Automating Abstraction For Potential-Based Reward Shaping

John Burden

Doctor of Philosophy

Computer Science
University of York
December 2020

Abstract

Within the field of Reinforcement Learning (RL) the successful application of abstraction can play a huge role in decreasing the time required for agents to learn competent policies. Many examples of this speed-up have been observed throughout the literature. Reward Shaping is one such technique for utilising abstractions in this way.

This thesis focuses on how an agent can learn its own abstractions from its own experiences to be used for Potential Based Reward Shaping. As the thesis progresses, the environments for which the abstraction construction is automated grow in complexity and scope — while also utilising less external knowledge of the domains. This culminates in the approaches *Uniform Property State Abstraction* (UPSA) and *Latent Property State Abstraction* (LPSA), which can both augment existing RL algorithms and allow them to construct abstractions from their own experience and then effectively make use of these abstractions to improve convergence time. Empirical results from this thesis demonstrate that this approach can outperform existing deep RL algorithms such as Deep Q-Networks over a range of domains.

Contents

1 Introduction **13**

1.1 Hypothesis and Scope . . . 14

1.2 A Note On Abstraction . . . 15

1.3 Overview of the Thesis . . . 16

2 Background and Literature **17**

2.1 Reinforcement Learning . . . 17

2.1.1 Markov Decision Processes . . . 18

2.1.2 Value Iteration . . . 19

2.1.3 Q-Learning . . . 21

2.1.4 SARSA . . . 22

2.1.5 Off-Policy and On-Policy . . . 22

2.1.6 Limitations of Reinforcement Learning . . . 23

2.2 Eligibility Traces And Watkins' $Q(\lambda)$ . . . 24

2.3 Generalisation in Reinforcement Learning . . . 25

2.3.1 Linear Approximation . . . 26

2.3.2 Tile Coding . . . 27

2.4 Abstraction In Reinforcement Learning . . . 28

2.4.1 Semi Markov Decision Processes . . . 29

2.4.2 Options . . . 30

2.4.3 Using Options . . . 32

2.4.4 Option Generation . . . 32

2.4.4.1 Bottlenecks . . . 33

2.4.4.1.1 Clustering . . . 33

2.4.4.1.2 Ant System Optimisation . . . 34

2.4.4.1.3 Other Methods 35
2.4.4.2 Extended Sequence Trees 35
2.4.4.3 Association Rule Mining 37
2.4.4.4 Concluding Remarks For Option Generation . . 38
2.4.5 Abstract Markov Decision Processes 39
2.5 Guided Learning 40
2.5.1 Curriculum Learning 40
2.6 Reward Shaping 41
2.6.1 Reward Shaping With AMDPs 42
2.6.2 Knowledge Revision 43
2.6.3 Automatic Shaping and Decomposition of Reward Functions 44
2.6.4 Multi-Grid Reinforcement Learning 46
2.7 Unification of AMDPs and SMDPs 46
2.8 Deep Learning 47
2.8.1 Deep Q-Learning 48
2.8.2 Convolutional Neural Networks 49
2.8.3 Deep Reinforcement Learning and Abstraction 50
2.8.4 Option Heads 52
2.8.5 Hierarchical Actor-Critic 52
2.8.6 Overview 53
2.9 Concluding Remarks 54

3 Empirical Analysis of the Feasibility of Uniform State Abstraction For Reward Shaping 55
3.1 Flag Collection Domains 56
3.2 Constructing the AMDP 56
3.3 Utilising the AMDP 61
3.4 Results 62
3.5 Issues with this Approach 66
3.5.1 Domain Knowledge 66
3.5.2 Intuitive Objections 67
3.5.3 Time Taken To Solve the AMDP 68
3.6 Conclusion 68

4 Uniform State Abstraction For Reinforcement Learning 71
4.1 Uniform Partition State Abstraction 71
4.2 Constructing the AMDP 72

4.2.1 Partitioning the State-space . . . 72
4.2.2 Exploration . . . 73
4.2.3 Abstract Reward . . . 75
4.2.4 Abstract Goals . . . 76
4.2.5 Putting it all together . . . 76
4.3 Utilising The AMDP . . . 77
4.4 MultiGrid Reinforcement Learning . . . 78
4.4.1 Differences . . . 79
4.5 Alternative Approaches . . . 80
4.6 Experiments And Results . . . 80
4.6.1 Mountain Car . . . 81
4.6.2 Continuous Puddle World . . . 81
4.6.3 Catcher . . . 82
4.6.4 Experimental Detail . . . 82
4.7 Limitations and Next Steps . . . 84
4.8 Conclusion . . . 87

5 Latent State Abstraction **88**
5.1 High Dimensional State-Spaces . . . 88
5.2 Continuous Abstract States . . . 89
5.3 Latent State-Spaces . . . 89
5.4 Auto-Encoders . . . 90
5.4.1 Variational Auto-encoders . . . 92
5.5 Using Auto-encoders and Latent State-spaces with Reinforcement Learning . . . 93
5.5.1 World Models . . . 93
5.5.2 Robotics . . . 94
5.5.3 Structured Latent State-spaces . . . 95
5.6 Latent State-spaces as Abstraction . . . 96
5.7 Utilising Latent State-spaces . . . 97
5.8 Method Overview . . . 98
5.9 Experimental Domain . . . 99
5.10 Training the Auto-encoder . . . 101
5.10.1 Auto-encoder Reconstruction . . . 105
5.11 Training the Abstract Network . . . 105
5.11.1 Differences to Uniform Partition State Abstraction . . . 109
5.11.2 Architecture and Hyper-parameters . . . 110

5.12 Utilising the Abstraction 110
5.12.1 Training the Ground Network 111
5.13 Results 112
5.13.1 LPSA Results 113
5.13.2 Caveats 116
5.13.3 Comparing Abstraction Functions 117
5.14 Conclusion 120

6 Conclusion And Future Work 121
6.1 Summary Of Contributions 121
6.1.1 Comparison of uniform partitions and hand-labelled examples. 121
6.1.2 Learning abstract transition functions 122
6.1.3 Latent property state abstraction 123
6.2 Limitations 124
6.3 Generality 125
6.4 Future Work 126
6.4.1 Abstraction Integration 126
6.4.2 Abstraction Height 126
6.4.3 Abstraction Width 127
6.5 Concluding Remarks 128

List of Tables

3.1 Descriptions of the variations on the Flag Collection domain that were selected. 57

4.1 Hyper-parameters used for experiments 84

5.1 Comparison of properties of MultiGrid-RL, Uniform Partition State Abstraction and Latent Property State Abstraction 110

5.2 Hyper-parameters used for experiments 110

5.3 Average number of steps per second performed by each agent over the 25 training runs (rounded to two decimal places). 116

List of Figures

2.1 Convolutional Layer Process 50

3.1 Graphic representation of each variation on the Flag Collecting domain. The agent begins at the red cell marked ‘S’ and must traverse to the green cell marked ‘G’, whilst moving through yellow flag cells marked ‘F’. 58

3.2 Visualisation of the partitioning of the states-pace into abstract states of size 7×7 for the basic Flag Collection domain 59

3.3 Reward graphs for each Flag Collection variation using different sized abstractions. Results shown on the left side are smoothed with a moving average of reward of the previous $NumberOfEpisodes/10$ episodes, with shaded regions denoting a confidence interval of 95%. On the right are the unsmoothed results. 63

3.3 Reward graphs for each Flag Collection variation using different sized abstractions. Results shown on the left side are smoothed with a moving average of reward of the previous $NumberOfEpisodes/10$ episodes, with shaded regions denoting a confidence interval of 95%. On the right are the unsmoothed results. 64

3.4 Time taken to solve each abstraction and simulate 10,000 episodes for the basic Flag Collection domain 65

3.5 Possible scenario in which the reward shaping may slow down the learning process due to the AMDP’s granularity. 69

4.1 Visualisation of mapping from ground-transitions to abstract-transitions. 74

4.2 The environments used to evaluate our method. 81

4.3 Neural network architecture for each of the agents utilised in this chapter. The architecture is shared across environments also, changing only the number of input dimensions and output actions. 83

4.4 Results Comparing DQN, MRL and Uniform State Abstraction for each environment The end of the exploration phase is denoted with a dotted vertical line. We show both the raw mean results as well as a smoothed figure for visual clarity. In the smoothed figure a moving mean window is applied with size 25 in order to smooth the curves. We plot the unsmoothed results for each episode before this point, explaining the oscillation observed in the figure. 85

4.4 Results Comparing DQN, MRL and Uniform State Abstraction for each environment The end of the exploration phase is denoted with a dotted vertical line. We show both the raw mean results as well as a smoothed figure for visual clarity. In the smoothed figure a moving mean window is applied with size 25 in order to smooth the curves. We plot the unsmoothed results for each episode before this point, explaining the oscillation observed in the figure. 86

5.1 Neural network architecture for an auto-encoder 91

5.2 A broad overview of the Latent Property State Abstract method. Arrows denote the flow of informational dependencies. Stages coloured blue correspond to the auto-encoder, red to the abstraction process, orange to the ground agent and green to the environment. 100

5.3 Visualisation OpenAI Gym's Car Racing domain, image captured from the domain running with visualisation on. This is what the agent "perceives" as an array of RGB pixel values. 101

5.4 The encoder-decoder components of the auto-encoder architecture. 103

5.5 A typical plot of loss against the number of training steps for our Variational Auto-encoder when trained on frames from the Car Racing environment. 104

5.6 A sample of environment frames and their reconstructions after 50 and 500 training epochs 106

5.7 A sample of environment frames and their reconstructions after 5000 training steps 107

5.8 Abstract Agent Neural Network Architecture 111

5.9 Ground Agent Neural Network Architecture 112

5.10 The average reward received by an agent plotted against the number of elapsed episodes. Each agent was trained for approximately 24 hours. The averages are based on 25 repetitions of the learning process for each agent and the shaded regions represent a 95% confidence interval. The smoothed agent used a moving average of window size 20 for World Models and 200 for the other agents (Due to the difference in the number of episodes). 114

5.11 Simple domain demonstrating that two states, X and Y, sharing an abstract state can be further away from each other than a state Z in a separate abstract state. 118

Acknowledgements

I would like to thank the many supervisors that I've had throughout this project. Daniel Kudenko, your advice, ideas, and continued support long after you have been my official supervisor has been immensely helpful throughout the PhD programme. I look forward to hopefully continuing to collaborate in future. James Cussens and Victoria Hodge, the feedback you both have provided on the thesis has been invaluable and has helped to shape it into something that I can be proud of. I want to give you both additional thanks for taking on the supervision role for this project despite it not being in your primary areas of interest. Rob Alexander, your help for the final push of completing this project has been very helpful, and again additional thanks for taking on the supervision of this project despite it being outside of your main research interests.

I would also like to thank Tom for being the best friend I could ask for. Your moral support throughout the PhD has been crucial. My mother, Wendy, you have been vital for the completion of this thesis and PhD. Thank you for always believing in me. You were always there for me during trying times and I really could not have done this without your support.

Finally, Katerina, thank you for all of the encouragement throughout the programme, as well as for always cheering me on. You have given me the confidence to work through all of the challenges this thesis has posed. Your love has inspired and motivated me beyond description.

Declaration

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

The contents of Chapter three appeared in:

- John Burden and Daniel Kudenko. Using Uniform State Abstractions For Reward Shaping With Reinforcement Learning. In Workshop on Adaptive Learning Agents (ALA) at the Federated AI Meeting. 2018.

While the contents of Chapter four appeared in:

- John Burden and Daniel Kudenko. Uniform State Abstraction For Reinforcement Learning. In proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020).

Work based on Chapter five appeared in:

- John Burden, Sajjad Kamali Siahroudi, and Daniel Kudenko. Latent Property State Abstraction For Reinforcement learning. In Workshop on Adaptive Learning Agents (ALA) at 20th International Conference on Autonomous Agents and Multiagent Systems (Virtual). 2021

Additionally, Sajjad Kamali Siahroudi was also responsible for adapting an implementation of the *World Models* algorithm (based on [30]) which the LPSA algorithm (the main contribution of Chapter five) was compared against in Chapter five.

Chapter 1

Introduction

Reinforcement Learning (RL) is a sub-discipline of Machine Learning. It differs from both supervised and unsupervised ML. Instead, RL is an agent based approach to ML. In RL agents explore an environment and observe how it reacts. After each action performed by the agent it receives a numerical reward. This reward is used to alter the agent's behaviour in order to increase the future expected reward received. This makes RL an interaction driven ML paradigm.

Unfortunately RL typically suffers from the so-called "Curse of Dimensionality" [6]. Due to the number of states in an environment growing exponentially with the number of state dimensions, many naive RL methods have difficulty converging on decent behaviour because there isn't sufficient time to visit each state a sufficient number of times. This can make RL ill-suited for complex tasks unless preventative measures are taken.

Many methods have been created to deal with this issue. Such methods often revolve around either trying to give the agent the capability to generalise over states in order to share the effect of learning over similar states, or the methods attempt to use external knowledge to guide the agent. Reward shaping is an example of the latter, where the agent is given an additional reward based on expert knowledge in order to guide the agent to more fruitious behaviour. The knowledge to construct the additional reward often has to come from an external source, usually a human expert. This is clearly not ideal due to the costs of employing human experts to encode their domain knowledge into a suitable form for the agent. Further, some tasks may be too difficult for even human experts to solve adequately.

The work here focuses on automatically generating this knowledge through the use of abstractions. If knowledge of sufficient quality can be generated in this way, it may remove the need for human experts to encode knowledge. This could also prevent human error from entering the additional reward which may hinder learning. Methods which remove human expertise from the learning process are often highly sought after — humans are expensive and prone to error. Whilst the idea of automatically constructing domain knowledge is not new, our approaches are novel and original, and give performance boosts to the learning processes that are not specific to any one RL algorithm.

1.1 Hypothesis and Scope

The hypothesis we try to demonstrate over the course of this thesis is thus:

> "Abstractions of problem domains can be automatically learned by the agent from its own experiences utilising little knowledge from a domain expert. These abstractions can then be utilised to reduce the time required for an agent to converge upon a satisfactory solution to the original task."

Throughout the thesis we primarily focus on abstractions over state-spaces rather than other aspects of the environment. This is justified by the state-spaces often growing exponentially with the number of state-dimensions, making sufficient exploration infeasible. Other environmental factors do not tend to grow at such explosive rates, and thus we reason that applying abstractions over states-spaces will likely give the most utility. For completion, other abstraction paradigms are explored in Chapter 2: Background and Literature Review. We also limit ourselves to the single-agent case to allow more focus on the abstraction concepts without needing to fret over inter-agent interactions. The abstractions created are used to create Potential-Based Reward Shaping (PBRS) functions. There are other methods for delivering advice and knowledge to agents but PBRS provides attractive theoretical guarantees for policy invariance as well as strong empirical performance over a wide range of domains. Other approaches to imparting such advice or knowledge to agents are again detailed in Chapter 2. In brief, the contributions of this thesis are three-fold. First is a detailed comparison between the capabilities, when used for potential-based reward shaping, of hand-labelled abstractions and abstractions generated using uniform partitioning of the state-space. This comparison occurs in a simple gridworld reinforcement learning setting. The second contribution is an extension of a pre-existing reinforcement learning algorithm for automatically abstracting a reinforcement learning state-space. The extension improves performance of the pre-existing method in the deep learning scenario. The final primary contribution is a new approach, Latent Property State Abstraction, for automatically creating state-space abstractions learned by the reinforcement learning agent itself. The potential function induced by this abstraction can improve the learning rate when compared to standard benchmarks. Each of these contributions push towards demonstrating the hypothesis directly.

