, and there are no intermediate rewards along the path, the agent can only guess at the

![Figure 4.1](image-url)
best course of action. Level-completing agents can address this problem by using distance travelled towards the goal as a heuristic progress measure. However, minimising the direct distance to the goal on maze-like navigation tasks may cause the agent to become stuck at a dead-end, such as in Figure 4.1c.

In Chapter 2, we noted that a classical way to address time scale granularity in a learning context is to introduce temporally abstract “skills” that span multiple time steps (see Section 2.4). However, in a planning context, typical mechanisms for incorporating temporal abstraction, such as the options framework (Section 2.4.1), further complicate the model acquisition problem. This is because, in general, skill policies are stochastic and have uncertain run times.

In our view, it is unlikely that human players learn a detailed stochastic model that captures all the possible outcomes of temporally abstract actions. Nonetheless, human players do appear to leverage temporal abstraction when tackling maze-like navigation tasks in classical platform games. In this chapter, we propose an approach that attempts to reconcile these observations, inspired by the authors’ own experience playing such games. We hypothesise that the “god’s eye” or synoptic view afforded in platform games is critical to planning, as it enables one to perform a crude lookahead by visualising a series of local translations of the protagonist. Further, we contend that a crude lookahead is usually sufficient for human players, as they can rely on reflexes to handle unforeseen circumstances as they arise during plan execution. Following this intuition, we propose an approach that performs abstract lookaheads in a similar manner, and ranks candidate plans according to the estimated difficulty of their constituent steps. We apply this approach to randomly generated navigation tasks in Infinite Mario, where not only is our agent capable of finding complex plans in real-time, but also significantly outperforms a state-of-the-art agent that exploits an exact low-level model.

4.1 Synoptic Vision Planning (SVP)

In this section, we present a formal approach motivated by the above intuition. To reflect the target domain of the approach – namely, environments where the agent is physically situated and is afforded a synoptic view of its physical surrounds – we title the approach Synoptic Vision Planning (SVP). In Section 4.1.1, we elaborate on the assumptions of the approach. We then present the core method, which proceeds in four steps. The first two of these occur over an offline, pre-training method:

- **Training a Local Movement Skill** (Section 4.1.2). While random level generation makes it difficult to learn extended tasks through repetition, local segments of a previously unseen level may have been experienced many times before, e.g. a pit that is 4 units wide, or a platform that can only be reached by performing a hook jump. These recurring subtasks are suited to learning through repetition. To facilitate
CHAPTER 4: SYNOPTIC VISION PLANNING (SVP)

transfer of local task knowledge between levels, we teach the agent a general policy for performing local translations, which we term the local movement policy.

- Learning the Likelihood of the Local Movement Skill Succeeding (Section 4.1.3). As discussed in the preamble to this chapter, predicting the exact outcome of an extended action is challenging. In light of this, rather than attempting to model all the possible outcomes of a local movement, we focus on the simpler, binary classification problem of predicting whether or not the local movement policy will succeed on a given task.

Following these offline training steps, the agent handles previously unseen, full-screen navigation tasks via an online component, which consists of two further stages:

- Planning via Synoptic Vision (Section 4.1.4). During planning, the agent performs temporally abstract lookaheads by projecting chains of local movements, as illustrated schematically in Figure 4.2. Future states are predicted optimistically by assuming that each step will succeed. The advantage of projecting successful outcomes only is that the position of the protagonist at the termination of each local movement becomes roughly known (as indicated by the transparent Marios in the figure). The plan with the least execution difficulty is calculated by leveraging the success likelihood estimator trained in the previous stage.

- Plan Execution and Replanning (Section 4.1.5). While the planner assumes that each local movement will be executed successfully, this may not occur in practice. Further, even after a successful step, the termination state may deviate meaningfully from the planner’s projection. For example, a previously unsighted enemy may have entered from the edge of the screen. For these reasons, sometimes it will be necessary

Figure 4.2: A schematic example of how SVP forms temporally abstract plans. A global plan for reaching the fire flower is projected as a sequence of local translations.
to replan. However, under SVP, it turns out that continually reassessing the plan can cause behavioural issues, as we illustrate later via a pathological example. We thus propose a more stable approach whereby the current plan is only updated if its suboptimality breaches a fixed threshold.

After presenting the full approach, we return to some subtleties regarding the training of the local movement policy. In Section 4.1.6, we show that the policy need not in fact be capable of solving arbitrary local navigation tasks, due to SVP’s planning component. We elaborate on this point by considering two alternative reward schemes for training the policy: Binary rewards, which encourage success via any means possible, and basis rewards, which discourage the agent from learning excessively complex manoeuvres. The idea behind the latter scheme is to offload as much high-level logic as possible to the agent’s planning component. In Section 4.1.7, we note that there are really two types of task failure: failure whereby the agent exceeds the task’s time limit, and failure whereby the agent dies. We propose a hybrid reward scheme to ensure that the agent will prefer living over dying in scenarios where the probability of success is remote.

4.1.1 Assumptions

As illustrated earlier in Figure 4.2, SVP predicts future states by manipulating the position of the protagonist while leaving the rest of the state representation intact. Implicitly, this means that the method requires access to a factored, object-based state representation. Object recognition is a well-established subfield within computer vision, but it is beyond the scope of this thesis. In our view, it is a separable component of the problem that can be addressed at a later time. Therefore, in addition to assuming a synoptic view, we assume in this chapter that the agent has access to a factored state representation that captures object positions, including that of the protagonist.

A further concern regarding the state representation is that the frame of reference used has ramifications on policy reusability. To illustrate this, compare the following representations of a training task: (A) “Move the protagonist from global co-ordinates (10, 5) to (14, 7)” versus (B) “Translate the protagonist from its current position by the vector (4, 2)”. The former, global co-ordinate representation links the task to a fixed point in the level, whereas the latter, agent-centric representation could equally describe tasks elsewhere in the level. As Konidaris and Barto [2007] show in their work on agent-space options, policies trained from agent-centric representations are inherently more reusable. Since it is important under SVP that both the local movement policy and its success likelihood estimator are reusable, we assume for the remainder of the chapter that they are trained from agent-centric representations.

A final point regarding reusability is that it is important for the agent to be trained from a variety of training scenarios. In games with random level generation, such as
Infinite Mario, this will occur naturally. In other games though, some types of task might arise frequently, while others might be rare. To extend our approach to these games, it could be necessary to balance the training scenarios. However, for simplicity, we assume in this work that the agent naturally encounters a sufficient variety of training scenarios, leaving the challenge of balancing the scenarios for future work.

4.1.2 Training a Local Movement Skill

There are many capabilities in platform videogames that could be regarded as “skills”. Examples include running, jumping, climbing a ladder, killing an enemy, collecting an item, and smashing a brick. However, since we are primarily concerned with navigation tasks in this chapter, we restrict our attention to a particular type of skill; namely, local movement. Local movement can be regarded as an umbrella skill, since it encompasses actions such as running, jumping and climbing ladders. While it does not explicitly encompass non-movement actions, such as killing enemies and smashing bricks, it may implicitly entail these actions when they are necessary to achieve the desired movement, e.g. it may be necessary to kill an enemy in order to run from A to B.²

To formalise the skill of performing a local movement, we begin by defining a tiling over the environment’s spatial dimensions. The tiling corresponds to a state abstraction, \( \psi : S \rightarrow T \) that maps each state \( s \in S \) to a tile \( t \in T \) based on the protagonist’s spatial coordinates in \( s \). Tiling the state space serves a dual purpose: On the one hand, it discretises the search space for the high-level planner, limiting the number of possible plans. On the other, it allows us to define the notion of a local movement more precisely as a translation of the protagonist between nearby tiles. The coarser the tiling, the less trajectories the planner must consider, but the less precise the goal of each local movement becomes. In classical platform games, where the game world conforms to a grid, a natural approach (and the one we take in our experiments) is to match the tile size to the game’s grid size.

To quantify the proximity that is considered “local”, we leverage a distance metric over the set of tiles, \( d : T \times T \rightarrow \mathbb{R}^+ \cup \{0\} \). Note that there is no longer a need to learn a distance function, as there was in the previous chapter, since knowledge of object positions is now assumed. Given a fixed neighbourhood size parameter, \( D \in \mathbb{R} \), we define the set of tiles neighbouring \( t \) as:

\[
N_t^D = \{t' \in T \mid d(t,t') \leq D, t' \neq t\}
\]  

(4.1)

In our experiments, we set \( d \) to a Manhattan metric, which means that \( N_t^D \) corresponds to a square grid of tiles centred on \( t \), excluding \( t \) itself, as illustrated in Figure 4.3. These tiles form the set of allowed targets for the local movement skill when the player is located at

²There are situations in which explicit non-movement skills are arguably needed. We discuss the limitations of considering only local movement and point to possible extensions for handling this in the conclusion to this thesis (Chapter 6).
SECTION 4.1: SYNOPTIC VISION PLANNING (SVP)

Figure 4.3: Under a Manhattan distance metric, $N^D_t$ corresponds to a square grid of tiles surrounding the current tile, $t$. At the start of each training episode, a random goal tile $t_g$ is assigned from within $N^D_t$.

tile $t$. To ensure that the agent’s range of movement is not restricted by this formulation, the neighbourhood size needs to be large enough to capture extreme movements, such as performing a maximum distance jump. On the other hand, increasing the neighbourhood size makes training the local movement policy more difficult and also increases the search width for SVP’s planner, so it should only be set as large as need be.

Rather than training a separate policy for every possible local translation, our approach is to train a parameterised policy that takes the target tile as an input. In other words, the local movement policy is a kind of UVFA (Section 2.4.3). Given an augmented state space $S^* = S \times T$ that captures both the environment state and the location of the target tile, we seek a policy $\pi_L : S^* \rightarrow A \times [0, 1]$ for navigating the agent to the target tile. As described in the pseudocode for our method (see Algorithm 4), we generate training tasks by choosing a random tile from within the current neighbourhood and designating it as the goal tile, $t_g$. The goal tile is reset whenever any of the following occur: (1) an episode commences, (2) the goal tile is reached, or (3) a time-out condition is met. The policy is trained via Q-learning from an auxiliary reward $r^*$ that encourages progress towards the target tile. We discuss the specifics of this reward in Section 4.1.6, as there are some subtleties to consider that are best understood after the entire approach has been presented. In practice, we estimate the Q-function via a neural function approximator, so we also incorporate experience replay and a target network. However, these components are omitted from Algorithm 4 for simplicity.

Given a local movement policy, one can derive an option for navigating from any given tile, $t_a$, to any neighbouring tile, $t_b \in N^D_{t_a}$. Recall from Section 2.4 in the background that an option consists of a tuple $\langle I, \pi, \beta \rangle$ where $I \subseteq S$ is the set of initiation states, $\pi$ is the option policy and $\beta : S \rightarrow [0, 1]$ is the termination condition. Following this formalism, we denote the derived option for navigating from tile $t_a$ to tile $t_b$ as the tuple $o^b_a = (\mathcal{I}_a, \pi^b_L, \beta_b)$ where:
• $I_a = \psi^{-1}(t_a)$, where $\psi : S \rightarrow T$ is the map from states to tiles.

• $\pi^b_L$ is a policy for navigating to $t_b$, derived by setting the local movement policy’s goal tile equal to $t_b$. That is, $\pi^b_L(s, a) = \pi_L(s^*, a)$ where $s^* = (s, t_b)$.

• $\beta_b(s) = \begin{cases} 
0, & \text{if } s \notin \psi^{-1}(t_b) \\
1, & \text{if } s \in \psi^{-1}(t_b)
\end{cases}$

Algorithm 4 Local Movement Training (via Q-Learning)

1: parameter: maximum number of time steps before resetting the training task, $K$

2: procedure MainLoop
3: Initialise $Q^{\pi_L}(s^*, a)$
4: for each episode do
5: Sample $s$ from start distribution
6: $t_s \leftarrow \text{getTile}(s)$
7: $\text{ResetGoal}()$
8: while $s$ is not terminal do
9: $s^* \leftarrow \langle s, t_g \rangle$
10: Choose $a$ from $s^*$ via $\epsilon$-greedy policy derived from $Q^{\pi_L}$
11: Take action $a$, observe $s'$
12: $s^{*'} \leftarrow \langle s', t_g \rangle$
13: Calculate local movement training reward $r^*$ on the basis of $s^*$, $s^{*'}$
14: $Q^{\pi_L}(s^*, a) \leftarrow Q^{\pi_L}(s^*, a) + \alpha_L[r^* + \gamma_L \max_{a'} Q^{\pi_L}(s'^*, a') - Q^{\pi_L}(s^*, a)]$
15: $s \leftarrow s'$
16: $t_s \leftarrow \text{getTile}(s)$
17: $k \leftarrow k + 1$
18: if $t_s = t_g$ then
19: Mark training task as successful
20: $\text{ResetGoal}()$
21: else if $k \geq K$ or $s$ is terminal then
22: Mark training task as failed
23: $\text{ResetGoal}()$
24: end if
25: end while
26: end for
27: end procedure

28: procedure ResetGoal
29: $t_g \leftarrow$ sample random tile from $N^{D}_{t_s}$
30: $k \leftarrow 0$
31: end procedure
### 4.1.3 Learning a High-Level Model

In order to leverage the local movement policy for temporally abstract planning, we need some way of predicting the outcome of a local movement. At the most granular level, the outcome of an option can be modelled via the following distribution:

\[
p(o, s', k|s) \equiv \Pr(o \text{ terminates in } s' \text{ after } k \text{ time steps when initiated from } s) \quad (4.2)
\]

In videogames, however, this function will typically be so granular that it is impractical to learn. Moreover, even if it were practical to learn the function accurately, forcing the planner to consider all possible low-level outcomes of an option goes against the main aim in SVP, which is to simplify the search in order to calculate long-term plans efficiently.

Our approach to this challenge is to adopt a simpler model. Rather than attempting to learn the full, granular distribution, we estimate only the likelihood of a local movement’s success, i.e. how likely it is that the protagonist will reach the target tile within a chosen time limit. Specifically, the agent learns the following function:

\[
\Pr(t_a, t_b, T_{\text{max}}|s) \equiv \sum_{s' \in \psi^{-1}(t_b)} \sum_{k=1}^{T_{\text{max}}} p(o^b_a, s', k|s) \quad (4.3)
\]

Due to the double aggregation over termination states and time, this function is much less granular than the low-level distribution over which it sums.

To learn this function, we take a supervised approach. After the local movement policy has been trained, we continue setting random target tiles within the protagonist’s neighbourhood. We apply the local movement policy to each task and observe whether it succeeds or not (as per lines 19 and 22 of Algorithm 4). For each step in the sub-episode, we log the current state, the current tile, the target tile, the time remaining to reach the target tile (\(T_{\text{max}} = K - k\)), and a label of 1 or 0 depending on whether the movement ultimately succeeded. For stability, we store this information in an experience cache and sample from it in the usual way during training.

Interestingly, Konidaris and Barto [2009] train a similar probability function in their work on *skill chaining*. However, they use it for an entirely different purpose; namely, to determine the set of initiation states for an option. (Their idea is that an option should only be initiated from states where there is a reasonable likelihood of it succeeding.) Further, their approach involves training a separate probability estimator for each option. Our function is trained over a wide variety of conditions such that, ideally, it will learn generalisable high-level knowledge, e.g. the fact that it is impossible to run through walls.

### 4.1.4 Planning via Synoptic Vision

In the preceding sections, we covered the offline training of the local movement policy and its success likelihood estimator. We now turn to the online planning component of the agent, which is tasked with calculating full-screen, abstract plans in real-time.
To formalise our method, we must first define what we mean by an “abstract plan”. Under SVP, plans are formulated as sequences of tiles. Specifically, an abstract plan for reaching goal tile $t_g$ is a sequence of tiles such that the first tile contains the initiation state, the final tile equals $t_g$, and each successive tile belongs to the neighbourhood of its predecessor. Accordingly, the full set of abstract plans for reaching tile $t_g$ from state $s$ (under neighbourhood size parameter $D$) is:

$$\{(t_1, t_2, \ldots, t_n) \mid \psi^{-1}(s) = t_1, t_n = t_g, t_{i+1} \in N_{t_i}^D\}$$  \hspace{1cm} (4.4)

The aim of the planner is to find the sequence within this set with the maximum likelihood of succeeding, assuming each step is to be executed by the local movement policy. Given a limit of $T_{\text{max}}$ time steps for performing each local movement, the probability of all steps in an abstract plan succeeding is:

$$\Pr[\text{all steps in } (t_1, t_2, \ldots, t_n) \text{ succeed}] = \prod_{i=1}^{n-1} \Pr(t_i, t_{i+1}, T_{\text{max}} | s_i)$$  \hspace{1cm} (4.5)

where $s_1$ is the initiation state and the remaining $s_i$ are projected on the assumption that all preceding movements were successful.\footnote{Under SVP the future $s_i$ are predicted deterministically, but under a stochastic model the right-hand sides of Equations 4.5 and 4.6 would need to be written as expectations. We consider extensions of SVP to handle stochasticity in the high-level model in Chapter 6.}

Rather than maximising this product directly, one can equivalently maximise the log-likelihood of success (due to the monotonicity of logarithms). This is a more convenient formulation, as the log product rule lets us rewrite the term to be maximised as a sum:

$$\log \Pr[\text{all steps in } (t_1, t_2, \ldots, t_n) \text{ succeed}] = \sum_{i=1}^{n-1} \log \Pr(t_i, t_{i+1}, T_{\text{max}} | s_i)$$  \hspace{1cm} (4.6)

Lastly, rather than maximising this sum, we can instead minimise the sum of negative log-likelihoods. This last formulation is equivalent to solving a classical shortest path problem. The equivalence arises by constructing a graph where the nodes correspond to the tiles and each edge weight equals the negative log-likelihood of the corresponding local movement succeeding. The abstract plan with the greatest likelihood of success can thus be found by applying Dijkstra’s algorithm [Dijkstra 1959].

Before we can proceed with this approach, however, we must address the fact that most of the local movements’ initiation states (the $s_i$ in Equation 4.6) are unknown. The initiation state for the plan is presumably equal to the current state, but all subsequent initiation states must be predicted somehow. Referring back to Figure 4.2, the problem is essentially that of predicting subfigures 2 – 6.

It is at this point where our assumption of a synoptic view is important, as it allows one to predict static aspects of future states (such as the level structure in Infinite Mario) with certainty. Further, since it is assumed that the chain of local movements preceding...
each initiation state was successful, one also knows that the protagonist must be located within the target tile of the previous movement. Accordingly, one can construct a loose estimate of an initiation state by simply teleporting the protagonist to the centre of the previous movement’s target tile while leaving the rest of the game world untouched. Of course, this is a very crude prediction method; it rounds the protagonist’s arrival location to the centre of the tile, ignores the protagonist’s velocity (we assume it is zero in all projected states), and ignores the intervening movements of enemies. However, for the purpose of calculating future movements’ success likelihoods, we posit that this method will typically yield reasonable estimates. After all, it is generally the static level structure that places hard limits on which movements are possible, while non-static game elements such as enemies and moving platforms are usually negotiable through reflexes. To reflect this, we refer to projections constructed in this manner, such as those in subfigures 2 – 6 of Figure 4.2, as archetypal states, meaning that they capture the fundamental essence of the local tasks that are expected to arise.

Of course, this naïve method will be ill-suited to games where moving objects besides the protagonist are critical to planning, e.g. in Ms. Pacman, where it is impossible to weave around enemies as it is in Infinite Mario. If there are multiple high-level scenarios that may arise, e.g. depending on the different choices a ghost might make at an intersection in Ms. Pacman, it may be better to take a probabilistic approach involving multiple archetypal states. However, since our experiments in this chapter focus on testing our core assertion (that it is okay to ignore details that can be handled via reflexive play), we only consider the case where enemies are passable through skill, and construct archetypal states in the exact manner described above. However, more sophisticated approaches are certainly possible, and we intend to explore these in future work.

4.1.5 Plan Execution and Replanning

The final stage in SVP is to execute the sequence of movements identified by the planner by passing them to the local movement policy. However, at this point the agent will almost certainly encounter difficulties that it did not foresee in its projections. These could be relatively minor, such as a new enemy emerging from the side of the screen, or major, such as the agent failing to execute a jump properly in Infinite Mario. There is, after all, no guarantee that the local movement policy will successfully execute each step. Accordingly, sometimes it will be necessary for the agent to replan.

One possible approach to replanning is to recalculate the entire plan at every time step. This is equivalent in essence to the classical idea of interrupting an option if its action-value drops below that of some other action [Sutton et al. 1999, Section 4]. Unfortunately though, there is a major flaw with this approach under SVP: Unless all the estimated success likelihoods are logically consistent (as explained shortly), the agent may become caught in a behavioural loop. This is exemplified by the pathological scenario shown
in Figure 4.4. At the beginning of the scenario, the planner assigns tile \( t_w \) as the next waypoint. However, during its attempt to reach \( t_w \), the agent encounters tile \( t_b \), from where it estimates a greater difficulty of moving to \( t_w \) directly than it did from \( t_a \). (This is what we mean by a logical inconsistency. It cannot be that \( t_b \rightarrow t_w \) is more difficult than \( t_a \rightarrow t_w \), since the easiest path from \( t_a \) to \( t_w \) passes through \( t_b \).) Under the approach of continual replanning, the agent will now redirect to \( t_a \), because the likelihood of the plan \( t_b \rightarrow t_a \rightarrow t_w \) succeeding is judged as greater than that of the direct plan \( t_b \rightarrow t_w \). Once the agent has retreated to \( t_a \) the cycle will begin again, with the agent forever stuck running between \( t_a \) and \( t_b \).

Of course, the above scenario begs the question of how such logical inconsistencies could arise in the first place. There are two points to note here: The first is that the function trained to estimate success likelihoods may simply be inaccurate. The second is that the likelihood estimator may not recognise the underlying logic of the problem. For example, in scenarios where there is a wall just to the right of the protagonist, the game rules dictate that the protagonist can reach tiles to the left of the wall, but not to the right. However, a function approximator may not learn this exact relationship; instead, it may simply learn that it is “bad” when the target tile is located to the right and there is also a wall there. This type of phenomenon did seem to occur in our experiments, such that it was not uncommon for scenarios like that in Figure 4.4 to arise.

Our approach to this issue is to replan only if the current step is deemed failed. This occurs if and only if:

\[
t_w \notin N^D_{t_b} \quad \text{or} \quad \min \left[ \frac{\Pr(t_b, t_w, T_{\text{max}}|s_b)}{\Pr(t_a, t_w, T_{\text{max}}|s_a)}, 1 \right] \leq k \tag{4.7}
\]

where \( t_w \) is the current waypoint, \( s_b \) and \( t_b \) are the current state and tile, \( s_a \) and \( t_a \) are the state and tile from which the step was initiated, and \( k \geq 0 \) is the replanning threshold.

The first condition ensures that the step is failed if the current waypoint is no longer in range of the local movement policy. The logic behind the second condition is as follows:

![Figure 4.4: An example from Infinite Mario where the agent will become stuck trying to reach the next waypoint, \( t_w \).](image-url)
The current likelihood of reaching \( t_w \) is compared to the original estimate. If the current estimate is significantly lower, it stands to reason that something has gone wrong during execution. A relative rather than an absolute threshold is used because the current step may have had a low chance of succeeding to begin with. (It may just be an intrinsically difficult step). Setting \( k = 1 \) is equivalent to continual replanning, while \( k = 0 \) means that the plan will only be recalculated if the current waypoint falls out of range. In principle, the parameter should be set low enough that cycles are avoided, but high enough that the plan will be reset if the agent makes a clear error. For the example in Figure 4.4, the cycle will be avoided so long as \( k < \frac{94}{97} \).

In addition to the above replanning logic, the agent progresses to a later tile in the plan whenever the current waypoint is reached or a later tile is judged easier to reach directly than the current waypoint. The latter condition ensures that the agent will not backtrack unnecessarily if it overshoots the current waypoint in a beneficial way.

### 4.1.6 Reward Schemes for Training Local Movement

In Section 4.1.2, where we described the training of the local movement policy, we deferred a discussion of the reward scheme used, as there are some subtleties to consider that are best understood in light of the full SVP approach. Now that all components have been presented, we return to this topic and consider two alternative schemes: binary and basis rewards. Binary rewards encourage the agent to take the fastest path to the target tile, regardless of that path’s complexity. Basis rewards penalise non-greedy behaviour, thus discouraging the agent from learning complex manoeuvres. While the former scheme can be expected to yield a more powerful policy, we explain in the following discussion why the remaining components of SVP may benefit from training a simpler one.

**Binary Rewards**

To train a policy for performing local movements, a natural scheme is to provide a reward of 1 for reaching the target tile and 0 on all other transitions. Under this scheme, which we dub *binary rewards*, the optimal behaviour is to reach the target tile as fast as possible (assuming a discount of \( \gamma < 1 \)). To improve the scheme’s efficiency, we can leverage knowledge of the protagonist’s distance from the target tile via *potential-based reward shaping*. As discussed in the background chapter (Section 2.3.3), potential-based shaping preserves optimal policies, providing an augmented reward of the form:

\[
\hat{r}_t = r_t + \gamma \Phi(s_t) - \Phi(s_{t-1})
\]

where \( r_t \) is the native reward (in this case, a plain binary reward) and \( \Phi \) is the *shaping potential*. Generally, an appropriate choice of shaping potential is an estimate of the task’s
optimal value function. Therefore, an appropriate potential for local movement tasks is:

$$\Phi(s) = \begin{cases} 
\gamma^{\text{steps}(s, t_g)}, & \text{if } s \text{ is non-terminal} \\
0, & \text{if } s \text{ is terminal} 
\end{cases}$$  \hspace{1cm} (4.9)$$

where \(\text{steps}(s, t_g)\) is an estimate of how many time steps it would take to reach \(t_g\) from \(s\) under an optimal policy. In our later experiments with Infinite Mario, we estimate this from knowledge of Mario’s maximum movement speed.

**Basis Rewards**

If our only objective was to maximise the agent’s success rate on local tasks, the scheme described above would be a natural choice. However, given that SVP entails a planning component, it is not actually necessary for the local movement policy to be capable of performing arbitrary local movement. As long as the policy is capable of performing direct movement along unobstructed lines, there is no need for it to be capable of “two stage” movements, as in Figure 4.5a, because the planner can decompose these into direct movements, as in Figure 4.5b. Furthermore, in the context of our full approach, there may be some benefit to training direct movement only. Under a policy capable of arbitrary movement, the complexity demanded of both the policy and its likelihood of success estimator may be substantial. However, a policy that is only capable of direct movement demands less functional complexity, and its success likelihood estimator must only be able to judge whether a direct path to the target tile is possible. Moreover, this should force the planner into placing waypoints more discriminantly, as plans with indirect segments will be assigned smaller likelihoods of success.

By analogy with vector terminology, we henceforth refer to the direct type of movement as *basis* movement. To train basis movement, we define a reward scheme that encourages greedy movement towards the target tile:

$$r_t = \Phi(s_t) - \Phi(s_{t-1})$$  \hspace{1cm} (4.10)$$

![Figure 4.5: The “two stage” movement in subfigure (a) can be decomposed into the two basis movements shown in subfigure (b).](image)
where $\Phi$ is defined identically to Equation 4.9.

This reward is identical to the shaping component of Equation 4.8, except that the leading factor of $\gamma$ has been removed. The removal of this factor is crucial, as otherwise all policies would be optimal (as explained in [Ng et al. 1999, Remark 2]). The effect of this modification is that a movement away from the target tile is no longer fully offset by an equal, subsequent movement back towards the target tile. Instead, backwards-forwards sequences like this will yield a net negative return. The extent of the penalty is determined by the discount factor used.$^4$ Under a discount of 0, the optimal behaviour is always to travel greedily in the direction of the target tile, regardless of whether the path is blocked. Under a discount of 1, all policies that eventually reach the target tile are optimal. Under intermediate values, in some scenarios it may be optimal to backtrack initially, provided that this ultimately allows the agent to reach closer to the target tile. In our experiments, we chose an intermediate value that allowed the agent to learn slightly curved movements, as otherwise the agent could not discover certain types of jump.