1.2 A Note On Abstraction

Throughout this thesis, we grapple with the notion of abstraction. As with many words in the English lexicon, abstraction can have many varied meanings. Before we begin using the term in a technical sense it is worth discussing exactly what it is we mean by “abstraction”. In modern English, the adjective “abstract” typically refers to entities “existing as an idea, feeling, or quality, not as a material object” [1] There is a verb “abstract” with the meaning “to draw away”. Both of these words have the same Latin root *abstrahere*, again meaning “to draw away”. This verb is actually key to understanding what we really mean by “abstraction” — corporeal objects have their principal properties “drawn away”, yielding an ephemeral Platonic ideal of these properties. We can think of this view of abstraction as an “is a” function. Socrates “is a” man. Aristotle “is a” man. Fido “is a” dog. There is some property of “man”-ness that both Socrates and Aristotle share, but not Fido. We can encapsulate the previous relations with a function *Species*: $Species(Socrates) = man$, $Species(Fido) = Dog$.

Equally, it is also the case that Socrates, Aristotle and Fido are (“is a”) animals. A function $Kingdom$ can again encapsulate this, where this time, all three entities share the same property of “animal”-ness. Other functions could distinguish Socrates as separate from Fido and Aristotle, perhaps $Hair\text{-}Colour$, or $Favourite\text{-}Food$. There are no “wrong” abstractions. Certainly some are more useful than others for certain purposes, but none are “wrong”.

This emphasises the “drawing away” of the etymological root. The process of abstraction — the function that maps “concrete” objects to their abstract counterparts — defines the properties that the abstract states embody.

We therefore, from a pragmatic perspective, want to find the abstraction process that gives the most useful abstract entities — embodying the most useful properties — for the task at hand. The purpose of this section on abstraction has been to highlight that the process of abstraction itself determines the properties the abstract entities will have and that this will in turn determine their utility. *Abstract* as a verb is more relevant than *abstract* as an adjective for the purposes of finding or constructing useful relationships between entities. This thesis focuses on trying to find abstractions over state-spaces for use within Reinforcement Learning that will decrease the time required for learning.

1.3 Overview of the Thesis

We begin the thesis proper in the next chapter, Chapter 2, with a comprehensive overview of the required background knowledge of Reinforcement Learning and relevant related fields.

Chapter 3 then explores the feasibility of creating reward shaping functions when they are based on abstract states that are formed from uniform partitions of the environment's state space. This chapter will show that uniformly partitioned state-spaces can yield reward shaping functions that are able to compete with shaping functions derived from hand-labelled abstract state-spaces.

In Chapter 4 we expand on the method from Chapter 3, removing more domain knowledge given to the agent. The agent instead learns to discover this knowledge for itself through environmental interaction. Here we see vast improvements over standard RL algorithms in popular benchmarking domains.

More complex tasks are introduced in Chapter 5, where we scale the central idea of the previous method to much larger domains. This involves the agent itself learning new ways to represent abstractions of states. So-called "latent state-spaces" are utilised to create a smaller abstract counterpart to the desired environment to train an agent. The agent can train in this abstract environment to produce a reward shaping function. Afterwards the agent can use the shaping function to assist its learning in the original environment.

Chapter 6 concludes the thesis, bringing together a detailed summary of contributions. Both the achievements and limitations of the work performed in the thesis are also discussed, in addition to how these limits could be overcome with future work.

Chapter 2

Background and Literature

This chapter introduces many of the topics and concepts which are required to understand the thesis. The fundamentals to Reinforcement Learning and Markov Decision Processes are given in Section 2.1. Extensions to Reinforcement Learning in the form of abstraction are introduced in Section 2.4. Reward Shaping, a key mechanism for delivering knowledge to RL agents, is introduced in Section 2.6. Deep Learning, a powerful technique when paired with RL , is introduced in Section 2.8 along with alterations to the basic concept, including the utilisation of abstractions.

This chapter also comprises an extensive literature review of the use of abstraction within Reinforcement Learning, highlighting many of the researched methods for creating such abstractions and delivering the insight they provide to the agent for improved learning.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is an interaction driven machine learning paradigm [65]. RL attempts to mimic learning processes that occur in nature, with agents altering their behaviour in order to avoid repeating mistakes. It is in this manner that RL differs from most other branches of machine learning. In RL there are no training examples of correct behaviour. Instead, an agent interacts with its environment and receives — typically numerical — feedback. This feedback is then used by the agent in order to alter its behaviour to be more fruitful.

From a practical perspective, most of the agents and environments used in RL are simulated. The distinction between real and simulated environments is important to make, there are important differences between the two. Whilst it is possible to apply RL techniques to the real world, it is impractical and time consuming to do so. Simulated environments and agents operate many times faster than "real-time". In addition, there is no cost of building physical apparatus when working in simulated environments. On the other hand, simulated environments require construction by the user. Doing so requires some knowledge of the domain, chiefly the dynamics of the environment, as well as how to provide the feedback. This potentially limits the applications of RL to domains in which these factors are known. Like other machine learning methods, RL suffers from the "Curse of Dimensionality" [6], which also limits its practical use, unless steps are taken to overcome this. This "curse" will be explained in detail in Section 2.1.6.

Despite these setbacks, Reinforcement Learning has enjoyed something of a renaissance in recent years. This is principally due to methods being composed to overcome the aforementioned limitations. These include Deep Learning and other methods that fully utilise Reward Shaping and generalisation frameworks. Combined with attention from large corporations and research groups, this has led to large scale deployment of RL in impressive projects. Among these is the agent that learnt to play the popular game "Go" [62] and defeated the world champion — a first for the game. Agents have also been trained to outperform humans in a large number of games for the Atari 2600 using Deep Q-Learning [45].

These recent events show that RL is a promising field for future study. But — to use the famous aphorism — in order to "stand on the shoulders of giants", one must first climb their back. The rest of this chapter is to be exactly that — a review of the previous work done within RL, from its humble beginnings to the most recent work on abstraction representation and inducement.

2.1.1 Markov Decision Processes

Reinforcement Learning is often modelled as interaction with a Markov Decision Process (MDP). An MDP is a tuple (S, A, R, P, γ). Here S is a set of states. A is a set of actions for each state. $R(s, a, s')$ is the immediate reward received by the agent when moving from state s to s' using action a. $P(s, a, s') = \mathcal{P}(s' \mid s, a)$, that is, the probability of transitioning to state s' when in state s and performing action a. Finally, γ is the discounting factor, this is used to weigh immediate reward against expected future reward.

MDPs have what is referred to as the "Markov property". The Markov property states that the future distribution of states depends only on the current state (and action selected) and not the sequence states prior. For an MDP to fully capture the dynamics of an environment, the environment should also have this property.

An agent explores (or interacts with) the MDP beginning in some initial state, and ending in some final state. A single run of the MDP is referred to as an episode. The agent has a function π called a policy that maps states in the MDP to an action for the agent to perform. The goal for the agent is to maximise the cumulative reward received over a single episode. The policy that does this is referred to as optimal.

Iterative approaches are often used to solve MDPs. These utilise what are known as "Temporal Difference" (TD) updates. These update values of states $V(s)$ or state-action pairs $Q(s, a)$, based on estimates of these value functions at different times.

Throughout the thesis, we will often refer to agents beginning in state s, performing action a and moving into state s'. We will often refer to this as a transition $s \xrightarrow{a} s'$.

When an agent is selecting actions, it is important to balance both exploration and exploitation [65]. This helps prevent the agent from becoming trapped in a locally optimum policy, while also allowing the agent to utilise the discoveries it has made. An example of a policy that does this is the ϵ-greedy policy. Under this policy, the agent selects the action it considers best with probability $1 - \epsilon$. With probability ϵ the agent selects an action at random. This guarantees that eventually every state-action pair will have been visited an infinite number of times, thus the agent will converge upon the optimal policy [65].

2.1.2 Value Iteration

Value Iteration [6] is a dynamic programming approach to solving MDPs — that is, finding a value function V, where $V(s)$ denotes the maximum expected return from state s. From this, Value Iteration also derives a policy selecting actions to achieve this expected return.

The core of Value Iteration is that the value of a state can be computed based on the values of its neighbours and the rewards and transition probabilities of moving to them. For an optimal value function V^*, the following equation holds:

$$V^*(s) = \max_a \sum_{s'} P(s, a, s')(R(s, a, s') + \gamma V^*(s'))$$

Intuitively we can read this equation as saying that the value of a state s is equal to the sum of the immediate reward received and the discounted value of its neighbours when the best action is taken and is appropriately weighted by the MDP's transition probabilities.

Iterative approximations of V^* can be made by applying the above equation to previous approximations:

$$V_{t+1}(s) := \max_a \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma V_t(s'))$$

Repeatedly applying this equation to all states will converge on a fixed solution [54]. An accuracy threshold δ^* is used to bound the maximum distance between successive iterations before convergence — the algorithm is guaranteed to converge in the infinite limit, so approximations are needed practically.

The complete Value Iteration algorithm is given in Algorithm 1.

Algorithm 1 Value Iteration

```
procedure VALUE ITERATION(MDP M: $(S, A, R, P)$, discount factor $\gamma$, accuracy threshold $\delta^*$ )
    Set $V_0(s) = 0$ for all $s$
    $\delta = \infty$
    $t = 1$
    while $\delta^* < \delta$ do
        $\delta = 0$
        for $s \in S$ do
            $V_t(s) \leftarrow \max_a \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma V_{t-1}(s')$
            $\delta \leftarrow \delta + \mid V_t(s) - V_{t-1}(s) \mid$
            $t \leftarrow t + 1$
    Initialise empty policy $\pi$
    for $s \in S$ do
        $\pi(s) \leftarrow \text{argmax}_a \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma V_t(s'))$
    Return $\pi$
```

Value Iteration has a few flaws that make it unsuitable for practical use in most cases. The first major obstacle is that Value Iteration requires the transition function P and reward function R to be known. For many environments, particularly complex ones, these functions are often not known, They can often be approximated through interaction with the environment, but this is costly. The second major obstacle to practical use of Value Iteration is the computation time required to converge on a sufficiently accurate solution. A single sweep through of the states to update all $V_t(s)$ requires $O(|S|^2|A|)$ steps. Further, due to the Curse of Dimensionality [6], for high dimensional problems there are an exponential number of states to begin with. This causes a single sweep through of states for Value Iteration to be of order $O(n^{2d}|A|)$ for the number of states per dimension n and number of dimensions d. For large-scale, practical problems this is simply intractable.

Due to needing the transition and reward functions, Value Iteration is referred to as a model-based approach to solving MDPs. This is in contrast to “model-free” techniques. There are many RL algorithms that fall into both categories.

The ones we focus on in the coming sections (and for the majority of the thesis) are model-free because they more accurately represent the case where an agent is interacting with a new, unknown environment. Additionally, model-free algorithms are more likely to benefit from the inclusion of abstraction and the imparting of domain knowledge.

2.1.3 Q-Learning

Q-Learning is a basic example of a temporal difference RL algorithm [65]. In Q-Learning the agent records estimates of the expected return for each state-action pair. This is recorded by function Q, often in the form of a look-up table. $Q(s, a)$ is then the estimated expected return for the agent when in state s, performing action a and thereafter following a greedy policy.

The crux of the Q-Learning algorithm is that after the agent makes transition $s \xrightarrow{a} s'$, the estimates are updated:

$$Q(s,a) := Q(s,a) + \alpha\Big(r + \gamma \max_{a*} Q(s', a^*) - Q(s,a)\Big)$$

The intuition behind this update rule is that if the agent can transition between state-action pairs (s, a) and (s', a'), then the value estimates of these pairs should be closer, as they are not that far apart in the agent's state-action space of the MDP.

The full Q-Learning algorithm is presented in Algorithm 2.

Algorithm 2 Q-Learning

```
procedure Q-LEARNING(MDP M: $(S, A, R, P)$, policy $\pi$, discount factor $\gamma$ )
    Set $Q(s, a) = 0$ for all $(s, a)$
    for Each Episode do
        $s \leftarrow$ Initial State
        while $s$ is not terminal do
            action $\leftarrow \pi(s)$
            Perform action $a$, yield $r, s'$
            $Q(s,a) := Q(s,a) + \alpha\Big(r + \gamma \max_{a*} Q(s', a^*) - Q(s,a)\Big)$
            $s \leftarrow s'$
```

Q-Learning is often used with ϵ-greedy policies to ensure convergence guarantees [65]. In the case of Q-Learning, an ϵ-greedy algorithm would correspond to selecting the action $\operatorname{argmax}_a Q(s, a)$ with probability $1 - \epsilon$ and a random action with probability ϵ. Other policy types also have convergence guarantees. An example of another policy of this sort is fully-random action choice with a decaying learning rate.

The most important thing to note about Q-Learning is that the agent never needs to see the reward function or transition function of the underlying MDP. Consequently, this means that Q-Learning can be utilised when such functions are unknown, allowing learning in environments that do not have fully known environment models.

2.1.4 SARSA

Another basic RL algorithm that is worth introducing is SARSA. SARSA is an acronym for State-action-reward-state-action derived from the necessary components for the update rule. Originally introduced as "Modified Connectionist Q-Learning" [60], the catchier SARSA name has become far more common.

Similarly to Q-Learning, SARSA uses a function Q, where $Q(s, a)$ is again the expected return for the agent when in state s, performing action a and thereafter following the policy that selected a.

The chief difference between SARSA and Q-Learning comes from the update rule. SARSA updates its Q-Function according to:

$$Q(s, a) := Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$$

Here, a' comes from the same policy that selected a. For completeness, the full SARSA algorithm is given in Algorithm 3.

Algorithm 3 SARSA

```
procedure SARSA(MDP M: $(S, A, R, P)$, policy $\pi$, discount factor $\gamma$ )
    Set $Q(s, a) = 0$ for all $(s, a)$
    for Each Episode do
        $s \leftarrow$ Initial State
        action $a \leftarrow \pi(s)$ Initial Action
        while $s$ is not terminal do
            Perform action $a$, yield $r, s'$
            action $a' \leftarrow \pi(s')$
            $Q(s, a) := Q(s, a) + \alpha\Big(r + \gamma Q(s', a') - Q(s, a)\Big)$
            $s \leftarrow s'; a \leftarrow a'$
```

2.1.5 Off-Policy and On-Policy

Ostensibly the differences between SARSA and Q-Learning are minor; though this is actually not the case — the differences between these two algorithms form a dichotomy within RL algorithms.

These two types of algorithm are referred to as on-policy and off-policy. To see the exact difference we need to directly compare the update rules for Q-Learning and SARSA:

$$Q(s,a) := Q(s,a) + \alpha\Big(r + \gamma \max_{a*} Q(s', a^*) - Q(s,a)\Big) \tag{2.1}$$

$$Q(s,a) := Q(s,a) + \alpha(r + \gamma Q(s', a') - Q(s,a)) \tag{2.2}$$

With equation 2.1 being Q-Learning and 2.2 SARSA.

These both follow the same pattern of incrementally moving the current $Q(s,a)$ value towards the immediate reward plus the discounted expected future value. However, Q-Learning's expected future value assumes that a greedy policy will be followed after the action a — this is given by the $\max_{a*} Q(s', a^*)$. On the other hand, SARSA assumes that the same policy will be used throughout.

This entails that SARSA will yield Q-values detailing expected returns for the training policy, whereas Q-Learning's Q-values reflect the expected returns for a greedy policy. We say that SARSA is an “on-policy” algorithm and that Q-learning is “off-policy”.

Off-policy algorithms have the advantage that non-optimal policies can still learn optimal Q-values [65], whereas this does not hold for on-policy algorithms. For training, Q-learning with a random policy would (eventually) lead to Q-values that, if followed greedily, would still perform optimally. The same does not hold for SARSA. However Off-policy methods can also contribute to algorithmic instability, especially when combined with function approximation and bootstrapping. The three of these together are referred to as the “deadly triad” [66] due to their volatile combination.

Both SARSA and Q-Learning are examples of so-called “tabular” approaches. By this it is meant that they explicitly store values in a Q-table against exact states. States which are not covered in the Q-table have no defined value associated with them. Tabular approaches are unable to generalise or interpolate values to unseen states.

2.1.6 Limitations of Reinforcement Learning

Due to the aforementioned “Curse of Dimensionality” the amount of states in an environment often grows exponentially with the size of its state representation (i.e., the number of dimensions used to represent a state). The Q-Learning algorithm typically requires that each state-action pair is visited at least once (though more than this) in order for the Q-function estimates to approach sensible values. This is problematic as for larger environments Q-Learning will take too long to converge on a suitable policy.

To address this issue, there are two primary options. The first of these is to give the agent the ability to generalise. This will aid the agent by allowing it to recognise that some state-action pairs are similar to others, and to make updates pertaining to this generalisation instead of just individual states and actions. Exactly how the generalisation is performed varies, but will typically be some form of gradient descent method that perhaps utilises Neural Network architectures.

The second option to address the limitations of RL is to use external knowledge in order to "guide" the agent towards more rewarding behaviour. Doing so prevents the agent from wasting a lot of time on policies that are very sub-optimal. There are a few methods in which this guidance is given. One approach is Imitation Learning, where domain experts "perform" desirable behaviour and the agent learns to mimic and improve upon this demonstration (See [32] for a survey of Imitation Learning methods). Another typical method in which guidance is given is called Reward Shaping and is discussed in more detail in Section 2.6. The downside to such methods is that very often, some knowledge about the domain is required. This is not always available or may be expensive to encode for the agent. On the other hand, the speed up in learning when using reward shaping can be a large factor, and not that much knowledge about the environment is typically required.

Whichever option is selected, the resulting RL system can become very powerful at effectively learning to perform at near-optimal levels for very large environments.

2.2 Eligibility Traces And Watkins' $Q(\lambda)$

Eligibility traces (first proposed in the field of Neurophysiology by Klopf[37] before being introduced to the field of RL [68][63]) provide a simple yet effective method for updating multiple Q-values at once. The central idea to eligibility traces is to propagate Q-value updates backwards through the agent's trajectory, "crediting" recent states more. In order to achieve this, a list of "eligibilities" is kept. $E(s, a)$ represents the eligibility of state-action pair (s, a). The idea here being that a higher eligibility value will result in a larger Q-value update and that more recent states will have a larger eligibility that will decay over time until the state is revisited.

We describe the most common variation of how eligibility traces are implemented. This is referred to as an "Accumulating" eligibility trace. Initially, $E(s, a) := 0$ for all state-action pairs. When the agent enters state s and performs action a, $E(s,a)$ is updated to the value $E(s,a)+1$. After each action, $E(s,a)$ is updated for each state-action pair to $E(s,a) := \gamma\lambda E(s,a)$ for discounting factor γ and hyper-parameter $0 \leqslant \lambda \leqslant 1$.

Watkins' $Q(\lambda)$ [74] is an algorithm that takes advantage of eligibility traces in order to update multiple state-action pairs after each transition. This algorithm is essentially Q-learning with the eligibility traces added.

Algorithm 4 Watkins' $Q(\lambda)$

```
procedure WATKINS' Q(MDP M: $(S, A, R, P)$, policy $\pi$, discount factor $\gamma$ )
    Set $Q(s,a) = 0$ for all $(s,a)$
    Set $E(s,a) = 0$ for all $(s,a)$
    for Each Episode do
        $s \leftarrow$ Initial State
        $a \leftarrow \pi(s)$
        while $s$ is not terminal do
            Perform action $a$, yield $r, s'$
            Select next action $a' \leftarrow \pi(s')$
            Select optimal action $a* \leftarrow \text{argmax}_\alpha(Q(s', \alpha)$
            $\delta \leftarrow r + \gamma Q(s', a') - Q(s,a)$
            $E(s,a) \leftarrow E(s,a) + 1$
            for $(s,a) \in S \times A$ do
                $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
                if $a' = a*$ then
                    $E(s,a) \leftarrow \gamma\delta E(s,a)$
                else
                    $E(s,a) \leftarrow 0$
            $s \leftarrow s'; a \leftarrow a'$
```

Watkins' $Q(\lambda)$ takes the additional step of reducing $E(s,a)$ to 0 if the action selected was not greedy — it makes no sense to punish prior state-action pairs in the trajectory for unfortunate exploration steps. The λ parameter has the effect of determining the extent to credit previous transitions in the agent's trajectory. It is worth noting that $Q(0)$ – that is, $\lambda \leftarrow 0$ — is equivalent to standard Q-learning.

2.3 Generalisation in Reinforcement Learning

One of the approaches to overcoming the Curse of Dimensionality is the use of generalisation. By generalisation we mean the agent makes decisions and updates based on not only the exact state it is considering, but also states that are in some way similar. This allows the agent to draw on a wider selection of experiences when making decisions.

A common way of giving this generalisation ability is through the use of function approximation. When state-spaces are very large or continuous it is infeasible to store them and their associated values tabularly. The solution to this is to construct an approximation function that either reduces the state space size or discretises continuous spaces. We will now look at a few examples of function approximation methods and how they are used within RL.

2.3.1 Linear Approximation

A simple function approximator is that of Linear Approximation. Despite its relative simplicity, it is easy to understand and gives valuable insight into function approximation in RL. For linear function approximation we require a feature extraction function, ϕ, mapping state-action pairs to a vector of features, $\phi : S \rightarrow A \rightarrow \mathcal{R}^n$.

The value of a state-action pair (s, a) is then simply the dot-product of a weight vector θ and $\phi(s, a)$. $Q(s, a) = \theta \cdot \phi(s, a)$. This makes the value of a state dependent only on features of (s, a) and the learned weights. Since the features of state-action pair (s, a) are static over the course of learning, the problem reduces to learning suitable values for θ.

Gradient Descent methods can be used to learn these values for θ. Let's recall a general gradient descent update rule:

$$\theta_{t+1} := \theta_t - \alpha \nabla F(\theta_t)$$

Where F is defined as a suitable cost function. Let us recall the tabular Q-Learning update rule:

$$Q(s, a) := Q(s, a) + \alpha(r + \gamma \max_a (Q(s', a) - Q(s, a))$$

If we consider $\hat{y} = r + \gamma \max_a Q(s', a)$ to be our "ground truth" value, and $y = Q(s, a)$ our predicted value, then taking

$$F(\theta) = \frac{1}{2}(y - \hat{y})^2$$

as our cost function for gradient descent yields update rule

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_t} \Big[\frac{1}{2}(r + \gamma \max_a Q(s', a) - Q(s, a))^2 \Big] \tag{2.3}$$

$$\theta_{t+1} = \theta_t - \alpha \Big[(r + \gamma \max_a Q(s', a) - Q(s, a)) \nabla_{\theta_t} (r + \gamma \max_a Q(s', a) - Q(s, a)) \Big] \tag{2.4}$$

However, calculus is hard, and we are only humble Computer Scientists, so when calculating the gradient $\nabla_{\theta_t}(r + \gamma \max_a Q(s', a) - Q(s, a))$, we elect to calculate only $\nabla_{\theta_t}(-Q(s, a))$. More seriously, this is done because $r + \gamma \max_a Q(s', a)$ is actually only an estimate of the expected return and not actually a ground truth value. This value depends on θ and is therefore not an unbiased estimator [66]. The simplification of the gradient calculation is used for this reason and works well.

As $Q(s, a) = \theta \cdot \phi(s, a)$:

$$\theta_{t+1} = \theta_t - \alpha\Big[(r + \gamma \max_a Q(s', a) - Q(s, a))\nabla_{\theta_t}(-\theta\phi(s, a))\Big] \tag{2.5}$$

$$\theta_{t+1} = \theta_t + \alpha(r + \gamma \max_a Q(s', a) - Q(s, a))\phi(s, a) \tag{2.6}$$

This is very similar to the original Q-update rule, except weighted by the feature vectors appropriately. Methods of this type are referred to as *Semi-Gradient Descent* methods [66] — since part of the gradient was ignored when deriving the weight update rule.

2.3.2 Tile Coding

Tile Coding [64] is an example of a linear function approximation method for RL. Tile Coding allows the representation of approximate value functions and provides local generalisation to the agent. Under Tile Coding, an environment's state-space is partitioned by multiple overlapping offset grids — although what we refer to as "grids" do not have to be strictly square or uniform, they just need to fully partition the environment.

Since these are partitions in the strict sense of the term, it follows that any state in the environment appears in exactly one "tile" in each partition. A feature vector ϕ can be constructed to denote which tiles the state is in for each of the partitions. This feature vector contains one dimension for each partition, with element i consisting of which tile — enumerated appropriately — is occupied by the state in feature i. To utilise this more abstract feature representation of the state-space, RL algorithms are applied to state spaces, learning values θ for $Q(s, a) = \theta \cdot \phi(s, a)$. This is a linear function with respect to the feature vector function ϕ.

Action selection is then based upon this Q-value evaluated for each available action. Weight updates are performed the same as as we saw in the preceding section on Linear Approximation methods, applying Q-update rule.

One big advantage of this method is a form of generalisation. Due to the offset nature of these grids and split weight updates, learning information about tile i in feature j generalises to even when all of the other tiles are not the same as when the agent experienced i.

A disadvantage to this method is that this generalisation is only local. Local, nearby states may always not be similar in value, whereas far off states encapsulating the same overall situation may in fact be more similar. Examples of this difference in value of nearby states can easily occur in fairly complex domains such as Chess, where a single piece being in a subtly different position can have huge ramifications for determining a player's advantage (or the board's value). Another disadvantage is that Tile Coding loses the ability to discern between states if they are sufficiently close. This may become an issue when very fine precision is needed.

2.4 Abstraction In Reinforcement Learning

Abstraction is a key element of human learning and thought; it is the process which allows humanity to relate ideas and consider events and implications far removed from their current situation or scale. Abstractions can be viewed as a similarity function, mapping similar "concrete" objects to the same abstract object. Different abstraction functions can be created to define what "similar" means, depending on what the creator wishes to abstract. Examples of abstract qualities that could be selected are colour, size, species, time, and so on. Chains of abstraction can be layered onto each other, weaving together vast hierarchical structures which give objects order by their properties.

These abstractions and structures permeate through the world that we know. We use it in learning — we apply solutions to multiple problems of the same type. Abstraction is also an incredible tool for categorisation, allowing us to group together objects of similar purpose. We can even view some aspects of science as aiming to construct abstractions about the world around us from many observations.

Due to the advantages abstraction can give us in terms of learning, it is also useful for improving the field of Reinforcement Learning. Doing so will often allow agents to make value updates for many states (those that fulfil some notion of similarity) from a single experience. Equally, abstraction can be used to drastically cut down the number of states that need considering at each action-selection step.

Abstractions in RL transcend space and time — that is, we can use both temporal and spatial abstractions to consider the effect of groups of long chains of actions trying to achieve the same thing, as well as abstractions over state-spaces, creating abstract states composed of multiple constituent states which all share some property. Now, in the coming sections, we will overview some of the prominent methods of abstraction in the RL literature.

2.4.1 Semi Markov Decision Processes

The aforementioned limitations of naïve RL severely hinder its utility to modern, realistic applications. Considering each state-action pair as individual, orthogonal entities is one of the main reasons behind this limitation. Generalising chains of primitive actions effectively can allow learning to occur on a larger scale more efficiently. A Semi-Markov Decision Process (SMDP) [9] is a generalisation of the MDP framework. SMDPs allow actions to take a continuous amount of time, rather than the single unit amount of time each action takes in an MDP. The introduction of continuous time allows for more accurate modelling of decision processes as well as essentially grouping actions.

Intuitively, the SMDP operates by selecting an action a when in state s. This action will lead stochastically to some state s' with well-defined transition probabilities much like with an MDP. However, this transition is not assumed to take unit time, but rather any continuous real value. Until this transition is complete, the SMDP cannot take any more actions. Whilst this transition is taking place, the SMDP accrues reward at a specified rate according to the reward function. The goal of the SMDP is still to find a policy that gives the maximum cumulative reward.

The differences between SMDPs and MDPs are worth highlighting. Apart from the obvious extension of time to a continuous random variable, reward is now given as a rate over time for the duration of a transition. This means that the same transition can have different reward values based on its duration. Furthermore, transitions taking a variable amount of time also affects the discounting applied to transitions of different lengths.

Formally, an SMDP is defined as a tuple (S, A, ρ, P, χ). Where S is a finite set of states, A is a set of actions for each state. $\rho(s, a, s')$ is the constant rate of reward when transitioning from state s to s' via action a. $P(s, a, s')$ is the probability of transitioning from state s to s' when action a is performed. Finally, $\chi(t|s, a, s')$ is the probability distribution for the time taken to complete the transition between state s to s' using action a. If $s(t)$ and $a(t)$ denote the state and action at time t, then the goal of the SMDP is to maximise the expected infinite horizon by finding an appropriate policy. For convenience (and sanity), the expected reward and discounting for a given transition and policy are defined (and the derivation not left as an exercise for the reader):

$$R(s,a,s') = \int_0^\infty \int_0^t e^{-\beta x} \rho(s,a,s') \, dx \, d\big(\chi(t|s,a,s')\big) \tag{2.7}$$

$$\gamma(s,a,s') = \int_0^\infty e^{-\beta t} \, d\big(\chi(t|s,a,s')\big) \tag{2.8}$$

The expected reward is the discounted rate of reward over the transition with respect to the random time variable. It is the accumulation of this expected reward over an episode that is desired to be maximised. Doing so is clearly not an easy task. Iterative approaches to solve this may be used, extensions of Q-Learning to account for continuous time exist [10], however they are much slower to converge than their discrete counterparts.

2.4.2 Options

Using and solving SMDPs is not a trivial task. Further, they are not conceptually simple when compared to MDPs. Options can alleviate this, acting as a more intuitive extension to MDPs while still retaining many of the advantages provided by SMDPs. Options [67] are an extension of primitive actions within an MDP. Essentially, Options allow primitive actions to be chained together into "macro actions". Each Option corresponds to one macro action. Many solutions to problems are best thought of as long chains of actions performed in sequence. Using Options allows any agent exploring an MDP to learn about chains of actions all at once.

More formally, an Option is defined in [67] as a tuple $(\mathcal{I}, \pi, \mathcal{B})$. Here, $\mathcal{I}$ is the initiation set, a subset of S in the MDP designating from which states the Option may be invoked. That is, Option o is only available in states $s \in S$ if $s \in \mathcal{I}$. π is the Option policy. This functions similarly to a policy over an MDP. The Option policy is a function mapping states $s \in S$ to primitive actions. This policy is used whilst the agent is invoking the relevant Option. This Option policy function is what describes the behaviour of the Option. Finally, $\mathcal{B}$ is a termination set, giving probabilities for each state $s \in S$ for termination of the Option.

The agent interacting with the MDP can choose to invoke an Option if it is in a valid initiation state for that Option. The Option then takes over control of action selection until the option terminates. The agent then re-assumes control and continues selecting Options. It is common to view Options as sub-policies that are designed to solve a particular sub-task. The agent then does not need to learn to perform the primitive actions in sequence, only which Options to invoke and when. The Option policy can also only being defined over relevant areas of the state-space in accordance with its initialisation and termination sets.