4.1.7 Encouraging Survival

In games containing hazards that can kill the player, such as enemies and bottomless pits, the reward schemes described above provide a natural incentive for the agent to remain alive, as dying prevents the future accumulation of positive rewards. However, in certain “hopeless” situations, basis rewards actually encourage the agent to die. For example, if the target tile is located on the other side of a pit that is impossible to jump over, jumping halfway across the pit yields a greater return than remaining still or backing away from the pit. Binary rewards never actually incentivise episode termination, but neither do they incentivise survival in such situations. To address this problem, instead of using plain basis or binary rewards in our experiments, we use a hybrid scheme that explicitly rewards survival:

$$r_{\text{full}} = (1 - L)r_{\text{orig}} + L(1 - \gamma)$$  \hspace{1cm} (4.11)

where $r_{\text{orig}}$ is the underlying binary or basis reward and $0 \leq L \leq 1$ is the survival incentive coefficient.

In the infinite horizon, the $L(1 - \gamma)$ term yields a total discounted return of $L$. On its own, this disincentivises the agent from terminating the episode via any means, including by reaching the goal. Since we do not wish to penalise success, we compensate the agent for the loss of future survival returns upon reaching the goal as follows: First, we append an additional output to the Q-network for estimating $V_{\text{surv}}^\pi(s)$, the expected return under $\pi$ based solely on the survival returns and ignoring terminations due to episode success.

$^4$Though there is no factor of $\gamma$ in Equation 4.10, the Q-learning algorithm used to train the local movement policy still involves a discount factor. It is this discount to which we refer.
Next, upon reaching the goal, we pay the agent a compensatory reward equal to value of this function in the termination state.

4.2 Experimental Configuration

To test SVP’s ability to formulate and execute medium- to long-term plans, we applied it to randomly generated navigation tasks in *Infinite Mario*. In this section, we provide further details regarding the experimental domain, the configuration of SVP’s various components and the agents that we benchmarked against.

4.2.1 Generating Maze-Like Navigation Tasks

The default level generator in *Infinite Mario* tends to create “flat” levels that are not especially challenging from a planning perspective. The simulation-based agents of [Togelius et al. 2010] and [Jacobsen et al. 2014] were able to storm through these levels by running greedily to the right of screen while jumping to avoid short-term threats, such as enemies and pits. Of course, a weakness of this strategy is that it is ill-suited to levels containing many dead-ends. To generate tasks that demand a longer planning horizon, we modified the game’s level generator to create maze-like structures such as that shown in Figure 4.6.

Rather than evaluating agents on their ability to complete full levels, we tasked them with navigating to a randomly assigned goal tile on the current screen. (Note, however, that an agent capable of arbitrary on-screen navigation may be configured to finish levels

Figure 4.6: An illustration of the goal placement zones used in our experiments. Note that the level structure shown here is just one sample possibility.
by fixing its goal to the right of screen.) For ease of analysis, we defined three goal placement zones. *Zone 1* reached up to 5 squares away from Mario, *Zone 2* was defined as a ring at a distance of 6–8 squares, and *Zone 3* was a ring at a distance of 9–11 squares. Agents were allowed a maximum of 15 seconds to reach the goal tile, and the level structure was regenerated after each episode. All success/time-out/death rates reported in our results were averaged over 5,000 episodes per agent.

### 4.2.2 Agent Configuration

In the following subsections, we describe how SVP’s various components were configured for *Infinite Mario*.

#### Local Movement Policy

To segment the game world of *Infinite Mario* into tiles, we followed the game’s natural grid layout by setting the tile size equal to that of a brick. The neighbourhood size parameter, $D$, which controls the reach of the local movement skill, was set equal to Mario’s maximum jump height of 5 bricks. Since mid-air and mid-wall locations are not natural waypoints in most platform games, we removed such tiles from all neighbourhoods.\(^5\) (This is equivalent to overriding the distance metric such that $d(t_1,t_2) = \infty$ if $t_2$ is in mid-air or part of a wall.) Ignoring these tiles helped to prevent impossible training tasks from being generated and also reduced the search width for the agent’s planning component.

We used a state representation very similar to that of Togelius et al. [2009]. It contained Mario’s velocity, the position of the training goal, and a binary encoding of the impassable tiles within Mario’s neighbourhood. Since this representation quantises the positions of impassable tiles to whole-brick distances, we also included Mario’s fractional offset from the grid centre. For experiments involving enemies, we also included the positions and velocities of the two enemies nearest Mario.

All local movement policies used in the course of our experiments were trained via *n-step Q-Learning* (Section 2.3.1) using the *Double DQN* modification (Section 2.2.2). Action-values were approximated via fully connected, feed forward neural networks with 2 hidden layers of equal in size to the input. To improve training stability, we employed experience replay and maintained separate training and target networks, updating the target network periodically. The remaining configuration settings are listed in Table 4.1. Note that the two alternative reward schemes demanded considerably different discount factors. Under basis rewards, a sharp discount was required to disincentivise complex

\(^5\)Another way we could have addressed unnatural waypoints, such as mid-air tiles, is by adding some sort of stability constraint to the criteria for a local movement’s success. For example, we could have required the agent to not only to reach the target tile, but to remain there for a few seconds. In *Infinite Mario*, this would effectively remove mid-air tiles from the agent’s planning considerations, but at the cost of increased training time and slightly slower plan calculation.
Learning rate \(2.5 \times 10^{-4}\)
Momentum 0.95
Discount factor, \(\gamma\) 0.98 for binary rewards, 0.7 for basis rewards
Survival incentive coefficient, \(L\) 0.2
Maximum episode length 15 seconds
Experience cache size \(1.5 \times 10^6\) frames
Cache pop. when learning commenced 50,000 frames
Target network refresh rate 10,000 frames
Action length 4 frames
Multi-step returns \(n\) 3
Exploration policy \(\epsilon\)-greedy
\(\epsilon\) decay schedule \(\epsilon = 1/(1 + kn), k = 0.4 \times 10^{-6}, n = \) frames elapsed.
Training time \(1 \times 10^8\) frames

Table 4.1: The training parameters used for Infinite Mario.

manoeuvres, but under binary rewards a much milder discount was required to ensure that the reward for reaching the target tile was not effectively hidden.

**Success Likelihood Estimator**

To train an estimator of the success likelihood function (Equation 4.3), we used the same architecture as described above, but appended an extra input to capture \(T_{\text{max}}\), the time remaining to complete the local movement. The maximum episode length was set to 15 seconds, so \(T_{\text{max}}\) varied from 0 to 15 seconds across training samples. Since the weight gradients in the neural network were generally larger than those encountered during policy training (due to the less predictable nature of the binary training labels), we reduced the learning rate to \(2.5 \times 10^{-5}\). The function was trained for \(10^8\) frames, yielding a total offline training time of \(2 \times 10^8\) frames with policy training taken into account. We saved weights intermittently and kept the network with the smallest mean squared error.

**Planner**

The search space for the planner was configured to be all non-mid-air, non-wall tiles on the current screen. The screen was buffered to extend at least 5 tiles to either side of the target tile and Mario’s current position. For calculating edge weights between tiles, we fixed the \(T_{\text{max}}\) input to the success likelihood estimator at 5 seconds. However, we found that varying this parameter had little effect; smaller/larger values resulted in smaller/larger edge weights, but the planner identified similar routes regardless.

For each local movement policy, we identified a near-optimal replanning threshold by trialling values from \(k \in \{0, 0.1, \ldots, 1\}\). For basis policies, SVP performed best with a rel-
The relatively small replanning threshold of $k = 0.2$, meaning that the agent only replanned when its likelihood of completing the current step dropped below 20% of its original estimate. For binary policies, the optimal setting was slightly higher at $k = 0.4$. The discrepancy here was probably due to the policies’ differing success rates during training. As explained in Section 4.1.6, basis rewards yield less powerful policies, so success likelihood estimators trained from basis policies may become more attuned to recognising failure and hence drop more readily. The surprising smallness of both $k$-values is discussed further in our experimental results.

### 4.2.3 Benchmark Agents

In the experiments that follow, we benchmark against two agents: A simulation-based agent that exploits exact knowledge of the game’s physics to perform a frame-level search, and a modified SVP agent that is configured to be blind to enemies.

For the simulation-based benchmark, we used a slightly modified version of Robin Baumgarten’s A* agent, which won the 2009 Mario AI Competition [Togelius et al. 2010]. In *Infinite Mario*, agents are only allowed 40 milliseconds to calculate each action, so it is computationally infeasible to expand an A* search all the way to a distant goal. Therefore, one must use a heuristic to select the most promising incomplete path when the time budget expires. Baumgarten’s original agent was designed to finish levels, so it was configured to select the trajectory reaching farthest to the right of screen. For point-to-point navigation, we modified this rule to select the trajectory reaching closest to the goal tile. Readers familiar with the spectacular performance of Baumgarten’s agent on flat-structured levels may be surprised by the modified agent’s poor performance on our tasks (as presented in the next section). However, to put the agent’s struggles into perspective, it should be noted that greedy heuristics are far better suited to flat levels than maze-like levels. We did in fact validate the modified agent on flat levels, and its point-to-point navigation was virtually flawless.

Another simulation-based method that we could have tried is Monte Carlo Tree Search (MCTS) [Coulom 2006], which evaluates search nodes by averaging the results of stochastic rollouts. This method has proven remarkably successful in some games where it is difficult to craft a strong heuristic, most famously in Go [Silver et al. 2016]. However, the success of MCTS in Go hinges on the ability to conduct rollouts all the way to episode termination. As we have already noted, it is impractical to do this in *Infinite Mario*. Accordingly, the rollouts must be evaluated at an incomplete point, which necessitates the use of some heuristic. This leads us back to the problem described above; if the rollouts are evaluated based on the remaining straight-line distance to goal, the search will favour the greedy direction and frequently lead to dead-ends. In addition, *Infinite Mario* permits very little simulation time per real action, so nodes must be evaluated based on few rollouts. This

---

*A video of Baumgarten’s agent may be seen at https://www.youtube.com/watch?v=DIkMs4ZHHr8*
yields high variance in the node evaluations, a disadvantage that A* does not have. Finally, we note that even the heavily streamlined MCTS agent of Jacobsen et al. [2014] performed marginally worse than Baumgarten’s A* agent on standard levels. In light of all these reasons, we selected A* for the simulation-based benchmark over MCTS.

The purpose of benchmarking against an enemy-blind version of SVP was to quantify the extent to which the default version of SVP is enemy-aware. To recap, despite the fact that SVP’s high-level planner ignores the movement of enemies, our claim is that the local movement policy, which is reactive to enemies, should be able to account for their true positions as the plan unfolds. However, since all agents will avoid some enemies through sheer luck, it was necessary to compare against an enemy-blind baseline to test the accuracy of this claim.

4.3 Results and Discussion

We present our results over two sections. Section 4.3.1 covers experiments from [Dann et al. 2017a] where enemies were disabled and the goal was placed in Zone 3 only (referring back to Figure 4.6). The aim in these experiments was twofold: To establish a proof-of-concept, and to investigate the basic properties of SVP in a simplified domain that affords ease of analysis. Section 4.3.2 covers experiments from [Dann et al. 2018b], where enemies were enabled and goals were placed across all zones. We investigate whether our agent is truly enemy averse (which is not a given, due to the fact that the high-level planner ignores enemy movements), and consider a hybrid between SVP and A* that seeks to retain the strengths of both approaches while mitigating their weaknesses.

4.3.1 Maze-Like Navigation Without Enemies

The enemy-free game is a near best case for SVP, because it means that the states that arise after local movements succeed will almost exactly match the archetypal states that were projected during planning. Still, there remain two key aspects of the approach that ought to be tested: Firstly, the success likelihood estimator might not train well enough for the agent to derive sensible high-level plans. Secondly, the local movement policy might not train well enough to execute the high-level plans reliably.

In this simplified setting, we trained two local movement policies: one via binary rewards, and another via basis rewards. Henceforth, we refer to the corresponding agents as the binary and basis agents. As Figure 4.7 shows, both variants of SVP were broadly successful. When allowed a full 15 seconds to reach the goal, they achieved far higher success rates than the A* benchmark. The A* agent reached more goals over the first few seconds because, for the problems it could solve, it calculated near optimal paths by exploiting access to the game’s internal model. However, on tasks where its greedy

\[^7\text{The only details that might differ are Mario’s velocity and his exact position within the tile.}\]
Figure 4.7: Success rate versus time taken for the \textit{basis} and \textit{binary} versions of SVP and the A* benchmark.

Figure 4.8: Alternative plans for the same task found by (a) the \textit{basis} agent and (b) the \textit{binary} agent. The goal is highlighted green, waypoints are highlighted yellow.

heuristic led it to dead-ends, it became permanently stuck. Conversely, the low-level behaviour of the SVP agents was not as precise (e.g. on occasion they had to repeat tricky jumps multiple times), but they generally found viable high-level plans.

Overall, the \textit{basis} agent outperformed the \textit{binary} agent. The \textit{basis} agent reached 97.1\% ($\pm$0.2\%) of goals within the full time limit, while the binary agent reached only 90.0\% ($\pm$0.3\%). Recall from Section 4.1.6 that the binary reward scheme can be expected to yield a more flexible local movement policy. This was in fact borne out in our experiments: By the end of training, the \textit{binary} policy solved around 95\% of local training tasks, while the \textit{basis} policy solved around 77\%. However, as anticipated, the inflexibility of the \textit{basis} policy was actually a strength during planning, as the agent was forced to segment its plans into simple, direct steps. On the other hand, the \textit{binary} agent was less precise

\footnote{While it appears in Figure 4.7 that the \textit{binary} agent may eventually catch the \textit{basis} agent, completion times in excess of 15 seconds are arguably unreasonable for the tasks generated. We did in fact try extending the limit to 30 seconds, but the \textit{basis} agent still led, 98.6\% ($\pm$0.1\%) to 95.1\% ($\pm$0.2\%).}
about its waypoint placement, because its success likelihood estimator was less sensitive to obstacles. This greater tolerance meant that the *binary* agent tended to generate less direct plans, as illustrated clearly by the example in Figure 4.8.