In order for the agent to use Options, the MDP needs to be reformulated to take them into consideration. For each state $s \in S$, Option O is available if $s \in O_{\mathcal{I}}$. O_s is used to denote the Options available in state s. There is a large similarity between how actions and Options are viewed by the agent within the MDP. Instead of giving the agent access to both Options and actions, each primitive action is encoded as a trivial one-step option. The set of Options available in each state then replaces the action set in the formulation of an MDP. Instead of having a policy π to select actions, the agent exploring the MDP with options now uses a policy μ to select Options in a similar manner.

Strictly speaking, an MDP using Options is no longer Markov. This is because "actions" — which are now Options — can take longer than one unit of time to complete. This causes the "flat" sequence of events to lose the Markov property as future states depend not on the current state, but on the state when the Option was initiated. The Options themselves (viewed as atomic entities), however, are Markov.

In [67], it is shown that MDPs augmented with Options are actually discrete-time SMDPs. This allows many problems that can be modelled with SMDPs to also be modelled by the conceptually simpler MDPs with Options. Sutton et al. [67] use $\mathcal{E}(o, s, t)$ to denote the event that option o is initiated in state s and at time t. Then they go on to give the appropriate reward model for an option beginning in a specific state and terminating after k-steps :

$$r_s^o = \mathbb{E}(\sum_{i=t}^{t+k} \gamma^i r_{t+i} | \mathcal{E}(o, s, t))$$

This is the discrete form of the reward for a SMDP in equation 2.7.

Sutton et al. then show that the transition function also needs to be updated in order to account for Options; if $\delta(s, k)$ denotes the probability that an option terminates in state s after k steps, then $\Delta_{s,s'}^o = \sum_{k=1}^{\infty} \delta(s', k)\gamma^k$ gives the probability of an Option o beginning in s and terminating in s'. An MDP utilising Options can now be defined formally as (S, O, r, Δ), where S is a set of states, O a set of options, r a reward function and Δ a transition function, all as defined above. Further, Sutton et al. show that the optimal state-Option pairs $Q^*(s, o)$ must satisfy the equation:

$$Q^*(s, o) = r_s^o + \sum_{s'} \Delta_{s,s'}^o \max_{o' \in O_{s'}} Q^*(s, o))$$

Finally Sutton et al. go on to show how Q-Learning can be extended to handle MDPs with Options, producing the update rule:

$$Q(s, o) \leftarrow Q(s, o) + \alpha\big(r + \gamma^k \max_{o' \in O_{s'}} Q(s', o') - Q(s, o)\big)$$

after each transition between states s and s' taking k steps, For a discount factor γ.

2.4.3 Using Options

Options can be interpreted as abstracting over time with respect to the action space of an MDP. Options allow chains of primitive actions (or sub-policies) to be viewed as atomic entities. Options have found a lot of use within RL. A few notable examples are given here to show the scope at which Options can be used. In [67], Options are used to represent sub-policies for moving between rooms in a navigational grid-like environment. The Options contrast with primitive actions that only allow the agent to move between adjacent grid cells. Encoding Options in this manner allows the agent to move between larger rooms with ease when searching for the goal state. It helps prevent the agent from becoming "lost" in any one room, and locate small hallways. Using Options for this task increased the convergence speed to near-optimal policies by a factor of 10.

Options and Deep Learning are combined in [39] in order to learn to play the Atari 2600 game "Montezuma's Revenge". This game requires the player to direct an avatar through a two dimensional environment, collecting keys and navigating through doors. In this work, the agent learned to play from pixel input. A convolutional neural network is used to learn to recognise objects from the visual input. A second large convolutional neural network is then used in two steps. Initially learning possible goals, then formulating these goals as Options and learning which Options to select to maximise reward. This touches on the next topic; that of Option generation. In [39] Options are generated from goals identified using a neural network. Generating Options is preferable to hand-coding them, due to concerns of time and difficulty. The next section describes this process in greater detail.

2.4.4 Option Generation

It cannot be assumed that pre-made Options will always be available to learn policies over. Creating Options by hand typically requires a domain expert and is expensive. Moreover, the number of possible options is exponential in Option length and the number of primitive actions. If Options are to be used as realistic solutions, then a way of generating possible sets of useful Options are needed.

By far the most common method for this is to identify "sub-goals" within the environment. The intuition behind this is that in order for an agent to complete its primary goal, it must first complete a number of sub-goals. Options direct the agent between these sub-goals. Then the overall problem is broken down into completing sub-goals in the optimal order, or deciding if any sub-goals are superfluous. Examples of potential sub-goals are picking up a key in order to unlock a door, or navigating between two rooms. Identifying sub-goals is seemingly the difficult part of generating Options. This is because the agent has no a priori knowledge of what is expected of it within a designated environment. We now look at some examples of Option generation that have appeared in recent research.

2.4.4.1 Bottlenecks

One heuristic for discovering sub-goals is to look for bottlenecks in the underlying state-space of the environment. Again, there are multiple ways in which this is attempted.

2.4.4.1.1 Clustering

In [34], a transition graph is created. Each state transition is examined and clusters are created on the basis of reachability by multiple paths. Then any transition connecting a cluster is the only one that does so. This edge of the transition graph is then considered to be the bottleneck between these two clusters, and utilised as a sub-goal. Options can then be created to and from the associated nodes of this bottleneck. The process of identifying bottlenecks takes time $O(E)$, where E is the set of transition edges, so does not scale impossibly with the size of the state-space. This method suffers from the same typical disadvantage as a lot of other Bottleneck methods, chiefly that it only captures bottlenecks of unitary “width”. For larger state-spaces this may not ever be the case — for example, in a gridworld environment where rooms are separated by doors with a width of two states. This can be somewhat alleviated by forming clusters based on different levels of reachability, perhaps requiring more transitions between clusters in order to join them. However, this is still a static heuristic, optimal values of which may not be known a priori.

The paper [34] also proposes locating bottlenecks based on state value, not just environment topology. This seemingly attempts to address the issue of bottleneck width by considering the difference in state value when forming clusters, rather than the number of transitions between clusters — of course, states must still be adjacent to a cluster in order to be added. Whilst this does help alleviate the issue of bottleneck width, the clusters formed may be far from optimal, since the only state values that the algorithm has access to are the current estimates. If these were close to the optimal values, then there would be no need for further learning. This is a particular issue at the beginning of the learning process, when state values are far from their optimal values. This may set up bottlenecks that are non-conducive to learning.

Regardless of the clustering method, the paper [34] then goes on to augment this with frequent item-set mining in order to deduce which bottlenecks are important in order to minimise the length of Options. Options are then generated by using Dynamic Programming to determine policies between neighbouring clusters, as well creating Options based on the frequent sequences constructively.

The results from [34] are somewhat positive. The proposed method was tested on a maze navigation task, as well as the "Taxi" domain and "Soccer bed" task. Within the maze navigation task, the proposed method performs only slightly better than standard Q-Learning. However the bottleneck method outperforms standard Q-Learning by a factor of approximately 2 after a few thousand episodes on the Taxi and Soccer bed domains.

2.4.4.1.2 Ant System Optimisation

Another method for identifying bottlenecks is proposed in [25], utilising the "Ant System Optimisation" algorithm which was pioneered in [14]. Intuitively, this algorithm uses n agents (or ants) to explore a state-space in the form of a graph $G = (V, E)$, where V is the set of states and E the set of state-transitions. The ants drop "pheromone" on each edge they traverse. The ants move randomly, but with a bias towards edges laced with pheromone, to which they are attracted. After each transition, some of the pheromones that have accumulated on each edge "evaporate" in order to prevent cyclical behaviour. The amount of pheromone deposited by an ant decreases in proportion to the amount of transitions made so far. The idea here is that shorter routes will accrue larger amounts of pheromone and will therefore attract more ants and thus, even more pheromone. Moreover, any bottlenecks will have a much larger proportion of ants moving through them. The authors of [25] construct a criterion they call "Roughness" which is a measure of the variance of pheromone gradient over time divided by the square of the total time difference. Lower roughness values indicate a higher likelihood of being a bottleneck. These edges are separated by analysing for growths in roughness when the edges are ordered by roughness. As with most bottleneck methods, once the bottleneck edges are found, Options are created that direct the agent between the bottlenecks. This proposed method again has the disadvantages that it favours bottleneck edges of width one, even when this may not constitute a bottleneck for particularly large environments. The Ant System Optimisation method for identifying bottlenecks [25] was tested on the Taxi Domain and the Playroom environment. The proposed method converges on a near-optimal policy 10 and 5 times faster than Q-Learning for each of these environments respectively. It is important, however, to reiterate that these environments are actually quite simple, and that the bottlenecks are all of unitary width.

2.4.4.1.3 Other Methods

The two methods highlighted here are only a small sample of methods composed to identify bottlenecks within environments. There are numerous others for which detail will need to be omitted for space considerations. These include methods that perform Spectral Clustering on the MDPs estimated transition graph to obtain similarity clusters [70] [38], as well as clustering based on entropy measures of the estimated transition graph in addition to the number of edges joining these clusters [41].

Ultimately, the method used to identify bottlenecks is somewhat irrelevant. The core idea common to all of these papers is to use the identified bottlenecks as subgoals and to define Options directing the agent between them. The clusters that appear from the bottlenecks are often considered small enough that Dynamic Programming methods can be employed to find optimal policies between these clusters. Most methods for identifying bottlenecks lack the ability to work with bottleneck widths larger than 1. This does not scale well for larger environments when a "bottleneck" may actually consist of multiple states or edges.

2.4.4.2 Extended Sequence Trees

Extended Sequence Trees (ESTs) are another method for creating Options automatically. ESTs construct Options based on the state transition history of the agent. They were pioneered in [27]. In order to understand ESTs, the notion of a Conditionally Terminating Sequence (CTS) must be introduced.

Given an MDP defined in the usual manner, (S, A, R, P, γ), A CTS is defined in [27] as a sequence of ordered pairs $\sigma = \langle (C_i, a_i) \mid 0 \leqslant i < n \rangle$. Here, $C_i \subseteq S$ is a subset of the states from the MDP and is referred to as the continuation set. $a_i \in A$ is an action available to the agent. After each time-step, action a_i is performed. This yields some state s', if $s' \in C_i$, then the agent continues to follow the CTS, otherwise it terminates the CTS. The paper goes on to give a construction showing that each CTS has a corresponding Option. Properly utilising CTSs is therefore equivalent to applying Options, and brings all of the corresponding benefits.

The paper [27] then introduces the notion of an extended sequence tree (EST). The purpose of the EST is to compactly represent a set of CTSs, which is important due to the fact that the amount of possible CTSs typically grows exponentially with the length of a CTS. An extended sequence tree is then defined as $\langle N, E \rangle$ where N is a set of nodes and E is a set of edges. Each node represents a unique sequence of actions. Node p is connected to node q by an edge representing action a if q can be formed by appending a to p. The edge representing this is denoted as (a, ψ) where ψ is an eligibility value. . Each node also contains a list of tuples $\langle s_i, \sigma_i, r_i \rangle$ which denote the continuation set of the incident action, as well as appropriate rewards and eligibilities for each state in the continuation set. Intuitively, the tree is formed to compactly represent a “union” of CTSs. The EST also needs ψ and σ decay rates and threshold values in order to determine by how much the eligibility of each node decays after every transition, as well as when to remove nodes from the tree.

A path from the root to another node (not necessarily a leaf) then represents a CTS. Furthermore, the sequence tree can encode many CTSs in a compact structure by taking advantage of branching. The eligibility values are updated to reflect the number of times the agent follows specific CTSs and their quality, with higher eligibility values representing more commonly used CTSs with higher rewards. New CTSs are added to the tree as they occur. The tree is periodically pruned to save space, removing branches with low eligibility values.

In order to use the EST, an agent must select which CTS it wishes the follow. It can then follow that CTS down the tree as a guide for which actions are valid, checking the continuation states at each step. The EST paper [27] suggests learning or using a probability distribution over the CTS to select actions. This allows the whole EST to be used as a high-level policy, deciding which CTS branches to follow. When the agent performs an action not in the tree or enters a state not in a relevant continuation set the agent defaults to its own policy. The agent will add new nodes and edges to the tree (representing new useful CTSs) as it repeatedly encounters transition histories of high reward.

$Sarsa(\lambda)$, Q-Learning and SMDP Q-Learning agents augmented with an EST were tested on three domains in the paper [27]. The agents were tested against agents based using the same RL algorithm without the EST augmentation. The domains they were tested on were a navigational maze, the Taxi Domain and the game of Keep Away.

For smaller Taxi Domain environments, $Sarsa(\lambda)$ and Q-Learning using ESTs converged to a near-optimal solution in 5 times fewer episodes than their more basic counterparts. For larger variations of this environment, the EST augmented agents performed even better comparatively. For the navigational maze environment, the EST augmented agents again massively outperform their counterparts. A notable result is that the EST augmented Q-Learning performs on par with SMDP Q-Learning with hand-crafted Options. This is important, as the ESTs have — over time — crafted Options comparable to a human expert. SMDP Q-Learning also does not improve when augmented with an EST, suggesting these Options are already optimal.

For the game of Keep Away, an SMDP adaptation of $Sarsa(\lambda)$ taking advantage of Tile Coding was used. This was then augmented with an EST. Both agents eventually learned to keep possession of the ball out of the other team’s hands for a maximum of 16 seconds. The EST augmentation agent, however, achieved this in a third of the learning time. It is worth pointing out the the SMDP-$Sarsa(\lambda)$ does not use handcrafted Options, but derives them from Tile Coding, thus the Options can be improved.

A limitation of ESTs to consider is how the size of the EST grows over time. After all, trees with paths of length n can grow to have (b^n) nodes, where b is the average branching factor — or number of actions available at each decision point in this case. With reference to an EST, this would represent every possible Option. However, even storing Options with very few initial actions can lead to a large tree. In [27], the authors have considered this issue. The total number of nodes in the tree seemingly tends to a certain value based on the EST's decay rate. As the decay rate increases, the total amount of nodes in the tree seems to increase exponentially. A trade-off appears to be required between performance and size of the tree; based on the decay rate. This parallels the minimum threshold value for storing λ eligibility values in $Sarsa(\lambda)$. The size of the tree will also affect running time in the obvious way.

The authors of [77] extend ESTs to allow for only partially observable environments. This is done by introducing belief states, where the agent estimates which state it is in based on its limited observations and previous beliefs. The agent models the belief state-space as an infinite grid that is discretised at regular intervals. These belief states are then used instead of states in the EST. This approach is tested on a large number of partially observable environments. Agents using adaptations of Q-Learning for partially observable environments are used. Multiple methods of dicretising the state-space are also compared. The important result, however, is that regardless of which dicretisation method is used, the corresponding belief-EST augmented agent outperforms the unaugmented agent.

2.4.4.3 Association Rule Mining

The third method we explore for constructing Options is Association Rule Mining (ARM). Whilst Association Rule Mining has been a general Machine Learning method for some time, it was first introduced for Option generation for RL in [26].

Within ARM, a trajectory of visited states $\{s_0, ..., s_n\}$ is referred to as a transaction. The goal is to find rules that explain the trajectory causality. A rule takes the form $A \rightarrow B$, where A and B are disjoint sets and is interpreted as "if set A occurs, then with high probability, B occurs also". Two prominent notions in ARM are that of *support* and *confidence*. Support is the rate at which both A and B occur together. On the other hand, the confidence is the frequency at which both A and B occur relative to how often A occurs. The support is often used to disregard a rule that occurs too infrequently. The confidence measures the reliability of the rule. Threshold values to compare these against are often used, with algorithms specifying a minimum support and confidence required for each rule.

For each sub-goal candidate — which need to be identified a priori — the support of the rules $\{sg_0, ..., sg_{i-1}, sg_{i+1}, ...sg_n\} \rightarrow \{sg_i\}$ are computed for each i and permutation based on the state trajectories. Rules with a support lower than the minimum support threshold are removed. The confidence of surviving rules is then computed. Again, these values are compared against the minimum confidence threshold and unreliable rules are removed. The remaining rules are then used to construct a tree structure in a similar manner to an EST, with rules sharing paths until they differ, at which point the tree branches. The goal is the root of the tree, and leaves are sub-goals in reverse order. The agent then starts working towards sub-goals at a leaf and aims toward the root.

Options are then constructed using an external method (possibly dynamic programming or the like) between sub-goals in accordance with the constructed tree. The agent then selects Options based on the sub-goal to which it is closest and progresses through the tree toward the route.

This method was tested on a number of environments similar to a Grid-world domain, except the agent has to collect multiple coloured and numbered keys in a certain order. Variations on this are used, with larger grids, and more keys. Q-Learning was compared to Q-Learning with Options determined by the proposed method. The ARM method massively outperformed Q-Learning, requiring vastly fewer episodes to achieve similar performance.

The main drawback of this method however is that there are exponentially many potential association rules to evaluate. This is somewhat alleviated by the fact that the rules only need to consider sub-goals and not individual states. There are typically a lot fewer sub-goals when compared to the number of states. The other obvious drawback is that this method requires sub-goals to have been identified beforehand. It does not identify them itself. However, it does provide a strong ordering to achieve these sub-goals in.

2.4.4.4 Concluding Remarks For Option Generation

As has been discussed, lots of methods have been proposed in recent years for generating Options automatically. Whilst the majority of these are based on identifying bottlenecks, other more niche methods have also been proposed. Overall, each of the methods has shown improvement over standard Q-Learning or other basic RL algorithms. Unfortunately the Option generation methods are often not directly comparable since the algorithms are not tested against other methods of generating Options. There are also variations in environment and hyper-parameters that prevent this from being done. This is quite disappointing since it is unclear whether any of these methods outperform others or if they each have strengths and weaknesses. Comparing the performance of these Option generating agents against other advanced RL methods such as Reward Shaping and Deep Learning could also aid the understanding of the efficacy and suitability of these methods.

2.4.5 Abstract Markov Decision Processes

When aiming to solve complex problems, it is often helpful to consider the problem from a more general perspective, with focus on the larger picture rather than the fine details. Abstract MDPs are the embodiment of this idea within RL.

An Abstract Markov Decision Process (AMDP) is very similar to an MDP. AMDPs typically correspond to an abstraction of a specific MDP. It is important to note that an AMDP is an MDP in itself. The main draw of AMDPs is that they are used to generalise large MDPs into smaller, more manageable parts, grouping similar aspects together.

The formal definition of an AMDP is similar to that of an MDP, but specifies an abstract entity. An AMDP $\mathcal{A}$ is a tuple $(S_{\mathcal{A}}, A_{\mathcal{A}}, R_{\mathcal{A}}, P_{\mathcal{A}})$. $S_{\mathcal{A}}$ is a set of abstract states. $A_{\mathcal{A}}$ is a function mapping abstract state to abstract actions available in the given abstract state. These are typically higher-level actions designating movements between abstract states. $R_{\mathcal{A}}$ is the reward function for the AMDP. Finally, $P_{\mathcal{A}}$ is the abstract transition function, this works in the same way as for an MDP.

For the AMDP to have any use, we also require a set of abstraction functions $Z = (Z_S, Z_A, Z_R, Z_P)$ which map elements (states, actions, rewards, and transition functions respectively) from a "ground level" MDP to an AMDP. This Z is not considered part of the AMDP, but is characteristic of the abstraction relation between the MDP and AMDP.

The abstraction Z needs to be constructed in a way that captures a distillation of the essence of the underlying MDP. The methods for how we do this form the core of research into AMDPs, but AMDPs can also be hand crafted by a domain expert.

In general, throughout this thesis, for an element of an MDP, we will denote its abstract counterpart using an overbar. Here we assume that the appropriate element of Z is applied to the MDP element. For example, if we have a state $s \in S_{\mathcal{M}}$, and abstraction Z, then we take $\bar{s} = Z_S(s)$, similarly, for an action $a \in A_{\mathcal{M}}$ we take $\bar{a} = Z_A(a)$. Using this notation helps keep our equations clear and concise.

2.5 Guided Learning

Learning can also be sped up if the learner is guided through the process. This is in fact, the whole basis for teaching within society. Two of the primary approaches to guiding agents in this way are Reward Shaping and Curriculum Learning. The work that this doctorate has utilised is Reward Shaping, but both methods are briefly overviewed here as some nice parallels can be drawn between them.

2.5.1 Curriculum Learning

Curriculum Learning (CL) is a Machine Learning technique that aims to increase convergence speed. It attempts to do so through constructing a curriculum of training examples, guiding the agent from "easy" examples to "hard". The intuition is that if the agent is trained on easier examples first, it should attain a level of competence that makes the harder examples more manageable. In CL's introductory paper [7] — which occurred a few years prior to the recent explosion in very deep neural networks — results show that a simple curriculum approach can improve a model's performance at identifying simple geometric shapes as well as predicting the next word in a simple English sentence.

For identifying geometric objects, the first curriculum stage reduced the set of images to circles, squares and equilateral triangles, as opposed to the final curriculum state (i.e., the task) where the images contained ellipses, rectangles or triangles. For the task of predicting the next word, after each curriculum stage the word vocabulary size was increased. It is clear that CL's biggest disadvantage is the need to construct curricula prior to training. This can be costly, as well as requiring an understanding of which training examples are "hard".

Curriculum Learning has also been extended to Reinforcement Learning in recent years [48]. As opposed to utilising easier training examples in the earlier stages of the curriculum, in RL easier "tasks" are used for training. These tasks are just simpler MDPs. The first paper to utilise this approach gave a myriad of ways in which these simpler MDPs can be constructed. These methods include focusing on scaled down areas of the original problem to achieve a subgoal (which translate into Options), detecting mistakes made by the agent and having them replay the moments leading up to the mistake, as well as reducing the action space unless prerequisites are met.

For the game of "Ms. Pacman", simply training the agent on previous levels (which entail simpler mazes) was enough to yield significantly improved performance compared to the Q-Learning baseline. A similar improvement in performance was achieved for the game of "Half Field Offence" — a sub-game of

"RoboSoccer". The aim of this game is for your players to score a goal against a number of defenders and goalie. The curriculum used here were a series of sub-tasks dedicated to teaching the agents to shoot and dribble the ball. This effectively taught the agents the skills needed to coordinate and position themselves. Some of the approaches presented in the paper [48] still require a high level of domain knowledge and knowledge of sub-task difficulty is still assumed. However this paper still extended Curriculum Learning to RL and other guidance methods typically suffer from the same drawbacks.

Recent work has focused on trying to address this issue of selecting tasks for a curriculum [49][69][50], and they have been successful in the limited domains they have been tested in. However a detailed explication of these approaches is beyond the scope of this brief overview.

2.6 Reward Shaping

Reward shaping is another example of guided learning within the field of RL. Domain experts can assist the learning agent to find state-action trajectories of high reward.

In reward shaping an additional reward is given to the agent based on the current transition. This additional reward function is termed F and referred to as the extrinsic reward, and represents prior knowledge of the environment that is given to the agent. Using Reward Shaping the Q-Learning update rule becomes:

$$Q(s,a) := Q(s,a) + \alpha\Big(R(s,a,s') + F(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big)$$

For transitions with a high extrinsic reward, the new $Q(s,a)$ value will become larger than without F. This leads to the agent being more likely to select action a in state s. If the extrinsic reward is chosen arbitrarily, then the policy that the agent converges to may change, and no longer be the optimal policy for the non-shaped problem [55]. In order to avoid this, each state must be given a potential $\phi(s)$. This potential is provided by a domain expert and may broadly correspond to the desirability of a state. The specific potential function used is not so important as how it is used to create the shaped reward. The extrinsic reward is defined as $F(s,a,s') = \omega(\gamma\phi(s') - \phi(s))$ for a scaling constant ω. It has been shown that doing this keeps the convergence properties of Q-Learning [51]. This is referred to as Potential-Based Reward Shaping (PBRS). Intuitively this occurs because the additional rewards "cancel out" over the course of the trajectory. In [51], the authors demonstrate that the total effect from this reward on the optimal Q-functions for the original MDP M and the shaped MDP M' is no larger than a constant ($\phi(s)$ for terminal state s). It follows from this that an optimal policy for M' is also optimal for M. One thing that we cannot guarantee is the type of optimal policy that will be identified. It is possible that new concerns such as reward hacking or safety issues with the shaped policy could occur, however these are not more likely to occur than with the unshaped policy and could just as likely be safer or prevent reward hacking.

Using PBRS can yield some impressive speed-up when compared to traditional RL. However, the potential function, representing domain knowledge, must be given as an input to the agent. It has also been shown that "bad" potential functions can slow learning [55]. Therefore care must be taken when selecting the domain knowledge to encode as a potential function to ensure it is correct. Consideration must also be given to the feasibility and cost of such an encoding.

The potential function used can be altered to include actions. This is referred to as Potential-Based Advice [75]. The extrinsic reward function can then be formulated as $F(s, a, s', a') = \omega(\gamma\phi(s', a') - \phi(s, a))$. Here a' is selected according to the agent's learning update. With Q-Learning this entails $a' = \text{argmax}_a(Q(s', a))$. In order to keep the same guarantees of policy invariance the agent must be what is called "biased-greedy". That is, the policy satisfies:

$$\pi(s) = \underset{a}{\text{argmax}}(Q(s, a) + \phi(s, a))$$

This is proven to be required for policy invariance [75]. An advantage potential-based advice has over standard PBRS is that potential-based advice can encode knowledge about both states and actions into the shaped reward. PBRS can only encode state-based knowledge. The state-action-based advice can be more useful if we want to encode information about desirable behaviour, rather than a desirable state. This can help avoid becoming trapped in local optima, where the agent reaches a locally optimum state and then is punished for any action it takes — potential-based advice will reward the agent for selecting a desirable action.

2.6.1 Reward Shaping With AMDPs

Instead of setting the value of the extrinsic reward for every transition — which represents an inordinate amount of domain knowledge — an AMDP approximation of the environment can be solved in order to yield the potential function [17]. The AMDP, and as consequence, abstract dynamics of the environment, are assumed to be given by a domain expert. It is also possible to construct this AMDP by exploring the state-space of the MDP, as is done in [42].

For small enough AMDPs we can solve them using Value Iteration. Value iteration gives a value $V(\bar{s})$ for each $\bar{s} \in S_{\mathcal{A}}$ of the AMDP. It has also been shown in [51] that shaping using $\phi(s) = V(s)$ for $s \in S$ of the MDP gives the optimal shaping function, although the authors are keen to point out that even when $\phi(s)$ and $V(s)$ are far apart, then $\phi(s)$ can still be a useful shaping function.

The potential function is then set as $\phi(s) = V(Z_S(s))$. This means that the potential of each state is set as the value of its abstract counterpart $\bar{s} \in S_{\mathcal{A}}$ where $\bar{s} = Z_S(s)$. Whilst this is not the optimal $\phi(s) = V(s)$, experiments in both [17] and [42] show AMDP based reward shaping outperforming Q-Learning by large margins.

2.6.2 Knowledge Revision

As previously mentioned, the AMDP, or important parts of it, are often assumed to be available as knowledge provided by an expert. Whilst this assumption cannot always be held true, the amount of knowledge imparted to the agent can vary with the level of abstraction used. This allows the knowledge requirements to vary.

Another concerning factor is the matter of the provided knowledge being incorrect. Much of the time, the abstractions are given by human domain experts. Humans are notorious for their ability to make mistakes — look outside for confirmation of this. Even machines may occasionally falter if the environment dynamics are not specified correctly, or if the environment suddenly changes. It therefore seems likely that the abstractions may, at times, contain incorrect knowledge that will guide the agent off-course. This can impede the learning abilities of the agent.

The solution to this issue is to allow the agent to update its abstractions to reflect its experiences. This is done in [18], where the agent keeps track of which low-level transitions were performed in an episode. These low-level transitions are then mapped to transitions in the AMDP. That is, if transition $s \xrightarrow{a} s'$ occurs on the low level, then $\bar{s} \xrightarrow{\bar{a}} \bar{s}'$ is recorded, where $\bar{a}$ is the abstract action between $\bar{s} = Z_S(s)$ and $\bar{s}' = Z_S(s')$. The transition probabilities for the AMDP — that is $P_{\mathcal{A}}$ — are then updated using:

$$P_{\mathcal{A}}(\bar{s}, \bar{a}, \bar{s}') = \begin{cases} P_{\mathcal{A}}(\bar{s}, \bar{a}, \bar{s}') + \alpha(1 - P_{\mathcal{A}}(\bar{s}, \bar{a}, \bar{s}')), & \text{if } \bar{s} \xrightarrow{\bar{a}} \bar{s}' \text{ occurred} \\ P_{\mathcal{A}}(\bar{s}, \bar{a}, \bar{s}') + \alpha(0 - P_{\mathcal{A}}(\bar{s}, \bar{a}, \bar{s}')), & \text{otherwise} \end{cases}$$

Note that we have altered the notation used in [18] in order to more coherently fit with the notations used throughout this thesis.

The intuition behind this update is that the agent is correcting the transition probabilities based on its empirical experiences to more accurately reflect the MDP it inhabits. Similar to the Q-learning update rule, this approach "nudges" the probability in the direction observed. If the agent observes a transition that is previously unseen, it adds this transition with probability 1. After each episode — or batch of episodes — the AMDP is re-solved with these new probabilities.

Doing all of this lowers the probability of a transition if it does not occur — either because it is a transition that gives low return, or because it is not actually possible to make. When the AMDP is re-solved, this smaller transition probability lowers the value given to these transitions, thus reducing the extrinsic reward accordingly.

In [18], this approach is tried on domains which extend the navigational gridworld problem. The Knowledge Revision agent is given varying incorrect knowledge, including irrelevant, missing and incorrect information about the locations of the goals. An AMDP-based reward shaping agent without knowledge revision is given the same incorrect knowledge. And a third agent, identical to the second, is given correct knowledge. The knowledge revision agent outperforms the agent with incorrect knowledge and no revision, and also performs comparably to the agent given correct information.

This is a positive result as the agent can, over time, self-correct any mistakes that the domain expert made. However, it must be said that this approach only fixes errors that were made when defining the transition probabilities. it does not address any issues that may arise from incorrect knowledge relating to the existence of abstract states or actions.

2.6.3 Automatic Shaping and Decomposition of Reward Functions

In *Marthi's* paper *Automatic Shaping and Decomposition of Reward Functions* [42] the author gives an algorithm for constructing an AMDP from ground-level observations. This approach shares some similarities with the approach we propose in the upcoming Chapter 4. We detail the approach here and highlight the differences in detail after our proposed method is introduced.

Marthi's approach begins with an initial learning phase. The process is given a state abstraction function z and a set of Options O. The agent repeatedly selects and follows options selected at random from O until Option termination. The agent records the initial state s, final state s', the option invoked o and the cumulative reward received r. This Option experience $(Z_S(s), o, r, Z_S(s'))$ is used to update estimates to AMDP transition and reward functions $\bar{P}$ and $\bar{R}$ using a running average. This process is repeated until a certain number of steps has elapsed. Then the AMDP $(Z_S(S), O, \bar{P}, \bar{R})$ is returned.