Having established that the *basis* agent was stronger, we sought to investigate the effect of varying its replanning threshold, $k$. We compared the tuned configuration ($k = 0.2$) against both continual replanning ($k = 1.0$) and replanning only when the next waypoint became out of range ($k = 0.0$). The results of this experiment are shown in Figure 4.9. The difference between the $k = 0.2$ and $k = 0.0$ agents can be easily interpreted: Both agents performed well on simple tasks, as step failure was not an issue, but the $k = 0.0$ agent struggled to recover from step failure on harder tasks. The $k = 0.2$ agent could usually recover from failure, albeit completing such episodes slowly. This is why the superiority of the $k = 0.2$ agent only becomes pronounced towards the right of the graph. Continual replanning was severely detrimental, with the $k = 1.0$ agent only reaching 57.2% (±0.5%) of goals within 15 seconds. Besides the issue of “behavioural loops”, as explained in Section 4.1.5, another reason why this agent performed poorly was that there often seemed to be many viable plans with similar success likelihoods. Small changes to the current state were often enough to alter the plan rankings, causing the agent to become excessively non-committal, i.e. to switch plans continually without making any progress. The fact that the optimal replanning threshold was so low ($k = 0.2$) suggests that frequent plan switching was the more significant problem, as the “behavioural loop” issue would not have demanded such a low $k$-value unless the edge weights were extremely inconsistent, which they were not.
4.3.2 Maze-Like Navigation With Enemies

In our second set of experiments, we enabled enemies and placed goals across all three zones. Following the findings of the previous section, we configured SVP to use basis rewards. To validate our claim that the overall approach should be reactive to enemies despite the fact that the high-level planner ignores enemy movements, we benchmarked against an enemy-blind configuration of SVP in addition to A*.

The results of these experiments are summarised in Table 4.2, with episodes broken down by termination cause (success, death or time-out after 15 seconds) such that all rows sum to 100%.

As in the enemy-free experiments, the A* agent performed well when the direct line to goal was unobstructed. However, its inability to find long paths around walls meant that its success rates were surpassed by even the enemy-blind configuration of SVP, despite the latter agent’s far higher death rates. Interestingly, the A* agent was slightly more successful with enemies toggled on, since its possession of an exact model meant that it very rarely died\(^9\), and its efforts to avoid harm occasionally caused it to escape dead-ends.

Despite experiencing more deaths than A*, the default SVP configuration was clearly enemy averse. It died much less often than the enemy-blind configuration overall, and its death rate scaled better with the task length. From Zones 1 – 3, its death rate progressed as 6.0 → 12.7 → 15.4% versus 10.8 → 21.9 → 27.9% for the enemy-blind agent. The enemy-aware agent timed out slightly more often than the enemy-blind version because it acted more “cautiously” and tended to wait for awkwardly placed enemies to move.

\(^9\)The non-zero death rate of the A* agent was caused by the level generator occasionally spawning Mario in positions from which it was impossible to escape.

<table>
<thead>
<tr>
<th>Zone</th>
<th>Agent Type</th>
<th>Success %</th>
<th>Death %</th>
<th>Time-out %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SVP</td>
<td>93.0 ±0.4</td>
<td>6.0 ±0.3</td>
<td>1.0 ±0.1</td>
</tr>
<tr>
<td>1</td>
<td>SVP (enemy-blind)</td>
<td>88.8 ±0.4</td>
<td>10.8 ±0.4</td>
<td>0.4 ±0.1</td>
</tr>
<tr>
<td>1</td>
<td>A*</td>
<td>86.1 ±0.5</td>
<td>0.2 ±0.1</td>
<td>13.7 ±0.5</td>
</tr>
<tr>
<td>2</td>
<td>SVP</td>
<td>84.9 ±0.5</td>
<td>12.7 ±0.5</td>
<td>2.4 ±0.2</td>
</tr>
<tr>
<td>2</td>
<td>SVP (enemy-blind)</td>
<td>77.2 ±0.6</td>
<td>21.9 ±0.6</td>
<td>0.9 ±0.4</td>
</tr>
<tr>
<td>2</td>
<td>A*</td>
<td>70.0 ±0.6</td>
<td>0.4 ±0.1</td>
<td>29.6 ±0.6</td>
</tr>
<tr>
<td>3</td>
<td>SVP</td>
<td>81.4 ±0.6</td>
<td>15.4 ±0.5</td>
<td>3.3 ±0.3</td>
</tr>
<tr>
<td>3</td>
<td>SVP (enemy-blind)</td>
<td>70.6 ±0.6</td>
<td>27.9 ±0.6</td>
<td>1.5 ±0.2</td>
</tr>
<tr>
<td>3</td>
<td>A*</td>
<td>59.0 ±0.7</td>
<td>0.3 ±0.1</td>
<td>40.7 ±0.7</td>
</tr>
</tbody>
</table>

Table 4.2: Results compared across the three goal placement zones. Error bounds represent one standard deviation.
Hybridising SVP with A*

In the experiments presented so far, the A* agent excelled on short and/or direct navigation tasks, but struggled on long, maze-like tasks. Conversely, SVP was able to find viable long-term plans around obstacles, but died more often to enemies and was slower to complete simple tasks. Based on the complementary nature of the agents’ strengths, in this section we explore a hybrid approach, whereby SVP is used to derive the high-level plan, but the execution of the local movements is handled by A*.

Under this approach, the success likelihood estimator should logically be trained from scratch, using A* as the underlying policy. However, because our A* implementation consumes the full 40ms time budget allocated per action in Infinite Mario, it prevents one from running the game at an accelerated frame rate. Because of this, training a fresh success likelihood estimator was impractically slow. (For reference, in our earlier experiments we increased the frame rate by roughly an order of magnitude.) Accordingly, for the hybrid agent, we opted to reuse one of the previously trained estimators. Given that A* is best at performing basis-style movements and its success rate is barely affected by the presence of enemies, we selected the basis version of the success likelihood estimator from the enemy-free experiments in Section 4.3.1.

As the results in Table 4.3 indicate, the hybridisation was successful, with the agent maintaining the low death rate of A* while surpassing the success rate of SVP. Its rare time-outs may have been partially attributable to the mismatch between the basis policy used to train the success likelihood estimator and the A* policy that actually executed the plans. Even though using a basis success likelihood estimator should have seen the planner return roughly linear segments, it occasionally returned curved segments that were too indirect for A* to execute. Another likely factor was the inherent inaccuracy in using a trained function approximator to estimate success likelihoods.

<table>
<thead>
<tr>
<th>Zone</th>
<th>Agent Type</th>
<th>Success %</th>
<th>Death %</th>
<th>Time-out %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SVP</td>
<td>93.0 ±0.4</td>
<td>6.0 ±0.3</td>
<td>1.0 ±0.1</td>
</tr>
<tr>
<td>1</td>
<td>A*</td>
<td>86.1 ±0.5</td>
<td>0.2 ±0.1</td>
<td>13.7 ±0.5</td>
</tr>
<tr>
<td>1</td>
<td>Hybrid</td>
<td>99.1 ±0.1</td>
<td>0.4 ±0.1</td>
<td>0.5 ±0.1</td>
</tr>
<tr>
<td>2</td>
<td>SVP</td>
<td>84.9 ±0.5</td>
<td>12.7 ±0.5</td>
<td>2.4 ±0.2</td>
</tr>
<tr>
<td>2</td>
<td>A*</td>
<td>70.0 ±0.6</td>
<td>0.4 ±0.1</td>
<td>29.6 ±0.6</td>
</tr>
<tr>
<td>2</td>
<td>Hybrid</td>
<td>98.2 ±0.2</td>
<td>0.3 ±0.1</td>
<td>1.5 ±0.2</td>
</tr>
<tr>
<td>3</td>
<td>SVP</td>
<td>81.4 ±0.6</td>
<td>15.4 ±0.5</td>
<td>3.3 ±0.3</td>
</tr>
<tr>
<td>3</td>
<td>A*</td>
<td>59.0 ±0.7</td>
<td>0.3 ±0.1</td>
<td>40.7 ±0.7</td>
</tr>
<tr>
<td>3</td>
<td>Hybrid</td>
<td>96.2 ±0.3</td>
<td>0.3 ±0.1</td>
<td>3.5 ±0.3</td>
</tr>
</tbody>
</table>

Table 4.3: The performance of the hybrid agent versus the parent methods from which it was derived.
As a final remark, we note that the success of the hybrid agent hinges on exploiting an exact model during plan execution. This limits the applicability of the approach, and earlier in this chapter we went so far as to argue that this is “cheating”. However, one domain where the hybrid approach could be applicable is automated game testing, since it is fair to assume that game developers will have access to the game’s precise model.

4.4 Related Work

We are not aware of any previous approaches that are organised in the same way as SVP’s four components, nor are we aware of any prior methods that leverage synoptic vision to project archetypal states. However, there has been work conducted in similar experimental domains, and there are previous methods that resemble one or more subcomponents of SVP. The purpose of this section is not to provide an in-depth coverage of this work, as much of the detail would deviate from the topic of this thesis, which is primarily on abstract methods. Instead, the aim is to highlight the main differences between our work and previous methods, so as to clarify which aspects of SVP are novel.

Previous Agents for Infinite Mario

The existing Infinite Mario agents that have already been discussed in this chapter — Baumgarten’s A* agent [Togelius et al. 2010] and the MCTS agent of Jacobsen et al. [2014] — both exploit access to the game’s internal model to perform a low-level search. However, several non-search-based agents have also been applied to this game. For example, Togelius et al. [2009] deployed an agent based on neuroevolution/genetic programming. Harutyunyan et al. [2015] trained a reinforcement learning agent via potential-based reward shaping, enlisting human participants to provide the shaping signal. Similarly to us, Mohan and Laird [2011] leverage hierarchy by training policies to solve local tasks, but rather than training a generic movement skill they train object-oriented policies that centre around a game entity, e.g. “get coin”, “kill enemy”, etc. Nonetheless, a crucial difference between all these non-search-based agents and SVP is that the former were designed for maximising the agent’s score on “flat” levels. Since they do not perform any kind of search, they are ill-suited to maze-like tasks.

The REALM v2 agent of Bojarski and Congdon [2010] also uses hierarchy and bears some strong similarities to our approach. At the top level, it uses an evolved rule set to decide between four “plans” (progress, attack enemies, evade enemies, gather powerup). Next, it chooses a goal position based on the current plan. Finally, the agent uses an approximate, grid-based A* search plus hard-coded rules to determine a low-level action. The way in which the grid weights are calculated is not described, though the

\[\text{\textsuperscript{10}}\text{The way REALM chooses the goal position is not spelled out in the paper, but it appears to based on hard-coded rules. Neither is it clear how the enemy-related plans translate into goal positions.}\]
paper mentions that positions close to enemies are penalised. As in our approach, the planner does not take enemy movement into account, and the agent leverages knowledge of Mario’s maximum jump distance. Despite these similarities, a major difference between SVP and REALM is that SVP *learns* both the high-level model and the low-level policy, while most of REALM’s decision-making is hard-coded. Further, similarly to the agents discussed above, REALM was designed and evaluated on “flat” levels. Its evolved rules act only on abstract boolean information, such as whether or not there is an obstacle close ahead. This is inappropriate for maze-like domains, where tiny variations in the level structure may be crucial in determining a route’s viability.

**Other Videogame Environments with Maze-Like Levels**

Besides *Infinite Mario*, two popular videogame environments where maze-like navigation problems arise are *DeepMind Lab* [Beattie et al. 2016] and *ViZDoom* [Kempka et al. 2016]. However, existing work in these domains generally differs from our own in one or more key respects. For example, the agent’s likelihood of becoming stuck is often mitigated somehow. In *DeepMind Lab*, Mirowski et al.’s [2016] levels contain “fruit” powerups that encourage exploration, while Mnih et al.’s [2016] levels contain “portals” that effectively remove dead-ends. Further, while the environment itself may be maze-like, the evaluation task may not emphasise long-term navigation. For example, in *ViZDoom*, the commonly used “deathmatch” scenario [Bhatti et al. 2016, Dosovitskiy and Koltun 2016, Lample and Chaplot 2017] emphasises combat ability over long-term navigation. In tasks that do emphasise navigation, the agent is not usually required to find an efficient route on its *first* attempt, as in our experiments. Instead, a common scenario requires the agent to locate the goal on an initial exploratory run, then return to it quickly thereafter [Duan et al. 2016, Jaderberg et al. 2016, Mirowski et al. 2016].

**Other Planning Techniques for Dealing with a Limited Time Budget**

One of the main challenges involved with planning in real-time domains is the fact that the environment is continually evolving. This makes it necessary to plan quickly, as otherwise the environment may change so much while the agent is deliberating that the solution eventually returned is obsolete. In the videogames AI community, study of this problem goes back at least as far as Agre and Chapman’s [1987] work in the *Pengo* domain. Beyond this, the problem also applies more broadly to embodied agent domains, such as robotics. In this chapter, we addressed the challenge of finding long-term plans under time constraints by leveraging temporal abstraction, which allowed us to transform granular planning tasks into much simpler ones. In our experiments with *Infinite Mario*, the tasks were simplified to such an extent that the high-level planner was able to return complete plans within a matter of milliseconds. As such, it was unnecessary to tune the planning algorithm or to consider cases where only a partial plan could be found within the time...
SECTION 4.4: RELATED WORK

limit. However, it is worth mentioning existing work along these lines, since it concerns essentially the same main problem as considered in this chapter (albeit attacking the problem at a low level rather than a high level), and may be useful in extending SVP to domains where the simplified planning problem is still computationally demanding.