One notable aspect of this approach is that it requires the state abstraction function Z_S and available options to be defined a priori, somewhat limiting its power to domains wherein these are already known. However, the authors do claim that the option set O can simply be primitive actions if no options are known. From the state abstraction function and options it infers abstract transitions and reward function directly, which would correspond to abstraction functions $Z_P(P)$ and $Z_R(R)$, though the functions themselves are not made explicit.

In [17], this approach is used to augment a Q-Learning agent play Othello. While this approach drastically improves the agent's performance compared to Vanilla Q-Learning, the details of the state abstraction function Z_S encode a lot of knowledge about the game. The baseline agent uses features to handle the game's size. The board positions are segmented into corners, edges, pre-corners (diagonally adjacent to corners) and internal positions. A notion of "advantage" is introduced over a set of states, detailing the number of pieces one player has over the other in that set of states. The game is further partitioned into three phases of equal length, and for each phase there is a feature equal to the advantage of the agent for each of the board position types. The baseline agent uses these features as the ground representation for Q-Learning.

The state abstraction function is then a map from the board position to essentially a tuple consisting of the advantage of corner squares, the phase of the game, and the advantage of non-corner squares binned into 5 intervals. This is implicitly encoding a lot of knowledge about the importance of corner positions and game-phase relative to the other features in Othello. It does this by merging the other features (pre-corner, edge and internal positions) into one and introducing binning to reduce the number of values that the merged feature can take on.

Two Options are then used, one-step random and one-step greedy, in the shaping function learning process. Although it is not made clear what the greedy option is greedy relative to given that this occurs prior to the Othello learning process, in the AMDP learning process. It may be that the one-step greedy Option is selecting action $a = \text{argmax}_a(\sum_{s'} \hat{P}(s,a,s')\hat{R}(s,a,s')))$ — the estimated weighted mean abstract reward the AMDP has learned *so far*, but this is an educated guess. Unfortunately the link to the code in the paper no longer works.

Whilst this approach is a definite improvement over manually labelling $\phi(s)$ for each state from a domain expert, a non-trivial amount of domain knowledge is still required, not just knowledge of the specifications of the game, but knowledge of advantageous board positions.

2.6.4 Multi-Grid Reinforcement Learning

Multi-Grid Reinforcement Learning (MRL) [29] is another method for shaping (using AMDP-based reward shaping techniques outlined in Section 2.6.1). a learning agent based on its own abstract experiences. In MRL, the environment is twice partitioned uniformly at resolutions. The finer-grained partition is used to represent the ground state-space, and the coarser partition is used to represent the abstract state-space. MRL was designed with continuous state-spaces in mind, and uses these two partitionings to discretise a continuous state-space in order to allow traditional tabular RL methods to work. The corresponding ground or abstract state for a continuous state is given by the functions G_S and Z_S respectively.

The agent utilises a Q function to record its estimation of the values of ground-state-action pairs — similar to most tabular RL algorithms. Additionally, an agent utilising MRL keeps a function V to record its estimation of the value of abstract-states. At every interaction the agent makes with its environment, it receives an experience tuple (s, a, r, s), the Q-function is updated for values $(G_S(s), a)$ similarly to a PBRS-augmented agent, basing its tabular RL update on $r + V(Z_S(s')) - \gamma - V(Z_S(s))$. If $Z_S(s) \neq Z_S(s')$ then the values $V(Z_S(s))$ are also updated, performing a simple update based on the accumulated reward Σ_r — the cumulative reward received by the agent while it was in abstract state $Z_S(s)$.

Whilst no explicit AMDP is used by MRL, the value function of the abstract states of one is implicitly built up in a model-free manner. MRL was shown to achieve higher performance in the Mountain Car domain when compared to a simple Q-Learning agent operating on just the fine-grained state-space. MRL is, however, a product of its time. MRL was published in 2008 — five years before neural networks came to dominate the RL landscape and change how continuous state-spaces are handled entirely.

One of the initial contributions of this thesis has been to modernise and improve MRL for this brave new world of Deep RL. It is not enough to simply replace the tabular ground partitioning with a neural network, changes must also be made to how the algorithm computes and utilises abstract values and transitions. The method itself is given in 4.4. Further, empirical results are given that show the improved version outperforms DQN as well as MRL with a neural network.

2.7 Unification of AMDPs and SMDPs

RL algorithms that use AMDPs and SMDPs share many similarities. Discrete-time SMDPs can be considered as analogous to MDPs with Options. Both of these methods abstract away aspects of the MDP, making the problem easier to solve. They also both have a hierarchical structure — MDPs augmented with Options can elect to invoke Options to temporarily control the agent and there is an intrinsic hierarchical relationship between an AMDP and its associated MDP. Both systems are dependent on external knowledge. Options need to be defined (although recent methods do exist for generating these empirically), AMDPs need state abstractions and abstract transitions. In either case, solving either an AMDP or MDP with options is far easier than solving an SMDP due to the added complexity of calculating the continuous time-steps and solving the integral for SMDP reward .

Options (and SMDPs) provide temporal abstractions to the agent , they chain together primitive actions to provide macro-actions. AMDPs, on the other hand, provide both spatial and temporal abstractions (strictly, we are only "abstracting" over space, but moving through this space takes time and we are considering this an atomic action in the abstraction). This is an important difference. Consequently, every AMDP-MDP relationship has a corresponding MDP with Options formulation. To see this, for some MDP and AMDP, create an Option representing each abstract transition over appropriate states. These Options are then equipped for us by the MDP.

The reverse is not true, however. Since MDP states can belong to many different Option initialisation sets. Typically each state in an MDP only maps to one abstract state in an AMDP. Options that use initialisation sets of these sorts cannot be represented in AMDP form. This makes SMDPs more expressive. However an AMDP-MDP relationship contains more structure. It is not entirely clear which property is more conducive to increasing learning speed. There has been very little (if any) work on comparing the two.

2.8 Deep Learning

In order to keep up with the relentless pace of modern RL research, an understanding of deep learning is necessary. The last decade has seen RL research move away from tabular methods and traditional function approximation methods, instead focusing on utilising Deep Learning. This has had a profound effect on RL as a whole, allowing much more complex and realistic tasks to be solved, but at the cost of apparent post-hoc analysis and theorising.

Deep Learning is usually comprised of using a neural network with many hidden layers, in order for the network to learn suitable salient features to help it achieve its goal.

A neural network consists of a number of nodes which take values as inputs, and output a new value based on the node's internal function, and the inputs the node receives. Chaining nodes together output-to-input, allows the network to grow deeper and perform more complex calculations. The output of a node is multiplied by the weight of the edge before being received by another node as input. This means that for a single node receiving vector x as input, with weights w and biases b, the output of a node will be $g(\vec{w} \cdot \vec{x} + \vec{b})$, where g is the node's internal function.

Propagating an input through the network is referred to as Forward Propagation. This is often repeated for many values at once, and can be sped up by replacing the vector calculation with a matrix calculation.

In order to train the network to made accurate predictions, a cost function is used to evaluate outputs from the neural network when compared against known values. From this cost function, derivatives are "back propagated" through the network using the chain rule in order to update the necessary parameters to decrease the cost.

After sufficiently many iterations of back-propagation, the cost function becomes close enough to a minimum value and the network is able to predict values accurately.

From a historical perspective, algorithms utilised neural networks before they became "deep". The Perception algorithm [58] enabled individual "neurons" to learn linear classification functions. The introduction of the back-propagation algorithm [59] allowed for an easy method to update weights of connected neurons. This led to what we now refer to as feed-forward neural networks. Their application to large-scale problems was limited by the available computing power at the time. The modern increase in available computing allowed for training larger, deeper networks. In the next section we see how this was applied to RL.

2.8.1 Deep Q-Learning

Deep Neural Networks can be used as a function approximation method for RL. A simple example of this is with Deep Q-Learning (DQN) [45]. DQN uses a neural network to represent the Q-Table from Q-learning. As an input, the network takes a state, and outputs a single value per available action — the $Q_\theta(s,a)$ approximation. The DQN algorithm mimics tabular Q-learning, selecting an action $a^* = \max_a Q_\theta(s,a)$, computing the next state s' from s and a, and then receiving reward r. DQN differs from tabular Q-Learning by storing these experiences $e = (s,a,s',r)$ in a memory buffer. Samples from the memory buffer are then drawn uniformly for the Q-update step in order promote a smoother learning process. Given a sample $E = \{(s_1, a_1, s'_1, r_1), ...(s_m, a_m, s'_m, r_m)\}$, y_i is computed as $r_i + \gamma \max_a Q_\theta(s_i, a)$. $\sum_1^m (y_i - Q_\theta(s_i, a_i))^2$ is then used as the cost and the network updates its weights θ for each of the m examples using back propagation. This is principally the same as the tabular Q-Learning update rule, except operating on batches of sampled experiences.

DQN was one of the first RL algorithms to utilise Deep Learning, and has already had its performance surpassed by newer approaches and iterations, however it still performs admirably one complex domains such as Atari games, as outlined in the original paper [45].

2.8.2 Convolutional Neural Networks

The real power of DQN and other deep learning becomes manifest when Convolutional Neural Networks (CNNs) are introduced. CNNs are a type of neural network architecture that are able to work with high dimensional data easily — most notably, images. CNNs were instrumental in DQN performing well on Atari games due to the fact that Atari games have pixel-based visual state-spaces.

A CNN utilises — as one would expect — layers that perform convolution operations on their input. A convolution — or, to be more accurate, a cross-correlation — is a binary operation that can be performed on matrices.

Allow us to represent images as matrices where $I(x, y, z)$ refers to element (x, y, z) of image I. We need three dimensions to account for colour-depth. Three-dimensional convolutions will also be necessary when we come to convolutional layers that take the output of previous convolutional layers as input.

We will denote convolutions as $\oplus$, where the convolution of image k with image I is :

$$I'(x, y, z) = (k \oplus I)(x, y, z) = \sum_{s=-N}^{N} \sum_{t=-M}^{M} \sum_{u=-O}^{O} k(s, t, u) I(x - s, y - t, z - u)$$

Where I' is the resulting matrix. For simplicity we are assuming the Image is of size $(2N + 1) \times (2M + 1) \times (2O + 1)$, although it's easy to account for images with an even number of pixels.

It's important to note that this operation is not associative. The matrix k is referred to as a *kernel.* Kernels are often much smaller than the image matrix and can be used to identify features present in the original image, with the relevant features appearing with higher values in the resultant matrix.

Within a neural network a convolutional layer consists of c kernels of three dimensions (width, height and depth), which are each applied to the input to that layer. It is important to note that typically the kernel depth is made to match the input depth. Each of the c resultant matrices are then two dimensional and can be "stacked" to create a three-dimensional output of depth c . The output

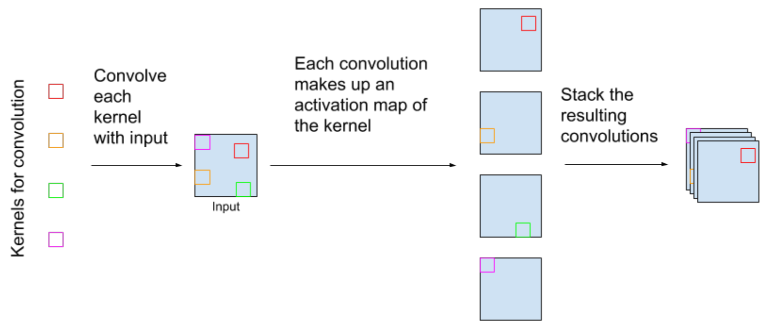

Kernels for convolution
Convolve each kernel with input
Input
Each convolution makes up an activation map of the kernel
Stack the resulting convolutions

Figure 2.1: Convolutional Layer Process

is also passed through an activation function. Figure 2.1 gives a visualisation of this process.

The keys to identifying useful features are the kernels. There are seemingly no “universal” kernels that always identify desirable features. The kernels are therefore learned by the agent as weights in the network.

The intuitive idea between chaining convolutional layers together is that each layer identifies certain features from its input, which the next layer can then deduce “higher-level” features from.

After a number of convolutional layers, there are usually fully-connected layers in order to interpret these identified features. The rest of the neural network is the same as a standard neural network.

Back-propagation operates on the same principle as fully connected-layers, although accounting for the difference in how the network nodes are connected. The kernel weights are updated from the back-propagation algorithm in the same way that the connection weights are updated in fully-connected layers.

As with any architecture or technique, the variations upon this basic idea are innumerable and this description is just an introduction to utilising convolutions within neural networks. CNNs see broad use within deep learning, particularly when working with images. We will make extensive use of CNNs in Chapter 5 in order to handle image-based RL environments.

2.8.3 Deep Reinforcement Learning and Abstraction

One of the appeals of deep RL is that the agent learns its own abstract features during training in an end-to-end fashion. However, due to the opaque nature of neural networks, relying on these end-to-end learned abstract features may not yield optimal abstractions. Some efforts to improve learning performance by learning additional abstractions have been made.

A notable example is Hierarchical Deep Q-Learning (HDQN) [39]. This method operates on two levels, a controller which learns to achieve sub-goals (Options) on a ground-level, and a meta-controller which learns to set sub-goals accordingly. The two levels of agent utilise an actor-critic structure, with the meta-controller providing a reward relating to achieving sub-goals to the controller. This approach was applied to the Atari game Montezuma's Revenge (MR). An unsupervised object detection algorithm was utilised to identify candidate objects from the screen. The meta-controller then learned to formulate goals as moving the agent to candidate objects or positions, for example, moving to a key or a door. This approach outperformed DQN which cannot make any significant progress in MR after millions of training steps.

AMDPs have been employed to increase learning performance in Deep RL in Deep Abstract Q-Networks (DAQN)[57]. Here, two levels of agent are used, similar to HDQN. The difference between DAQN and HDQN is chiefly that DAQN has its varying levels of agents operate over different state-spaces. HDQN uses the same state-space for both levels of agent. In DAQN, the abstract state-space is sets of attributes defining the current low-level state-space at a high-level. Abstract actions can then be considered as the intended changes in attributes. The low-level agent is given rewards based on achieving goals set by the abstract agent. The abstract agent receives rewards from the environment based on the low-level agent's actions. The abstract agent sets goals based on abstract actions it selects, and has learnt to exist from previous experience. Another difference between HDQN and DAQN is that in DAQN, the abstract level uses a tabular representation, and the lower level uses a neural network. DAQN was tested on a toy version of MR. This toy version reduced the continuous pixel representation into a gridworld-esque environment. An abstract state function was supplied to the agent, outlining locations of doors and items. DAQN outperformed Double DQN and HDQN on this environment, however performs worse on the real MR domain. The authors of DAQN [57] contend that this is because in the real MR environment, there are timing based elements that the abstract level cannot handle well. Although it is also possible that the tabular-AMDP representation is hindering learning as well, and that a more powerful abstract representation could improve upon this.

Both of the above approaches suffer from needing information about the environment provided a-priori, either in the form of a pre-trained object detector as in HDQN, or as abstract states being predefined and identifying areas of interest as in DAQN. Both of these options are expensive, either in terms of human effort, or computational time.

2.8.4 Option Heads

The concept of Option heads [5] is a method that has tried to utilise Options within Deep RL. The idea here is to use a Neural Network for a few layers to identify features, after these features are identified the network branches into smaller layers which are kept separated. These branched layers are referred to as “Option heads” and intuitively represent different Options for the network to learn and use. During training, an Oracle is used to select which option head to evaluate and train based on the current time. Each Option head has its own memory replay buffer and learns in the same manner as vanilla DQN. During evaluation, a separate supervisory network selects which option heads to evaluate. This is trained to select option heads separately. It is not clear how to justify the existence of an oracle for training, nor why the supervisory network is not trained concurrently with the option heads. The option heads paper [5] seems to be trying to show that the initial layers of the neural network can learn common features that are useful to different Options, (Especially when these initial layers are convolutional) reminiscent of transfer learning. While their results confirm that for their simple testing domain this is true, the paper does not expound on how such a network could realistically be trained without the use of an Oracle.