One of the most common ways of speeding up heuristic search methods relates to the concept of an admissible heuristic. An admissible heuristic is one that never overestimates the distance to the goal. In the A* algorithm, so long as an admissible heuristic is used, the algorithm will only expand a search node if it is certain that the shortest path to that node has already been found. This requirement is most burdensome for nodes that are far from the start state, so advanced nodes can sometimes take a long time to be expanded. On the other hand, if a non-admissible heuristic is used, the algorithm becomes more lenient with regard to node expansion, requiring only a “good enough” path to be found before a node is expanded. Generally speaking, this reduces the time taken to find the goal, but the path returned is no longer guaranteed to be optimal. The degree of speed up obtained depends on how suboptimal the solution is allowed to be. While in some domains it is possible to improve the performance of granular search agents significantly in this way — in fact, the previous Infinite Mario agents discussed in Section 4.2.3 make several similar approximations that trade optimality for speed — using a non-admissible heuristic is geared more towards addressing action space granularity than time scale granularity, the latter of which is more problematic in our evaluation tasks.

Another way of adjusting heuristic search methods to better handle time constraints is to modify the way that they behave when the search terminates prematurely. Recall from our experiments (Section 4.3) that when the A* agent could not find a complete path within the time budget, it tended to become stuck at local minima of the heuristic, i.e. at dead-ends. One way of addressing this issue is to update the heuristic value for a search node when it becomes clear that it is wrong. This is main idea behind the LRTA* algorithm [Korf 1988; 1990]. Under LRTA*, the heuristic may initially lead the agent to a dead-end, but after enough time the heuristic value at the dead-end will increase to the point that the agent is encouraged to escape. Unfortunately though, as Bulitko et al. [2010] note, this heuristic relearning is typically too slow to be practical in videogames. To address this, they propose an extension to LRTA* that leverages subgoals acquired over an offline pre-training phase. However, their method is only suited to fixed task instances, making it ill-suited to our evaluation task.

Finally, we note that Cserna et al.’s [2018] recent extension to LRTA* has a semantically similar aim to that of our approach; they too aim to mitigate the issue of “dead-ends” in real-time search. However, their definition of a “dead-end” differs from ours. They refer to a state with no successors (a termination state under our terminology) whereas we refer to a state where the agent must retreat because the direct line to the goal is obstructed.
CHAPTER 4: SYNOPTIC VISION PLANNING (SVP)

Model Learning Methods

We are not the first to use a learned model for planning in videogames. However, existing work in this area has mostly focused on learning a granular, frame-level model [Bellemare et al. 2013; 2014, Oh et al. 2015, Chiappa et al. 2017]. Due to videogames’ rapid frame rates, learned models must be extremely accurate in order to prevent small errors from compounding quickly. Remarkably, state-of-the-art models for Atari games can sometimes make qualitatively accurate predictions several hundred frames into the future [Oh et al. 2015, Chiappa et al. 2017]. Currently though, the accuracy of these models is very game-dependent, and they are susceptible to losing track of moving objects. For example, Oh et al.’s [2015] model occasionally forgets bullets in Space Invaders, while Chiappa et al.’s [2017] model sometimes loses track of all foreground objects in Q*bert. Further, both cited methods assume that the environment is deterministic, and they are slower to roll out than the Atari emulator [Machado et al. 2018a]. As the performance of our A* agent for Infinite Mario illustrates, even if these methods learned perfect models, they would need to be rolled out much faster than real-time to achieve sufficient planning depth for substantial navigation problems.

Hierarchical Search Methods

Recently, there have been attempts to improve the scalability of MCTS by incorporating hierarchy [Vien and Toussaint 2015, Bai et al. 2016]. Broadly, this is the same approach we took with the hybrid SVP/A* agent. However, under hierarchical MCTS, the outcomes of temporally extended actions are still simulated via granular rollouts. Under our hybrid approach, granular rollouts are used to determine the route to the next waypoint, but the hierarchical plan itself is calculated via a high-level, learned model.

4.5 Chapter Summary

In this chapter we focused on a type of problem where human players far surpass artificial agents, but which has received little attention in the literature to date: The first time solution of maze-like navigation tasks where the relevant level structure can be seen in advance. Our proposed approach, Synoptic Vision Planning (SVP), learns a transferable local policy and a high-level model over an initial training phase, then calculates abstract plans in real-time via the projection of idealised, “archetypal” states. We applied this approach to randomly generated navigation tasks in Infinite Mario. On tasks with few structural obstacles, our agent’s performance is not as visually spectacular as that of previous agents that exploit access to the game’s internal model, but our method is less reliant on domain knowledge and scales better with goal distance on maze-like tasks. In addition, SVP can be hybridised with granular search, yielding an agent that remains strong at long-term planning but also exhibits precise reactive play.
Exploration in Continuous Control Tasks via Continually Parameterised Skills

A common thread through Chapters 3 and 4 was our focus on environments where feedback is sparse relative to the time scale over which the agent acts. A further commonality was that the environments we considered all had finite action spaces. However, many videogames involve continuous input devices, such as joysticks, steering wheels and motion sensors. Likewise, many real-world control problems have continuous action spaces. Despite the abundance of empirical work concerning exploration in sparse reward, finite action domains, such as the Atari videogames we studied in Chapter 3 [Machado et al. 2015, Bellemare et al. 2016, Osband et al. 2016, Ostrovski et al. 2017, Burda et al. 2018b], there have been relatively few such studies in continuous control tasks. This is possibly due to differing assumptions regarding the reward function: In many discrete action domains, such as board games, Atari and gridworld tasks, the rewards are seen as being intrinsic to the environment, arising naturally from events such as success/failure or changes in the game score. On the other hand, research into continuous control has historically been geared towards robotics, where the reward is more often seen as a component that the experimentalist specifies. Since sample efficiency is paramount in real-world applications, researchers tend to employ reward schemes that provide incremental feedback after every time step. For example, in training a robot agent how to open a door, Gu et al. [2017] reward it incrementally for moving its gripper towards the handle, then for gradually swinging the door ajar. Under such schemes, the importance of exploring efficiently is diminished, since all actions – good and bad – yield useful information.

There are, however, drawbacks to relying on an incremental reward. To begin with, one of the main tenets of reinforcement learning, as opposed to supervised learning, is to avoid the need for prescribed solutions. From this point of view, the reward scheme

\[1\]This chapter covers material previously published in Dann et al. [2018a].
described above is unsatisfying, since it does not require the agent to \textit{conceive} the solution; rather, all it must learn to do is trace out a predetermined trajectory. Moreover, for some tasks it is very difficult to specify the desired behaviour via an incremental reward. For example, in a moving obstacle avoidance task, specifying the desired behaviour via an incremental reward would require one to prescribe a safe path for every possible situation. In this example it would be far preferable if one could just specify a sparse reward that penalises collisions and let the agent arrive at a solution by itself.

Of course, the other side to these points is that specifying tasks via sparse rewards may make learning much slower, and calls for a more sophisticated approach to exploration. Moreover, compared to exploring in discrete action domains, achieving thorough exploration of continuous control environments may be complicated by the degree of action coordination required. To illustrate this point, recall from Chapter 3 that even a uniform random agent for \textit{Montezuma’s Revenge} will occasionally descend ladders and jump across dangerous gaps. However, in navigation tasks with a continuous control component, such as a robot learning to roller skate through a maze, the na"ive exploration strategy of adding isotropic noise\footnote{Noise is \textit{isotropic} when the strength and likelihood of the perturbation is the same in all directions.} to the robot’s joint activations will hardly ever yield any meaningful progress.

Stark examples such as this seem to call for some form of \textit{hierarchical reinforcement learning} [Parr and Russell 1998, Sutton et al. 1999, Dietterich 2000]. For example, suppose that the roller skating agent were first taught a \textit{parameterised skill} [da Silva et al. 2012; 2014b:a] for skating at any desired velocity. If it were then to explore by isotropically varying its target velocity, as opposed to its individual joints, it would reach a wider range of locations and be much more likely to reach the goal.

To emphasise a subtle point, note that the target velocity in this example is non-static. In this regard, the approach suggested differs from traditional applications of hierarchical RL. To give an example of traditional use, da Silva et al. [2014a] describe a robot that is tasked with moving objects around a warehouse. If the robot is equipped with a parameterised “pick up object” skill that can handle objects of different shapes and sizes, it will not have to be retrained when it encounters new objects in the future. Furthermore, it can leverage \textit{temporal abstraction} by decomposing long-term tasks into sequences of subtasks (e.g. “pick up object” \rightarrow “move object” \rightarrow “put down object”). However, note that the skill parameters in the warehouse example (the size and shape of the object) are static for the duration of each subtask. By contrast, the roller skating agent’s target velocity is treated as a continuous action. As such, the “skate at velocity \(v\)” skill \textit{does not introduce temporal abstraction}. Rather, the agent still acts at the most granular time scale, but it learns over an abstract action space where it does not have to ensure the coordination of individual joints. To distinguish the two types of parameterised skill exemplified here, we refer to the “pick up object” as a \textit{ statically parameterised skill}
and the skating skill as a *continually parameterised skill*.

While there has recently been some work on exploration in sparse reward domains with continuous actions [Houthooft et al. 2016, Plappert et al. 2017], we are not aware of any previous research that proposes using continually parameterised skills for this purpose. In robotics, *Dynamic Movement Primitives (DMPs)* [Schaal et al. 2005] are commonly used to parameterise movement, but they are merely low-level building blocks and do not, by themselves, bring the level of action abstraction required for tasks such as our robot skating example. At the other extreme, in synthetic domains the low-level control layer is often abstracted away completely. For example, Masson et al.’s 2016 simulated soccer agent is given direct access to parameterised *shoot* and *dribble* actions, but there is no underlying action space of joint actuations or limb movements over which to compare learning. This leaves a research gap, because such skills would in reality need to be trained, and no precise formalism for hierarchical learning with continually parameterised skills has previously been laid out. Further, there may be undiscovered practical issues that arise when every layer is trained, rather than hardcoded, and it is unclear what benefit, if any, this type of hierarchical learning might offer in domains where the need for hierarchy is less obvious, e.g. in domains where exploring over primitive actions *is* sufficient to discover rewards.

In light of these points, the first contribution of this chapter is to provide a formal framework for the approach sketched out thus far. Our second contribution is an empirical case study that aims to clarify how our approach can be practically applied, and to study its properties in a domain where the need or otherwise for exploiting hierarchy in the action space is non-obvious. Since we could not find any domains in the literature that constitute an interesting middle-ground, we created our own based on the classic arcade game *Asteroids*. Within this environment there are two tasks defined by sparse rewards: a *goal-seek* task where the only feedback provided is a +1 reward for reaching a goal zone, and a *keep-alive* task where the only non-zero reward is a −1 penalty for colliding with an asteroid. The agent must coordinate two continuous-valued thrusters to steer the ship, while contending with inertia. We compare a hierarchical agent that is equipped with a parameterised skill for controlling the ship’s velocity versus a *flat learner*, i.e. an agent that does not exploit hierarchy in the action space and controls the thrusters directly. While both agents make reasonable progress, we demonstrate that the hierarchical agent tends to explore actions with greater task relevancy than those chosen by the flat agent. As a result, its long-term performance is superior on both tasks, and it is less troubled by trivial local optima in the keep-alive task.

---

3As previously mentioned, continually parameterised skills do not introduce temporal abstraction, so this problem is different to that of hierarchical learning with *options* [Sutton et al. 1999].

4https://en.wikipedia.org/wiki/Asteroids_(video_game)
CHAPTER 5: EXPLORATION IN CONTINUOUS CONTROL TASKS VIA CONTINUALLY PARAMETERISED SKILLS

5.1 Hierarchical Control via Policy Composition

In this section, we formally describe our framework for hierarchical learning via continually parameterised skills. In Section 5.1.1, we explain how the various layers connect and how high-level actions are translated into primitive actions. In Section 5.1.2, we explain how the hierarchy is trained. Finally, in Section 5.1.3, we explain how the hierarchy can be used to generate efficient, non-isotropic noise over the primitive action space.

5.1.1 Layer Organisation

Under our proposed framework, action selection proceeds in a top-down manner: Starting from the most abstract level at the top of the hierarchy, each layer selects the parameters for the layer underneath until, finally, the bottom layer selects a primitive action. To give a concrete example, a car-driving policy might be decomposed such that the top layer selects the car’s target direction and speed, an intermediate layer translates this into target positions for the steering wheel, accelerator and brake, and the bottom layer determines joint movements required by the driver of the car.

In the above example, note that the input and output types differ across layers. The top layer receives the current state and returns parameters for the layer below. The intermediate layer receives a state-parameter pair and returns further parameters. Finally, the bottom layer also receives a state-parameter pair, but returns a primitive action. To distinguish these three types of layers, we make the following definitions:

**Definition 2.** A base continually parameterised skill is a function \( \alpha_1 : \mathcal{S} \times \mathcal{T}^1 \rightarrow \mathcal{A} \) that maps a state-parameter pair to a primitive action.

**Definition 3.** An intermediate continually parameterised skill is a function \( \beta_i : \mathcal{S} \times \mathcal{T}^i \rightarrow \mathcal{T}^{i-1} \) that maps a state-parameter pair to the parameters of an underlying continually parameterised skill, which itself may be either base or intermediate.

**Definition 4.** A parameter policy is a function \( \phi_i : \mathcal{S} \rightarrow \mathcal{T}^{i-1} \) that maps a state to the task parameters of an underlying continually parameterised skill. Again, the underlying layer may be either base or intermediate.

For the remainder of the chapter, we omit the term “continually parameterised” and just refer to “base skills” and “intermediate skills” for brevity.

Given a parameter policy, a base skill and a suitable arrangement of intermediate skills (such that the layers form a descending hierarchy), a deterministic policy over actions \( \mu : \mathcal{S} \rightarrow \mathcal{A} \) can be recovered through composition:

\[
\mu(s) = \alpha_1(s, \cdot) \circ \phi_2(s) \quad \text{[two tiers]} \quad (5.1)
\]

\[
\mu(s) = \alpha_1(s, \cdot) \circ \beta_2(s, \cdot) \circ \beta_3(s, \cdot) \circ \ldots \circ \beta_{n-1}(s, \cdot) \circ \phi_n(s) \quad \text{[n tiers, } n > 2\text{]} \quad (5.2)
\]

\[\text{Our use of the term “parameter policy” follows similar terminology introduced by Masson et al. [2016].}\]
Note that the approach can be modified to allow non-determinism by changing the layer definitions so that each layer maps to a probability density over its output space. The reason we have chosen a deterministic approach is because, in our experiments, we train each layer using the Deep Deterministic Policy Gradient (DDPG) algorithm [Lillicrap et al. 2016], which assumes that the policy being trained is deterministic. Also note that the model is not designed to handle discrete switching between different types of behaviour (e.g. dribbling a soccer ball / kicking a ball / heading a ball). We envisage that any discrete switching would be handled via the Q-PAMDP algorithm of Masson et al. [2016], with an additional Q-learning layer sitting atop our hierarchy.

5.1.2 Training the Hierarchy

We train the skill hierarchy in a bottom-up manner, first training low-level skills, then training higher level skills that exploit the skills already trained. Each successive layer is trained from a progressively more abstract MDP. In the discussion that follows, we refer to MDP of the original task as the native MDP, and denote the \(i\)th layer’s MDP as \(\mathcal{M}_i = \langle S_i, A_i, P_i, R_i, \gamma \rangle\) with \(i \in \{1 \ldots n\}\). The components of each \(\mathcal{M}_i\) are set as follows:

**State space:** The parameter policy, \(\phi_n\), is trained over the native state space, i.e. \(S^n = S\). For the remaining skill layers, the state space is supplemented with the skill’s parameter space so that the skill can perceive the objective passed from the layer above. That is, \(S^i = S \times T^i\) for \(1 \leq i < n\). Note that this is similar to how we trained the local movement skill in Chapter 4, and it means that we are essentially treating the parameterised skills as universal value function approximators (Section 2.4.3).

**Action space:** The base layer acts over the primitive action space, while all other layers act over the parameter space one tier below, i.e. \(A^1 = A\) and \(A^i = T^{i-1}\) for \(1 < i \leq n\).

**Transition model:** Transition probabilities in the modified MDP follow by descending the skill hierarchy and applying the eventual primitive action. For example, at the \(i\)th intermediate layer this works as follows: Let \(\tau_i^t\) be the parameters passed to the layer at time \(t\). The primitive action is calculated as:

\[
a_t = \alpha_1(s, \cdot) \circ \beta_2(s, \cdot) \circ \beta_3(s, \cdot) \circ \ldots \circ \beta_i(s, \tau_i^t)
\]

The evolution of the augmented state in \(S \times T^i\) then proceeds in two parts: The native state evolves by applying the primitive action in the native MDP. The skill parameters evolve differently depending on which level of the hierarchy is currently being trained. When training the \(i\)th layer, the parameters are held fixed, i.e. \(\tau_{i+1}^t = \tau_i^t\). (For example, during training of a “skate at velocity \(v\)” skill, \(v\) would be held fixed.) When training the
higher layers, the skill parameters evolve as per the action selected one layer above.

**Reward function:** The parameter policy at the top of the hierarchy is designed to tackle an action-abstracted version of the native task, and is therefore trained from the native reward that arises after applying the primitive action at the base layer. For each remaining parameterised skill layer, one is free to choose an auxiliary reward that is suited to the skill in question. Importantly, even if the native task is hard to specify via an incremental reward, parameterised skills’ objectives are often easy to specify in such terms. For example, if the objective of a parameterised skill is to bend a joint to \( \theta \), the agent can be incrementally rewarded for making progress towards that angle. (We provide a more specific example of such a scheme in Section 5.3.)

**Discount:** Since the parameter policy is intended to tackle the native task, it must be trained using the native discount. However, for the lower layers one is free again to choose appropriate settings for the skills in question. In doing so, it should be noted that the temporal horizon of a parameterised skill may be relatively short compared to the time frame of the overall task. For example, in a car driving navigation task, the overall goal location might be a large distance away, but a skill for adjusting the steering wheel to a target angle would presumably act over a short time scale. Therefore, in this case it would make sense to use a sharper discount for the steering skill than for the overall task.

### 5.1.3 Exploring Efficiently via Continually Parameterised Skills

Many RL algorithms for continuous control tasks require the policy being trained to be stochastic. A common way of representing stochastic polices is via function approximators, such as neural networks, that output the mean and variance of their respective action distributions [Schulman et al. 2015; 2017]. Under this approach, the noise (i.e. the offset from the mean action) is temporally uncorrelated, meaning that the noise at one time step has no bearing on the noise at the next step.

Unfortunately, a problem with temporally uncorrelated noise is that it tends to nullify itself. (Informally, randomly jiggling a car’s steering wheel in an unbiased way will tend to leave the car on its original heading.) This will be problematic if rewards are sparse and can only be discovered via persistent exploration in one direction. A second, more subtle reason why temporally uncorrelated noise is ill-suited under our framework is that sudden, sharp changes in a skill’s parameters may break its learned assumptions. For example, when a person lifts one foot off the ground while walking, they are not expecting the target direction to change suddenly and leave them wrong-footed. Similarly, the acts of swinging one’s arms and lowering one’s head in the direction of travel are designed to build momentum, which is predicated on the target direction remaining similar from one moment to the next.
For these reasons, our hierarchy is best trained via an off-policy actor-critic algorithm [Degris et al. 2012, Silver et al. 2014]. Recall from Section 2.2.1 that off-policy methods allow an agent to learn one policy while using a separate, potentially more sophisticated policy for exploration. One such off-policy method, which is our method of choice in this work, is the well-known DDPG algorithm [Lillicrap et al. 2016]. In DDPG, noise is generated via an Ornstein-Uhlenbeck process [Uhlenbeck and Ornstein 1930]. Since the time scale in reinforcement learning is never truly continuous, the discretised version of the process, $X_t$, is used:

$$X_{t+1} = X_t + \theta(\mu - X_t)\Delta t + \sigma \sqrt{\Delta t} \mathcal{N}(0, 1) \quad (5.4)$$

where $\mu$ is the process mean, $\theta > 0$ controls the rate of mean reversion, $\sigma > 0$ controls the rate of divergence, $\Delta t$ is the time increment between steps, and $\mathcal{N}(0, 1)$ is the standard normal distribution.

This process yields temporally correlated noise, and on physical control tasks it meets Wawrzynski’s 2015 requirement that the amount of randomness in the state trajectory should not depend on the granularity of the time scale. While it does not intrinsically deliver the kind of coordinated noise that we argued for with the robot skating example at the beginning of this chapter (since Ornstein-Uhlenbeck noise is usually applied independently across action dimensions), this is what our hierarchy aims to achieve. The key point is that adding isotropic noise over a parameter space will in general yield non-isotropic noise at the primitive level. For example, if the target direction and speed of a “skate at velocity $v$” skill are varied independently according to Equation 5.4, the resultant distribution of primitive actions will include those required to attain various velocities, but, assuming the skill was well-trained, it will exclude “bad” actions that tend to destabilise the agent. A clear illustration of this effect can be seen later in our results (Figure 5.5).

### 5.2 The Rocketship Domain

Over the coming sections, we conduct a case study that aims to show how our approach may be practically applied, and to illustrate some of its pros and cons versus ordinary, flat learning. In order for this comparison to be non-trivial, we sought an experimental domain where the need for hierarchy is not completely obvious (as in the example of a roller skating robot), but where, conversely, the task is not so straightforward that hierarchy is clearly an unnecessary overhead. Since we were unable to identify a suitable domain in the literature, we designed our own: the Rocketship domain. The Rocketship domain is inspired by the classic arcade game, Asteroids, and is intentionally designed to have the following properties:

- Natural objectives exist that are far easier to specify through a sparse reward than a continual reward.
• Stable control requires a high degree of precision at the primitive action level.

• Local optima may be significant issue. (We save a detailed explanation of this phenomenon for where it manifests in our experimental results – see Section 5.4.3).

As its name suggests, the Rocketship domain involves an agent controlling a rocket. The rocket has two thrusters that are capable of outputting thrusts between 0 and 1, yielding a continuous action space of $\mathcal{A} = [0, 1] \times [0, 1]$. The torque on the rocket is equal to the thrusters’ differential, while forward thrust is proportional to the shared output:

$$\text{Torque} = \text{LeftThrust} - \text{RightThrust}$$ (5.5)

$$\text{ForwardThrust} = 2 \times \min(\text{LeftThrust}, \, \text{RightThrust})$$ (5.6)

The rocket quickly spins out of control if there is a persistent, one-way imbalance in the thrusters. To make control slightly easier, the rocket is limited by drag so that its linear and angular velocities are bound and it can slow down by reducing thrust.

As in Asteroids, the environment’s topology “wraps” so that travelling out one side of the screen brings the rocket to the opposite side. However, for the sake of simplicity, the rocket is unarmed and both the rocket and asteroids collide via circular hit boxes.

The state representation consists of the following:

• The distance from the rocket to each asteroid, encoded so that nearer asteroids send larger signals (1 input per asteroid).

• The direction of each asteroid (2 inputs per asteroid).

• The rocket’s linear and angular velocities (3 inputs).

• Where applicable, the radial and tangential components of the asteroids’ velocities (2 inputs per asteroid, see Figure 5.1a).

All directional information is captured relative to the rocket’s current orientation and wrapped according to the domain’s topology.

Within the Rocketship domain there are two scenarios: the goal-seek task and the keep-alive task. Both tasks have sparse reward schemes, but the task difficulties are configured such that even the non-hierarchical agent receives sufficient feedback to improve.

### 5.2.1 Goal-Seek Task

In the goal-seek task, the asteroids are stationary and the agent’s aim is to avoid them while navigating the rocket to a goal zone (see Figure 5.1b). There is a single +1 reward for reaching the goal, but if the rocket crashes then the episode terminates with 0 reward.
Figure 5.1: The Rocketship domain. Subfigure (a) shows how asteroid velocities are represented via radial and tangential components. Subfigure (b) shows one random spawn point in the goal-seek task.

The asteroids and goal zone are repositioned randomly at the start of each episode. To give the agent visibility of the goal zone, three additional variables indicating its distance and direction are included in the state representation.

5.2.2 Keep-Alive Task

In the keep-alive task, the asteroids are no longer static and there is no longer a goal zone. Instead, the agent’s objective is just to avoid crashing for as long as possible. The only non-zero reward provided is a $-1$ penalty for hitting an asteroid, which terminates the episode. Asteroid positions and velocities are randomised at the beginning of each episode. Occasionally, this results in lucky spawns where it is relatively easy to avoid crashing. (For example, sometimes the asteroids end up moving in parallel and thus rarely collide, making the avoidance task less chaotic.) To diminish the effect of these lucky spawns, episodes are additionally terminated after a fixed maximum episode length.

5.2.3 Level Configuration

To help gauge agents’ robustness to the size and number of asteroids present, we designed three levels for each task. The settings for each level are shown in Table 5.1. To put the asteroids’ size in context, those shown in Figure 5.1b have a radius multiplier of 1.0.
5.3 Applying Our Framework to *Rocketship*

In this section we explain how we applied our framework to the *Rocketship* domain.

5.3.1 Skill Hierarchy

For simplicity, we adopt a two tier setup.\(^6\) First, the agent learns a base skill for flying the rocket at a target velocity. Next, it tackles the goal-seek and keep-alive tasks by learning a parameter policy for controlling the base skill’s target velocity. The velocity parameter is represented in polar coordinates, \(\tau = (\rho_{\tau}, \theta_{\tau})\), where \(\rho_{\tau}\) is the target speed and \(\theta_{\tau}\) is the target heading (relative to the rocket’s current orientation). Accordingly, under the formalism of Section 5.1.1, the velocity skill is a function \(\alpha_1 : S \times T^1 \to A\), where:

- \(T^1 = [0, \text{maxSpeed}] \times (-180^\circ, 180^\circ)\), where \(\text{maxSpeed}\) is the theoretical maximum rocket speed, taking drag into account.
- \(A = [0, 1] \times [0, 1]\) is the primitive action space of left and right thruster outputs.

Similarly, the parameter policy takes the form \(\phi_2 : S \to T^1\), and the agent’s full policy \(\mu : S \to A\) is recovered via \(\mu(s_t) = \alpha_1(s_t, \cdot) \circ \phi_2(s_t)\).

5.3.2 Training the Velocity Skill

One small implementation problem that arises in *Rocketship* is that the size of the state space is variable across tasks and levels (owing to differing asteroid counts and the existence/non-existence of the goal zone). To avoid having to train the velocity skill multiple times, we trained it in an empty game world and then excluded the asteroids and goal zone from the representation when applying it to the goal-seek and keep-alive tasks. As such, the parameter policy is entirely responsible for obstacle avoidance.

---

\(^6\)While we have adopted only a two tier hierarchy here, it is worth noting the following: If a two tier hierarchy outperforms non-hierarchical learning in certain cases, the possibility of additional tiers yielding further benefit follows by induction, since either layer in the two-tier hierarchy might itself be better trained via decomposition.
The velocity skill was trained on fixed-length episodes lasting 1,500 frames (150 agent actions) with a constant target velocity $\tau_{target} = (\rho_{target}, \theta_{target})$ set randomly at the start of each episode. The agent was rewarded for making incremental progress towards the target velocity, as per the following equation:

$$r_t = \frac{\rho_{max} - \|\rho_{target} - \rho_{actual}\|}{\rho_{max}} \times \frac{180 - \|\theta_{target} - \theta_{actual}\|}{180} \times (1 - \gamma)$$  \hspace{1cm} (5.7)

where:

- $\rho_{max}$ is the maximum possible speed (calculated according to the domain dynamics).
- $\rho_{actual}$ and $\theta_{actual}$ are the rocket’s actual speed and heading respectively at time $t$.
- $\gamma$ is the discount rate.

Breaking this down, the first two factors incentivise the agent to match its actual velocity to the target velocity. The multiplicative form of the reward means that there is more to be gained by addressing whichever component is farther from its desired value. The final factor of $(1 - \gamma)$ ensures that the discounted sum of expected rewards is capped at 1.