As an interesting note, [53] tries a similar sharing of convolution layers for the more challenging domain of Atari games and finds that the agent learns to become more proficient in certain games at the expense of others. Vanilla DQN outperforms their convolution sharing variant.

2.8.5 Hierarchical Actor-Critic

Another approach to employing abstraction within deep RL is that of Hierarchical Actor-Critic (HAC) [40]. learns a hierarchy of policies of varying levels of abstraction simultaneously. The abstractions are constructed temporally rather than spatially. All of the hierarchical levels share the same state-space. Each hierarchical level proposes a “goal” in the form of a state for the lower level to reach. These goal propositions are the actions of the above-ground-level agents. The ground-agent uses the primitive actions of the environment.

This algorithm is heavily influenced by Hindsight Experience Replays (HER)[3], HER is an extension of standard experience replays where goals are added to the learning agent’s experience replay as well as states, actions and rewards. The goals added are both the actual goal, which may or may not have been achieved, as well as the actual state reached reformulated as an “achieved” goal.

HAC aims to reformulate HER into a hierarchical setting. Within the HAC approach, each level operates by attempting to achieve its goal in a set number of actions (or sub-goal propositions). After this set number of actions, a SARSA-like tuple is added to the experience replay. These so called “hindsight action transitions” are novel in the fact that the “action” is replaced with the state that was actually reached. In essence this is using “hindsight” to change the “intention” of the action.

Additional transitions called “hindsight goal transitions” are also added to the experience replay. This time, two transitions are added. The first is the original transition that actually occurred. The second is where the proposed goal is changed to reflect the final state that was actually reached by the agent. The intuition behind this is that, even though the goal wasn’t achieved, the agent has learned a viable path to the state that was reached, and that this might be useful later on.

These two transition types added to the experience replay work in tandem to reduce the sparsity of the environment reward. They make up the core of the method, and other aspects of the method exist only to ensure that proposed goals are sensible and that sufficient exploration occurs on each level without damaging the levels above.

The HAC method was evaluated on a grid-world domain, as well as more challenging physics-based simulation environments. They demonstrated that their approach outperformed relevant baselines, as well as showing that a 3-level version of HAC outperforms a 2-level version.

One detail about HAC that is worth criticising is that it is not clear how to test whether sub-goals or goals were achieved. Since the goals are states, there is clearly some distance measure to the goal that needs to be evaluated (in the case of continuous environments). However, useful distance measures are hard to construct in very high-dimensional space. In the Appendix of the HAC paper it is said that “Goal and sub-goal achievement thresholds were hand-crafted.”. It is hard to evaluate from this the extent to which domain knowledge was used to hand-craft such thresholds and the extent to which these thresholds transfer over to other environments.

2.8.6 Overview

Each of these approaches utilises abstraction in some form for deep learning. The different methods share similarities in the “spirit” of what they are trying to achieve. They differ in either the dimension they abstract (space or time) or by the external information they require. None of the approaches are “end-to-end” and require some form of expert knowledge to properly operate.

2.9 Concluding Remarks

This past chapter has been a rather whirlwind overview of the current state of RL — particularly in its relation to abstraction. There are a great many other aspects to RL that have not been covered, the field is so vast and moving at such a relentless pace now that it is impossible to cover it all.

Focusing on what we have seen, however, shows a plethora of methods for utilising and generating abstractions, whether that be through Options or AMDPs or any of the myriad of other approaches.

Abstractions in general imbue the agent with a sense of the “bigger picture”. The abstraction allows planning and feedback with regards to this “bigger picture” which helps the agent consider its actions and experiences in a broader context. It is no surprise that utilising these abstractions can help the agent learn faster.

It is the generation of these abstractions that are most interesting and have the most potential for improvement. We saw that even simple methods, such as generating abstractions from bottlenecks in state transitions, have been used effectively to increase learning performance.

Whilst a vast amount of research has been done into this area, there are still many directions in which further work into abstraction generation can be done. It is this direction that my work has ventured.

The following chapters delve into this topic further. Chapter 3 shows that AMDPs based on uniformly partitioned state-spaces can improve learning performance for a flag collection domain so long as the available abstract transitions are known a priori. Chapter 4 builds on this approach, removing the assumption of existing abstract transition knowledge. This chapter also elevates the technique to handle environments with small continuous state-spaces utilising neural networks. Chapter 5 removes the discretization in abstract states and allows for continuous state abstraction. This chapter also begins to utilise convolutional layers and auto-encoders in order to handle higher dimensional data.

Chapter 3

Empirical Analysis of the Feasibility of Uniform State Abstraction For Reward Shaping

In this chapter we take a first look at a simple method for utilising reward shaping based on uniform state-space partitions irrespective of the dynamics of structure of the underlying domain. We construct an AMDP from these state-space partitions and formulate a reward shaping function from this AMDP. Many such AMDPs are constructed from uniform partitionings of varying granularities. We utilise variations of a basic navigation domain to evaluate our shaping functions and compare them to shaping functions devised by a domain expert. The aim here is not to produce a generalised algorithm for generating shaping functions but instead to focus on the feasibility of successfully using shaping functions constructed from AMDPs consisting of uniformly abstracted states.

We see broadly positive results from this method, performing comparably to or better than agents shaped by functions derived from hand-crafted AMDPs. We go on to note some potential issues that may arise when constructing AMDPs from uniformly partitioned state-spaces, but we also see that these issues do not manifest in practice. The overall method outlined in this chapter also serves as an introduction to the method in Chapter 4, which builds and expands on the approach here.

3.1 Flag Collection Domains

The class of domains which we utilise for this feasibility study are that of Flag Collection domains. A Flag Collection domain is an augmentation of the navigational Gridworld environment. The agent is tasked with traversing a discrete grid of cells, locating and picking up flags and taking them to the goal. The states of the environment are the grid cells, as well as which flags have been picked up so far. The agent can move in the four cardinal directions one cell at a time. In this environment the agent is given a reward of -1 after almost every transition. The exceptions to this are when it reaches the goal, at which point the agent receives a reward proportional to the number of flags collected (1000 for each flag collected). The agent also receives a smaller reward of 100 when it picks up a flag. The agent's task is made more difficult by impassable walls, spread throughout the domain. Further, the domain does not "roll", the agent cannot move out of the area and reappear on the other side. Episodes within the flag domain only terminate when the agent reaches the goal state.

Flag collection domains were chosen for their low difficulty and the ease of creating AMDPs due to the intuitive "natural" abstraction structure the state-space allows. Further, the simplicity of the transition dynamics of the environment makes analysis of the agent's decisions and policies easier. These domains are small and simple enough that they can be solved easily with tabular RL methods. The focus then can be on evaluating the effects of introducing reward shaping via AMDPs with uniform states and comparing those to agents employing shaping functions from hand-labelled AMDPs as well as unshaped agents.

Six variations on the flag collection domains were used, each trying to highlight a property the domain may possess. Table 3.1 summarises the individual variations. A visual depiction of each domain is further given in Figure 3.1.

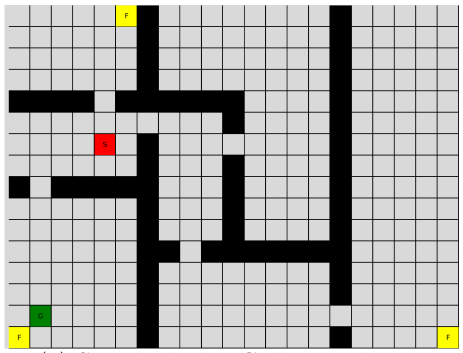
3.2 Constructing the AMDP

Recall that an AMDP shares the same form as an MDP, that is, we must construct an AMDP $\mathcal{A} = (S_\mathcal{A}, A_\mathcal{A}, R_\mathcal{A}, P_\mathcal{A})$, along with abstraction functions Z mapping elements from the original MDP $\mathcal{M}$ to $\mathcal{A}$. We will visit each of these elements in turn and show how we construct them. Many of these are constructed by hand utilising domain knowledge for the Flag Collection task. This is fine, however, since we only want to show that reward shaping functions based on the uniform partitions of the state-space can be beneficial to the learning process.

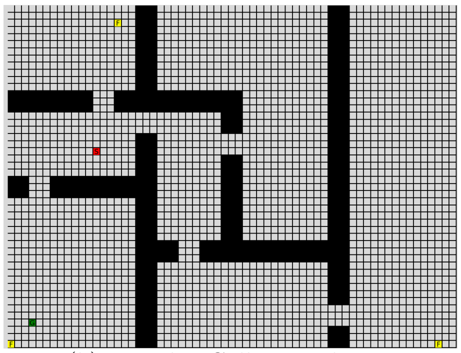
The state-space $S_\mathcal{A}$ is constructed using a uniform state-space partition. With discrete state-spaces there are two options for how this can be done. Either the

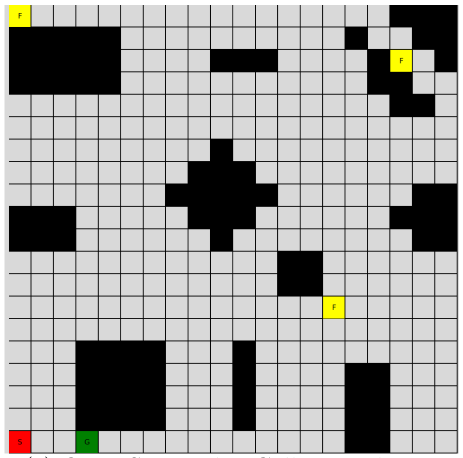
Name	Description
Standard (see Figure 3.1a)	This is the basic environment for the Flag Collection domain. This has been used to test many RL algorithms previously and serves as a standard benchmark [16][18].
Big	The ‘Big’ variation is the same as the basic one, except every cell in the basic environment becomes a set of 3x3 cells. This increase in size will test the scalability of the method.
Open	This variation is mostly open space with a few obstacles to traverse around. This variation was selected to see how the agent handles a lot of wide open space and many action choices available.
Strips	The ‘Strips’ variation is composed of rooms of long, thin strips. These were chosen to directly contradict the assumptions made by the upcoming method that abstractions are uniformly square. This was done in order to see if the agent is able to adapt to environments that are very different from their abstract representation.
High Connectivity	The ‘High Connectivity’ environment is full of rooms that are very interconnected. This fits with the abstract agents assumption that it can always transition from one room to the next.
Low Connectivity	The ‘Low Connectivity’ variation uses the same room layout as the ‘High Connectivity’ variation, except now the agent cannot transition between most rooms. More walls are present to block off its movement. Both the high and low connectivity versions were selected in order to compare how the agent copes when transitions the abstract agent uses for shaping do not exist.

Table 3.1: Descriptions of the variations on the Flag Collection domain that were selected.

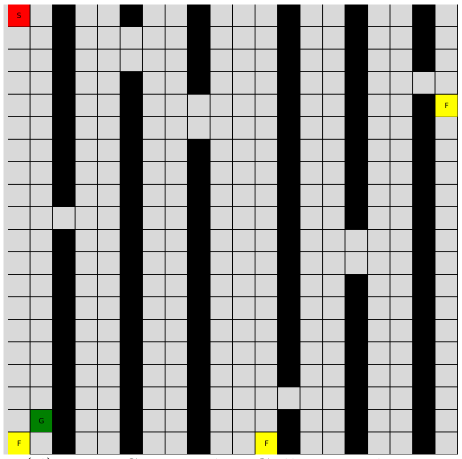


(a) Standard Flag Collection domain

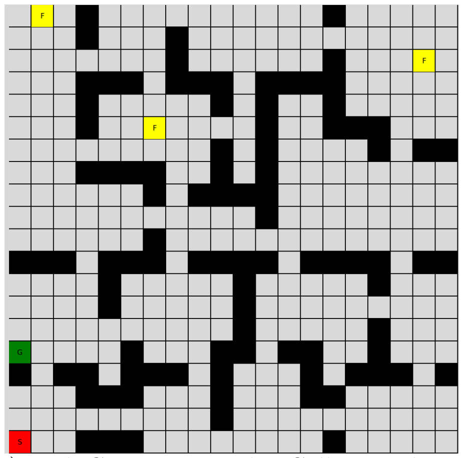

(b) Big Flag Collection domain



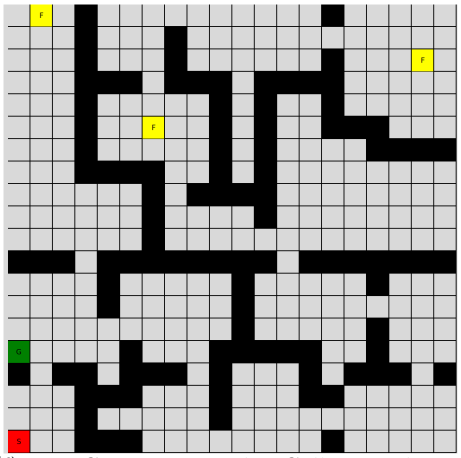
(c) Open Space Flag Collection domain



(d) Long Strips Flag Collection domain



(e) High Connectivity Flag Collection domain



(f) Low Connectivity Flag Collection domain

Figure 3.1: Graphic representation of each variation on the Flag Collecting domain. The agent begins at the red cell marked 'S' and must traverse to the green cell marked 'G', whilst moving through yellow flag cells marked 'F'.

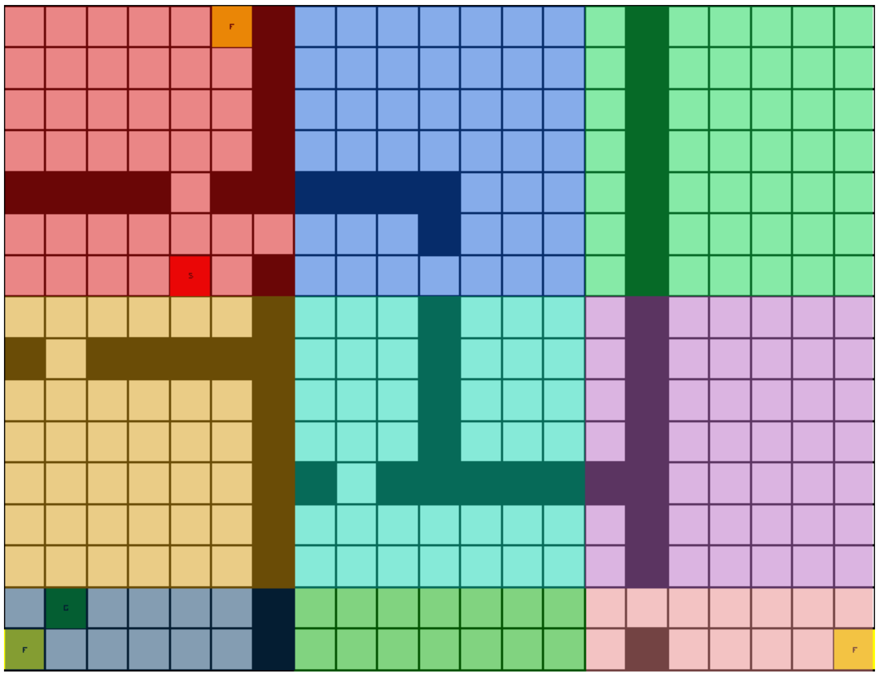


Figure 3.2: Visualisation of the partitioning of the states-pace into abstract states of size 7 × 7 for the basic Flag Collection domain

state-space can be split into a given number of abstract states of roughly equal size (accounting for dimensions where the number of states is not divisible by the desired number of abstract states) or abstract states of uniform size can be tiled along the state-space until there is no further room for tiles, if there is any space left over (due to, again, the divisibility of the state-space size and number of abstract states) then we fill in the remaining space as one abstract state. Neither of these approaches are perfect. Here we went with the latter when necessary to try and keep most of the abstract states as uniform as possible.