### 5.3.3 Policy Representation and Training Parameters

Both the velocity skill and parameter policy were trained via the Deep Deterministic Policy Gradient (DDPG) algorithm, using experience replay and soft target updates to stabilise learning, as described by Lillicrap et al. [2016]. However, despite the name of the training algorithm, there was nothing “deep” about our network architecture. Actor and critic functions were parameterised as feed-forward neural networks with two hidden layers each comprising 60 rectified linear units. Since having the agent default to a thrust of zero at the start of learning would be problematic for reward discovery, we squashed the outputs of the velocity skill’s actor network to $(-1, 1)$ via tanh activations and interpreted outputs of 0 as thrusts of 0.5. Similarly, for the parameter policy, the network’s output space of $[-1, 1] \times [-1, 1]$ was mapped linearly on to the range of valid target velocities, $[0, maxSpeed] \times [-180^\circ, 180^\circ]$. Critic outputs were left unsquashed. As per Lillicrap et al. [2016], final layer weights were randomised very close to zero to ensure that, initially, both the actors’ outputs and the policy gradient estimates would be small.

Since all asteroids in the Rocketship domain are identical, a reordering of the asteroids in the state representation should not matter to the agent. To ensure that this would be so, we averaged the outgoing weights from like input neurons after each minibatch. In preliminary experiments, we found that this greatly increased the learning speed.

Additional training details are listed in Table 5.2. Note that we used a much more aggressive discount to train the velocity skill ($\gamma = 0.8$) than we used to train the parameter policy ($\gamma = 0.99$). Our rationale for this was that reaching a target velocity is a relatively short-term aim compared to the parameter policy’s objective of tackling the overall task.
CHAPTER 5: EXPLORATION IN CONTINUOUS CONTROL TASKS VIA CONTINUALLY PARAMETERISED SKILLS

<table>
<thead>
<tr>
<th>Training Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame rate</td>
<td>25 frames per second</td>
</tr>
<tr>
<td>Action duration</td>
<td>10 frames</td>
</tr>
<tr>
<td>Training Time</td>
<td>$1.5 \times 10^8$ frames for both the velocity skill and the parameter policy ($3 \times 10^8$ frames total)</td>
</tr>
<tr>
<td>Maximum episode length</td>
<td>Velocity skill training: 1,500 frames</td>
</tr>
<tr>
<td></td>
<td>Parameter policy training: 15,000 frames</td>
</tr>
<tr>
<td>Discount factor, $\gamma$</td>
<td>Velocity skill training: 0.8</td>
</tr>
<tr>
<td></td>
<td>Parameter policy training: 0.99</td>
</tr>
<tr>
<td>Actions per TD error</td>
<td>4</td>
</tr>
<tr>
<td>Actor learning rate</td>
<td>$1 \times 10^{-7}$</td>
</tr>
<tr>
<td>Critic learning rate</td>
<td>$2 \times 10^{-5}$</td>
</tr>
<tr>
<td>Replay buffer size</td>
<td>300,000</td>
</tr>
<tr>
<td>Minibatch size</td>
<td>500</td>
</tr>
<tr>
<td>Soft target updates</td>
<td>Every 10,000 frames with an update weighting of 0.005</td>
</tr>
<tr>
<td>Noise</td>
<td>Ornstein-Uhlenbeck with $\mu = 0$, $\theta = 0.1$, $\sigma$ linearly annealed from 0.08 to 0.02 over $10^8$ frames</td>
</tr>
<tr>
<td>Final layer weight</td>
<td>Actor: (-0.001, 0.001)</td>
</tr>
<tr>
<td>initialisation range</td>
<td>Critic: (-0.00001, 0.00001)</td>
</tr>
</tbody>
</table>

Table 5.2: The settings used in our experiments.

5.4 Results and Discussion

In this section we present the results of our approach in the Rocketship domain. To gauge the effectiveness of decomposing the agent’s policy via the velocity skill, we compare against “flat” reinforcement learning, i.e. where the agent learns a direct mapping from states to primitive actions.

5.4.1 Noise Adjusted Flat Benchmark

Where applicable, we trained the flat agent using the same settings described in the previous section. However, one setting that is not possible to configure identically is the amount of noise injected into the agent’s policy. As explained in Section 5.1.3, adding noise at the skill parameter level yields a different-shaped noise distribution versus adding noise directly at the primitive level. In an attempt to balance the agents’ effective noise levels, we recorded their initial success and crash rates on Level A of the goal-seek task when acting purely through Ornstein-Uhlenbeck noise (with $\sigma$ at its initial value of 0.08). As the results in Table 5.3 show, the hierarchical agent recorded higher rates in both categories. The reason for this became clear from observing the agents’ behaviour: Loosely speaking,
the hierarchical agent’s trajectories tended to be smooth and wavy, while the flat agent spun out more and took longer to veer from the start location.

Intuitively, the higher rate of feedback provided to the hierarchical agent at the start of training ought to give it an advantage. However, this is arguably just a result of the hierarchical agent having a lower effective noise level. Indeed, after scaling the flat agent’s noise down by 30%, it achieved marginally higher rates than those of the hierarchical agent (see the third row of Table 5.3). Therefore, in the experiments that follow, to quantify the extent to which performance differences are attributable to different noise magnitudes, we benchmark against both versions of the flat agent.

### 5.4.2 Goal-Seek Task

On the goal-seek task, the hierarchical and noise adjusted flat agents clearly did benefit from their higher initial feedback rates, progressing faster early on compared to the un-adjusted flat agent (see Figures 5.2a – 5.2c). However, the long-term performances of the two flat agents were virtually identical, suggesting that the 30% noise adjustment was of little consequence in the long run. On the other hand, the hierarchical agent maintained a persistent edge across all three levels. Compared to the stronger of the flat agents on each level, its average return by the end of training was 6% greater on Level A, 4% greater on Level B, and 7% greater on Level C. However, it is worth noting that these advantages are relative to the mild discount of $\gamma = 0.99$; sharper discounting would have seen the hierarchical agent achieve a greater edge in percentage terms, owing to its tendency to fly faster than the flat agents (as we explain shortly).

Given that the hierarchical agent ultimately acted over the same underlying action space as the flat agents, it is interesting that its long-term performance was superior. One possible explanation is ease of policy representation: In order for the hierarchical agent to head towards the goal zone, all it had to do was set its target velocity equal to the direction of the goal. This was perhaps an easier mapping to learn than a direct map from states to thruster outputs. However, one would expect this effect to be most significant early in training. The fact that the hierarchical and noise adjusted agents initially progressed at a similar rate seems to discount this explanation.

Delving further, it appears that the cause of the hierarchical agent’s outperformance was that it learned superior control over the rocket’s speed. This is evidenced by a more

<table>
<thead>
<tr>
<th>Agent</th>
<th>Successes / min.</th>
<th>Crashes / min.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchical</td>
<td>$0.22 \pm 0.01$</td>
<td>$0.84 \pm 0.01$</td>
</tr>
<tr>
<td>Flat</td>
<td>$0.18 \pm 0.01$</td>
<td>$0.69 \pm 0.01$</td>
</tr>
<tr>
<td>Flat, noise adj.</td>
<td>$0.24 \pm 0.01$</td>
<td>$0.91 \pm 0.01$</td>
</tr>
</tbody>
</table>

Table 5.3: Pre-training success and crash rates on Level A.
detailed view of the learning curves, where the hierarchical agent’s curves exhibit interesting patterns that were absent from the flat agents’ curves. For example, see Figure 5.3, which plots the hierarchical agent’s average return alongside its average success and crash rates from a single training run on Level C. At around $1 \times 10^8$ training frames, there is a noticeable dip in success and crash rates, caused by the agent starting to reduce the rocket’s speed. This also created a temporary drop in average return, but seemingly allowed the agent to learn harder tasks that required more careful navigation. As the agent became more competent at these tasks, it learned to fly the rocket faster again, while retaining an improved success-to-crash ratio. This pattern then repeated, albeit for less net improvement, starting at around $1.2 \times 10^8$ training frames. In contrast, both flat agents flew at a roughly constant speed throughout the course of training, learning how to steer but not how to speed up or slow down drastically.

The reason the hierarchical agent learned better control of its speed is most likely because it explored different speeds more thoroughly and in a more stable manner. For the flat agents to explore extreme speeds, their thrusters’ independent noise processes had to generate strong noise at the same extreme simultaneously, since firing only one thruster at an extreme merely destabilises the rocket. However, the hierarchical agent only needed strong noise in a single dimension (the speed parameter) and small-to-moderate noise in the other dimension (the angle parameter) to experience the same effect.

![Figure 5.2: Average return achieved from the initial state of the goal-seek task, averaged over five runs.](image)
Figure 5.2: Average return achieved from the initial state of the goal-seek task, averaged over five runs.
5.4.3 Keep-Alive Task

The hierarchical agent also had a clear advantage on the keep-alive task (see Figures 5.4a – 5.4c). By the end of training, its crash rate was 12% lower than that of the noise adjusted flat agent on Level D, 18% lower on Level E, and 17% lower on Level F. Across all levels, its edge over the unadjusted flat agent was even greater. As explained below, we believe that this advantage was largely attributable to factors: (1) The hierarchical agent was less influenced by the trivial local optima of “spinning out” the rocket. (2) The hierarchical agent more thoroughly explored the range of rocket speeds.

Trivial local optima

A significant training issue in the keep-alive task is the presence of trivial local optima in the solution space. This is illustrated by Table 5.4, which shows the agents’ initial crash rates.

<table>
<thead>
<tr>
<th>Agent</th>
<th>Initial crash rate (per minute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchical</td>
<td>1.43 ± 0.01</td>
</tr>
<tr>
<td>Flat</td>
<td>1.39 ± 0.03</td>
</tr>
<tr>
<td>Flat, noise adjusted</td>
<td>1.40 ± 0.02</td>
</tr>
<tr>
<td>Constant thrust of (+1, -1)</td>
<td>1.15 ± 0.01</td>
</tr>
</tbody>
</table>

Table 5.4: Crash rates prior to learning on Level D.
Figure 5.4: Crash rates versus training frames on the keep-alive task. Curves are averaged over five training runs.
Exploration in continuous control tasks via continually parameterised skills

Figure 5.4: Crash rates versus training frames on the keep-alive task. Curves are averaged over five training runs.

rates on Level D. At the start of training, all agents collide with an asteroid roughly 1.4 times per minute, while a policy that outputs a constant thrust of (+1, −1) crashes only 1.15 times per minute. The reason for this is that heavily imbalancing the thrusters causes the rocket to spin on the spot, which is safer than flying about randomly. Since introducing even a moderate steering bias to an untrained policy causes a slight reduction in crash rate, all agents were prone to developing a bias early in training.

While none of the agents degenerated to the point of learning a permanent (+1, −1) or (−1, +1) thrust, the flat agents in particular developed strong biases that took some time to unlearn. On the other hand, since the velocity skill was trained from a reward that penalised oversteering (refer back to Equation 5.7), the hierarchical agent was less prone to building large angular momentum, even when strong noise was applied in the steering action dimension. As such, it experienced less benefit from developing a steering bias, and even though it still typically developed a slight bias, the dampening effect of the velocity skill mitigated its impact.

Exploration of rocket speed

As in the goal-seek task, the hierarchical agent may also have benefited from its more thorough exploration of rocket speeds. Figure 5.5a shows the primitive action distribution
Figure 5.5: Primitive action distributions for the hierarchical and flat agents when biased to turn to the right.

of an untrained hierarchical agent with a fixed right turn bias of 20%, or 36°, while Figure 5.5b shows the distribution for a noise-adjusted flat agent with the same level of bias.\(^7\) The extent of the agents’ speed exploration is indicated by the amount of spread along the diagonal. While both distributions correspond to looping trajectories, the flat agent’s loops tended to be closer to regular circles, whereas the hierarchical agent explored a greater diversity of seashell-shaped trajectories, which we believe helped it to learn additional modes of escape from asteroids.

5.5 Related Work

Recently, there have been several attempts to address exploration in sparse reward continuous control tasks via non-hierarchical means. As one might expect, the idea of intrinsic motivation, as detailed in Chapter 3, can also be applied to continuous control tasks. Houthooft et al. [2016] utilise the errors obtained from training a model of the environment as an intrinsic reward. They demonstrate the effectiveness of their approach by applying it to reward-sparsified versions of classical control tasks such as *mountain car* and *cart-pole*. Tang et al. [2017] tackle a similar set of tasks via count-based bonuses derived from a hashing of the state space.\(^8\) Plappert et al. [2017] propose an alternative way of generating exploratory noise, whereby rather than perturbing the action distribution

\(^7\)In both cases, the mean thruster differential is equal to 0.19.

\(^8\)A more detailed discussion of Tang et al.’s 2017 method can be found in Chapter 3, as it bears several similarities to our state space partitioning scheme based on exploration effort. Refer to Section 3.4.
directly, noise is added to the policy’s internal parameters. Note, however, that while the terminology used in their work is similar to our own (we both discuss perturbing policy “parameters”), there is a significant difference between varying the weights inside a neural network and varying a policy’s parameterised goal.

Despite the fact that leveraging skills for exploration is an old idea, the subfield of research into parameterised skills is relatively recent. So far, more work has focused on acquiring parameterised skills than applying them [da Silva et al. 2012; 2014b;a, Fabisch and Metzen 2014, Kober et al. 2012]. Probably the most closely related research to ours is Masson et al.’s 2016 work on parameterised action Markov decision processes (PAMDPs). The authors propose an algorithm that interleaves the learning of parameter policies with learning when to switch between different types of discrete behaviour. To give an example, in a simplified soccer domain they train one parameter policy for dribbling the ball around the keeper, and another for shooting at goal. The former policy parameterises an action “dribble the ball to (x, y)”, while the latter parameterises an action “shoot at position z along the goal line”. Alternating with this training, the agent learns a discrete policy for determining when to switch between dribbling and shooting. While clearly related to the ideas presented here, a significant difference between their setting and ours is that the parameterised actions in their work are hardcoded into the environment. That is, there is no primitive action space underlying the parameter space. As such, the question of whether hierarchical learning over parameterised actions offers any benefit compared to flat learning does not arise.

In this work, we chose to model the velocity skill via a neural network, including the target velocity in its input alongside the state representation. In other words, we modelled the skill as a universal function approximator (UVFA) (Section 2.4.3). However, we are not the first to apply UVFAs to videogames. For example, Dosovitskiy and Koltun [2016] applied them to the classic first-person shooter, Doom, training an agent to adapt to a flexible goal that was expressed via priority weights for killing enemies, maximising health and conserving ammunition. In fact, the paper that introduced UVFAs applied them to Ms. Pacman, and even went so far as to mention that they can be used to derive temporally abstract options [Schaul et al. 2015]. However, to the best of our knowledge, we are the first to propose using continually parameterised UVFAs for policy decomposition, which, as we have noted throughout this chapter, yields a distinct effect and does not introduce temporal abstraction.

In most previous work on parameterised skills, they are not modelled via UVFAs.

---

9This approach is itself very similar to noisy nets [Fortunato et al. 2017], but noisy nets were only applied to the discrete action domain of Atari in the paper that introduced them.

10Parameterised actions are essentially a superclass of parameterised skills. Parameterised skills can be used as parameterised actions, but in PAMDP theory the internal workings of an action are effectively a black box, i.e. a parameterised action does not have to derive from an underlying policy.

11Dosovitskiy and Koltun [2016] do not explicitly use the term “UVFA”, though their approach is the same conceptually.
A more common approach, as introduced by da Silva et al. [2012], is to first learn a set of fixed skills for differently-parameterised tasks, then learn an interpolation from task parameters to policy parameters. However, this approach is sensitive to the type of policy parameterisation used. Previous work along these lines [da Silva et al. 2012; 2014b;a] uses dynamic movement primitives, which have relatively few learned parameters. (As a reference point, there are 37 learned parameters in da Silva et al.’s 2012 experiments.) On the other hand, we chose neural networks. Our reason for this was that, unlike DMPs, neural networks are capable of representing arbitrary polices. However, they also typically contain thousands of learned weights, and successive training runs may yield completely different weights, even under fixed task parameters. While the mapping proposed by da Silva et al. [2012] is much more sophisticated than a simple linear interpolation, it is unlikely to function well in this setting.

5.6 Chapter Summary

In this chapter we proposed a novel application of parameterised skills. Traditionally, skills are employed for temporal abstraction and knowledge reuse, but we demonstrated a further possible application: In continuous control tasks, decomposing a policy into a hierarchy of continually parameterised skills may help in improving exploration at the primitive action level.

To show how our approach can be practically applied, we presented a detailed case study in the Rocketship domain. Results in both the goal-seek and keep-alive tasks provided evidence that our approach improved the quality of exploration. The hierarchical agent explored a greater range of speeds than the flat agents, and maintained more control when it explored turns. In the keep-alive task, the hierarchical agent was less affected by “spin out” local optima because it was less exposed to such states in the first place, and because it was more likely to discover escape manoeuvres when it did build large angular momentum. As a result of these advantages, the hierarchical agent outperformed the flat agents on both tasks.

To be clear, we would not expect our approach to be beneficial on all tasks, especially on simple tasks where there is clearly no need for hierarchy. As under all well-known hierarchical learning frameworks, our approach entails the additional overhead of skill training, and the upper layers in the hierarchy may degrade if the lower levels are not well-trained. Nonetheless, it is interesting that despite the imprecision incurred by using a trained skill in our experiments, and despite the fact that hierarchy was not obviously necessary in the Rocketship domain (as the flat agents were able to find rewards and improve), the hierarchical agent still outperformed non-hierarchical learning. This highlights the non-triviality of determining when hierarchical policy decomposition is likely to be useful, which we believe is an interesting direction for future research.
Conclusion

In videogames, as in many real-world problems, agents must act over granular time scales and continuous action spaces. These properties pose distinct challenges that rarely arise in older games AI benchmarks, such as classical tabletop games. Over the course of this thesis, we introduced several methods for achieving efficient learning and planning in videogames. Each of these methods leveraged task decomposition in some way. In this chapter, we recap our contributions (Section 6.1), discuss some broad limitations of our work (Section 6.2), then finally propose some directions for future research (Section 6.3).

6.1 Summary of Contributions

Returning to the research questions laid out in Chapter 1, let us now summarise our main findings and highlight the novel contributions made over the course of this thesis:

R1: In videogames with sparse rewards, what learning methods can be developed to identify subgoals autonomously and improve exploration efficiency?

Autonomous subgoal acquisition is a long-studied problem in hierarchical reinforcement learning, being particularly challenging in domains where the environment state is represented via raw sensory information. In Chapter 3, we proposed a new approach that is capable of identifying subgoals in Atari games from raw pixels. Compared to the few existing competing methods, our method was more sample efficient and more selective in the subgoals it identified. We leveraged the derived subgoals via a novel intrinsic reward scheme called pellet rewards. Across three sparse reward games, our pellet rewards agent learned faster and more consistently than a cloned configuration with no pellet rewards, and also achieved significant progress in the notoriously difficult game of Montezuma’s Revenge.
R2: Given the granular time scale in videogames, what planning techniques can be developed to leverage “god’s eye vision” and achieve a lookahead depth comparable with that of humans?

In Chapter 4, we hypothesised that human players are able to formulate long-term plans in the face of time scale granularity by looking ahead in temporally extended steps, such as “jump across that gap” and “run to the end of that platform”. Further, we hypothesised that the future uncertainty arising from such steps can be handled efficiently by conflating similar outcomes into archetypal states, provided that the agent can handle small deviations from the plan via reflexes. Based on this intuition, we proposed Synoptic Vision Planning (SVP). Under this approach, the agent first learns a transferable local movement skill, and the limits of that skill, over an initial pre-training phase. Subsequently, when faced with a previously unseen, full-screen navigation task, the agent uses god’s eye vision to project archetypal states, assigning edge weights between archetypal states based on knowledge of the local movement skill’s limits. The resultant high-level planning problem is solved via Dijkstra’s algorithm. We showed in our experiments that this approach was capable of finding viable plans for full-screen, maze-like navigation tasks in real-time without exploiting an exact model of the game’s physics. Moreover, it scaled far better with distance to goal than a streamlined low-level search agent that exploited access to an exact simulator.

R3: In sparse reward videogames with continuous actions, what advantages does acting hierarchically over continually parameterised skills offer over ordinary learning?

In continuous action domains, effecting meaningful change in the environment sometimes requires a basic level of action coordination. If, in addition, the task’s reward scheme is sparse, the need to explore efficiently is amplified. A natural approach to this problem is to first teach the agent basic action coordination (in the form of a continually parameterised skill) and only afterwards train it on the task itself. However, while this type of approach is clearly necessary in extreme examples, such as a roller skating agent learning to navigate to some faraway reward, we could not find any previous research examining edge cases. As such, it was unclear if the only advantage of this approach lay in addressing reward sparsity. In Chapter 5, we explored this question by devising the Rocketship domain, an environment containing two tasks that lie in the grey area where the need for hierarchy or otherwise is non-obvious. Interestingly, we found that even when the tasks were configured so that the hierarchical and non-hierarchical agents had the same initial reward discovery rate, the hierarchical agent’s long-term performance was superior. Further analysis of the agents’ action distributions suggested that the reason for this was that the hierarchical agent explored a wider range of task-relevant actions.
6.2 Overall Limitations

For the first contribution in this thesis, namely our autonomous subgoal identification method, we sought to avoid reliance on human expert representations. Our main reason for this was to maintain comparability with other agents, as it is common practice in the deep reinforcement learning community, and in the Atari domain in particular, to require the agent to learn from raw pixels. Ultimately though, our approach does still leverage some domain knowledge. Besides the standard forms of domain knowledge that almost all deep learning agents possess, such as specialised neural network architectures, our agent exploits the task-specific intuition that making meaningful progress in many videogames requires the player to oversample actions.

A more serious limitation to our approach, however, is that the high-level knowledge captured by the exploration effort function is largely inaccessible. That is, it is distributed across many neurons and connection weights, making it difficult to extract and reason over symbolically. On the other hand, when humans play Montezuma’s Revenge, it is clear that they are able to view the environment state in terms of separable factors, such as the current room ID, the position of the protagonist within that room, and whether or not certain items have been collected. Currently, it is beyond our capability to extract factored representations from videogames without exploiting emulator tricks or leveraging domain knowledge heavily. As such, the later agents developed in this thesis cannot be considered truly autonomous, since they require factored representations. For example, in Infinite Mario, we constructed archetypal states by editing the \((x, y)\) position of Mario in the state representation. This requires Mario’s \((x, y)\) position to be a separable factor. Similarly, in Chapter 5, we taught the agent how to manipulate and reason over the rocket’s velocity, which requires the agent to see the rocket’s velocity as a separable factor. More broadly, we contend that the difficulty of extracting high-level, factored state representations is one of the main reasons why it remains difficult to integrate deep reinforcement learning with traditional planning and symbolic reasoning methods.

Finally, recall from the introduction that our motivation for studying videogames was not just to build strong videogame agents \textit{per se}, but also to gain insight into issues that arise in real-world tasks. While we stand by this motivation, a major barrier that remains to deploying our algorithms in the real world is the difficulty and cost of gaining experience. Our Atari agents, for example, were trained for 30 million frames of experience, which amounts to almost six days of non-stop gameplay. Moreover, they all suffered in-game deaths frequently, and recovering from failure might not be so easy in the real world. To apply our ideas outside of simulated environments, it will first be necessary to address the safety and sample complexity of reinforcement learning algorithms.
6.3 Ideas for Future Research

In this section, we suggest some possible avenues for extending our work.

Pellet Rewards and Exploration Effort (Chapter 3)

As noted during our discussion of existing intrinsic reward schemes (Section 3.1.2), one area where current agents for sparse reward games still struggle is in achieving high-level exploration. For example, upon reaching the first key in Montezuma’s Revenge, the player is faced with the high-level choice of which door to open: the left door or the right door. When faced with this decision, most existing agents form a strong preference for one of the doors, and rarely explore opening the other door. Currently, this weakness also applies to our pellet rewards agent. However, since our partitioning algorithm is already capable of identifying high-level subgoals, it should not be too difficult to address this weakness. For example, instead of reactivating all pellets at the start of a training run, it might be better to randomly reactivate only some of the pellets. Then, provided there are pellets close to each of the doors, the reward scheme will sometimes bias the agent to select the left door, and sometimes bias it to head right. Similarly, one might try reactivating all the pellets, but add a random multiplier to their values. Finally, one might try the more sophisticated approach of learning a meta-policy over the pellet values, with the aim of managing the agent’s high-level exploration more intelligently.

Going forward, we believe it may also be valuable to incorporate uncertainty estimates into the exploration effort function, perhaps leveraging recent work by Gal [2016] and others. Our rationale is that, ideally, the derived distance measure should account for changes to the state distribution as the agent progresses. When the agent first experiences states with large visual novelty (e.g. when it first reaches beyond the first room in Montezuma’s Revenge), the EE estimates are liable to be inaccurate. On the one hand this is not a major problem, since inaccurate EE values will usually result in large distance estimates (due to the way Equations 3.5 and 3.6 are structured). Accordingly, the agent will tend to generate new partitions – and hence spawn new pellet rewards – in such areas. And generally speaking, spawning pellets in new areas ought to be beneficial. However, to ensure the reliability of the distance estimates, it would be better if EE values with large uncertainty were excluded from the calculation of distance (Equation 3.6).

Synoptic Vision Planning (Chapter 4)

In its current form, synoptic vision planning is best-suited to classical 2D games where the game world is laid out according to a grid. Besides Infinite Mario, other examples of such games include Metroid, Alex Kidd in Miracle World and Gauntlet. However, laying a square grid over the game world is unlikely to generalise well to less structured environments. The reason for this is that, currently, the way in which the agent constructs...
archetypal states is by transporting the protagonist to the exact centre of the target tile, which assumes that variations in the protagonist’s position within the tile are unimportant. However, in less structured environments, there might be a physical barrier running through a tile. Ideally, it would be best to avoid this by taking physical barriers into account when defining the tiles. Finding a way to achieve this quickly and autonomously in new levels would mark a significant improvement to our work.

In order to establish a proof of concept for the central idea behind SVP – namely, that is it not necessary to predict all future state details accurately so long as the agent possesses strong reflexes – we simplified the planning problem in certain ways. For example, we restricted Mario to “small” mode, so that it was impossible for the agent to smash bricks. In the complete game, it is sometimes necessary to smash certain bricks, or to leave certain bricks intact, to reach a particular part of the screen. As such, it might be that a local movement succeeds in the sense of Mario reaching a local target, but fails in the sense that it renders the completion of a later step impossible, due to a critical brick having been smashed. Another key simplification that we made was considering only the probability of a local movement failing, but not the severity of failure. In reality, when Mario misses a jump, sometimes it may be very easy to recover, while in other cases it may cause a significant loss of time, or even death. Modifying the planner to consider a richer range of local movement outcomes is a promising direction for future work.

Continually Parameterised Skills (Chapter 5)

One issue that we did not address when applying continually parameterised skills to the Rocketship domain (Section 4.2) was why we trained a “fly at velocity $v$” skill, and not some other ability. The answer is merely that it seemed like an intuitive skill to train. From a purist standpoint, it would be more satisfying if we had some way of autonomously identifying the rocket’s velocity as an appropriate target variable for a parameterised skill. Traditional work on autonomous skill acquisition has focused on finding substructure within the task, e.g. identifying the fact that an agent must enter a doorway in order to access locations in other rooms. However, our work in Chapter 5 focused on exploiting substructure within the action space, e.g. the fact that driving a car in a certain direction involves setting the steering wheel at a certain angle, which in turn involves moving various joints in one’s arms, etc. Since autonomous acquisition of standard, fixed skills has been studied for decades without reaching a consensus on the best approach, there is likely much room for further work here. Lastly, we note that the hierarchical control structure proposed in Section 5.1 entails some loss of fine control. That is, when the agent is forced to act over a high-level parameter space, certain low-level actions may become unavailable. It might be worth trying an additive functional form for the policy, whereby a rough policy is obtained via skill composition, then an additional policy over the primitive action space is added to recover fine control.
Bibliography