We give a visualisation of the abstract state partition for our basic flag collection domain in Figure 3.2. This figure uses abstract states of size 7 × 7. That is, each abstract state consists of 7 ground states in each direction until there is no further space. In continuous settings this issue does not appear as the state dimensions can always be partitioned equally due to each state dimension being continuously divisible.

The state abstraction function $Z : S_{\mathcal{M}} \to S_{\mathcal{A}}$ is then simply a map from ground states to the abstract state in which they lie. This is easy to compute — particularly for this domain, we simply need to store the ground-state values for the bounding box of each abstract state and can compare against the desired ground-state. This operation will take time proportional to $\mathcal{O}(|S_{\mathcal{A}}|)$ and will require memory proportional to $\mathcal{O}(|S_{\mathcal{A}}|)$. Equally, if space is more abundant than time, each state could also be assigned its abstract state as a label, requiring $\mathcal{O}(1)$ time and $\mathcal{O}(|S_{\mathcal{M}}|)$ space instead.

We desire that abstract actions correspond to a change in abstract states. Therefore a pass is performed over the state-space, checking which abstract states are adjacent and are not obstructed by walls. $A_{\mathcal{A}}$ is then created as a function mapping an abstract state to a set of adjacent abstract states that are not obstructed. The transition function $P_{\mathcal{A}}$ is built very simply. It is a deterministic map from state-action-state tuples (s, a, s') denoting 1 if the abstract states $\bar{s}$ and $\bar{s}'$ are adjacent and $\bar{a}$ is the same action that achieves this (with this action usually being denoted as $\bar{s}'$ also). That is:

$$P_{\mathcal{A}}(\bar{s}, \bar{a}, \bar{s}') = \begin{cases} 1, \text{if } (\bar{s}, \bar{s}') \text{ adjacent and } \bar{a} = \bar{s}' \\ 0, otherwise \end{cases}$$

The abstract action space and abstract transition function are intertwined here. Despite being found in an “automatic” manner, this is only possible due to the simplistic nature of the environment. Domain knowledge was exploited — knowledge that the ground agent can only move in cardinal directions one cell at a time. This lets us easily find adjacent states in the ground transition function. From there, identifying adjacent abstract states is easy. However, it is important to note that this is not the case for vast numbers of environments, particularly when state dimensions do not correspond to physical space. Therefore, we treat the abstract action space and abstract transition function here as domain knowledge. In later chapters we work to remove this domain knowledge from our approach.

Finally, the abstract reward function $R_{\mathcal{A}}$ is fairly sparse, it gives the agent a reward upon entering the abstract state containing the terminal state of $1000 \times$ No. of Flags collected and a smaller reward of 100 for entering the same abstract state as a flag. In all other cases the abstract reward given is 0. Again, we must note that the abstract reward function comprises domain knowledge. We must know a priori which abstract states contain flags or the terminal state.

Combining these yields the desired AMDP $\mathcal{A} = (S_{\mathcal{A}}, A_{\mathcal{A}}, R_{\mathcal{A}}, P_{\mathcal{A}})$ as well as state abstraction function Z_S (which once we have constructed the AMDP is the only part of Z we really need). This is a fully-fledged AMDP that can be treated as if it were an MDP — in fact it is. This AMDP represents an abstract instantiation of the original environment. Much of the complexity has been removed while retaining the core elements of the task.

As a fleshed out example, let's consider the AMDP that would be created for the MDP in Figure 3.1a using the state-space partitioning from Figure 3.2. In the original MDP, the state-space consists of a tuple (x, y, a, b, c), where x and y corresponds to the agent's position on the grid and a,b and c are Boolean variables denoting whether the associated flag has been collected. There is additionally a special abstract goal state to denote the agent has completed the task. The partition defines the state-space abstraction function Z_S which splits the MDP across each dimension, giving an abstract state set $S_{\mathcal{A}} = \cup \bar{s}$ for each coloured region in Figure 3.2. Abstract states therefore take a similar form to that of the MDP $(\bar{x}, \bar{y}, \bar{a}, \bar{b}, \bar{c})$, where $\bar{x}$ and $\bar{y}$ now correspond to the enumerated position of the abstract state and $\bar{a}$, $\bar{b}$ and $\bar{c}$ correspond to the flags in the same manner as before. The abstract action function $A_{\mathcal{A}}$ maps abstract states to their actions moving the agent to adjacent abstract states. There is also an abstract action to collect any available flags as well as enter the goal state (available if the agent is in the correct abstract state). The transition function $P_{\mathcal{A}}$ enables the movement deterministically, as well as allowing the pick up of flags. For instance, if the abstract agent is in abstract state $(\bar{x}, \bar{y}, \bar{a}, \bar{b}, \bar{c})$ and picks up the flag corresponding to $\bar{b}$, the agent moves into abstract state $(\bar{x}, \bar{y}, \bar{a}, 1, \bar{c})$. Finally the abstract reward function is 0 except when a flag is collected (giving 100 reward) or when the agent completes the task (giving $1000\times$ the number of flags collected).

To construct this AMDP we utilised a lot of domain knowledge. In some form or another, knowledge of the environment was used to create each component of the AMDP except for the abstract state space $S_{\mathcal{A}}$ (where even then, knowledge of how many partitions to divide $S_{\mathcal{M}}$ into has been used). However this is not an issue for now, the focus of this chapter is to show that AMDPs constructed from uniform partitions can be beneficial to learning and comparable to hand-crafted abstract state-spaces. In future chapters as much of this domain knowledge will be removed as is feasible.

3.3 Utilising the AMDP

Value iteration can be used to solve the constructed AMDP. This produces a mapping V from abstract states to values, representing the expected abstract return received by an abstract agent selecting abstract actions greedily based on value.

Utilising potential-based reward shaping (PBRS), after each ground transition $s \to s'$, we check for a change in abstract states. After each such abstract transition we then give the agent an additional reward of $F(s, a, s') = \omega(\gamma V(Z_S(s)) - V(Z_S(s')))$ for a scaling constant ω.

This is identical to the AMDP-based reward shaping we saw in Section 2.6.1. As with previous work utilising PBRS, we hope that the additional rewards will shape the agent's behaviour to yield more potent policies more quickly.

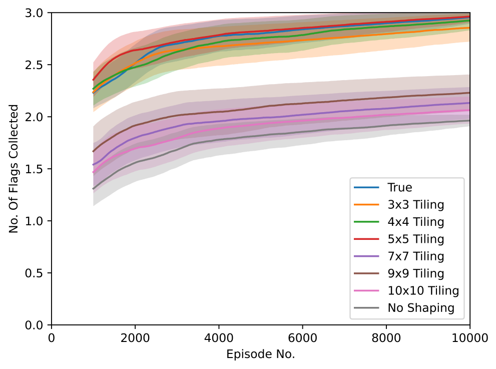
3.4 Results

We now look at the results of applying the proposed PBRS method based on uniformly partitioned AMDPs on the variations of the Flag Collection domain as described in Section 3.1. The reward graphs are presented in Figure 3.3. Each variation of the Flag Collection environment was completed by the agent ten times. The results shown are the mean values of each attempt. Each agent is using the standard $Q(\lambda)$ algorithm as its underlying learning algorithm. The agents utilising reward shaping are $Q(\lambda)$ simply augmented to receive the additional reward calculated by PBRS. The uniform tiling was performed using tiles of varying sizes depending on the variation size — but typically using tiles from size 3×3 up to 10×10. The agent also completed each variation using a 'True' hand-labelled abstraction which was labelled by the author based on the bottlenecks between rooms, as well as comparing against an agent that uses no reward shaping —– just the standard $Q(\lambda)$ algorithm.

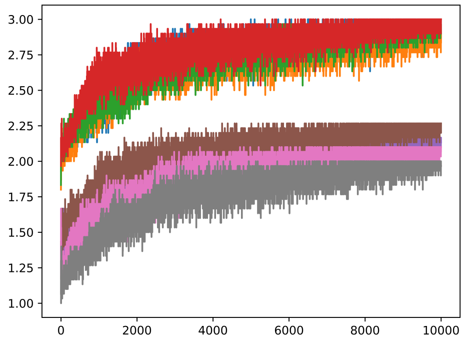
Each algorithm and environment used the same parameters, with the exception of the number of episodes the environment was run for (but this was still kept constant for each agent type) — this was tuned for each environment to show the differences in convergence times easier to visualise. The parameters used were $\alpha = 0.1$, $\lambda = 0.9$, $\gamma = 0.99$, $\omega = 20$, and $\epsilon = 0.5$ initially and linearly decaying to 0.05. These parameters were all found empirically to give strong results over a wide range of environment variations and abstractions.

In Figure 3.3 there is a clear trend showing that many of the uniform tilings actually can compete with hand-labelled examples over a range of domain variations, and sometimes even converge more quickly. This is actually quite intuitive, as using smaller tilings over a state-space will typically yield more transitions between abstract states. This means that there are more states in which reward shaping is fully utilised, as $\phi(s) = \phi(s')$ if both $s, s' \in t$ for abstract state t. The extrinsic reward is then equal to $\gamma\phi(s') - \phi(s) = (\gamma - 1)\phi(s)$. In the case of the these experiments, the extrinsic reward then becomes $-0.01 \times \phi(s)$. The point here is that having more state transitions with a high extrinsic reward will give the agent more guidance. Therefore, even though the hand-labelled abstractions have "better" domain knowledge, the uniformly shaped abstractions can perform better in some cases.

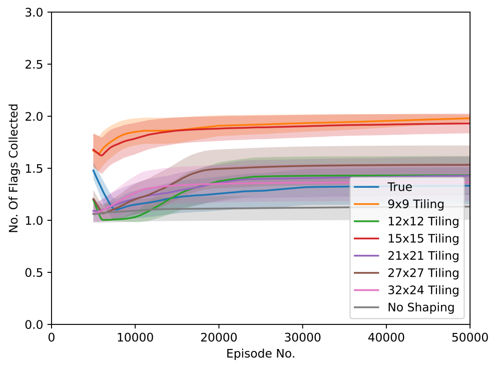
Whilst using smaller tilings may improve the number of episodes required to converge to a near-optimal policy, solving the AMDP created becomes harder the more abstract states it has. In Figure 3.4 the time taken to solve each abstraction and simulate 10000 episodes is shown — that is, the total time a single run of the experiment took to perform. This figure shows the vast increase in time required for smaller tilings. Many of the smaller tilings take much longer than the hand labelled abstraction. However, the 5×5 tiling actually takes a time that is not egregiously dissimilar to that taken by the hand



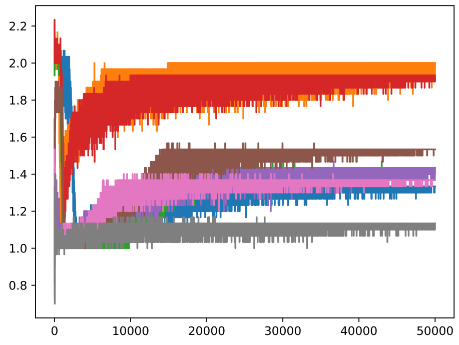
(a) Basic Smoothed



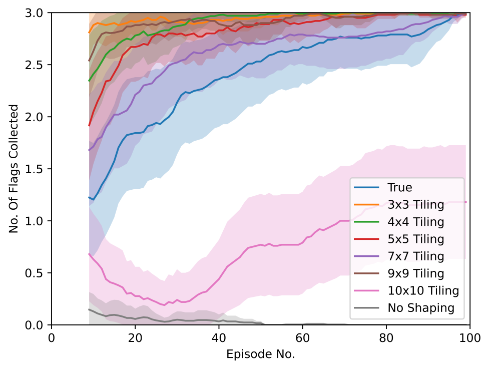
(b) Basic Unsmoothed



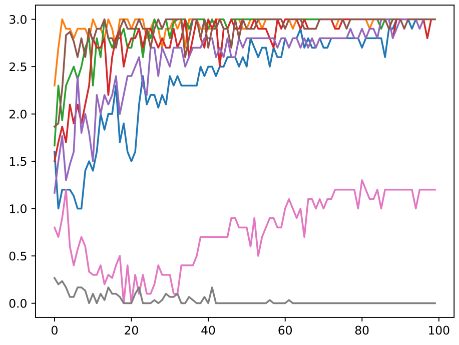
(c) Big Smoothed



(d) Big Unsmoothed Results

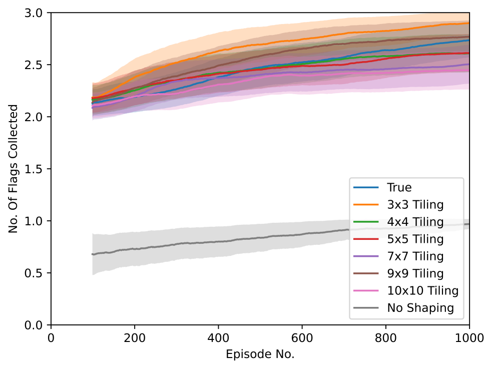


(e) Open Space Smoothed

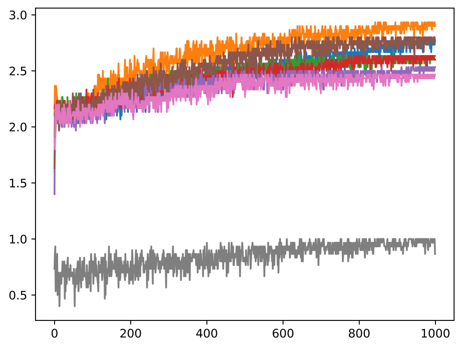


(f) Open Space Unsmoothed

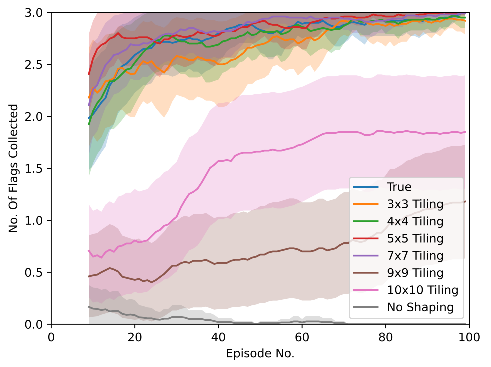
Figure 3.3: Reward graphs for each Flag Collection variation using different sized abstractions. Results shown on the left side are smoothed with a moving average of reward of the previous $NumberOfEpisodes/10$ episodes, with shaded regions denoting a confidence interval of 95%. On the right are the unsmoothed results.



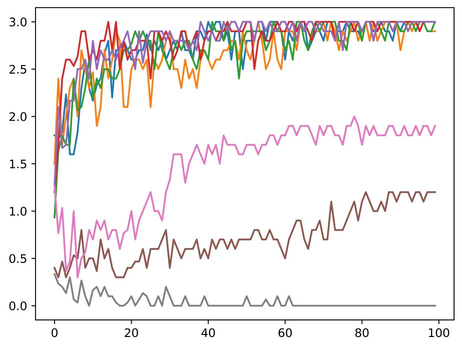
(g) Long Strips Smoothed



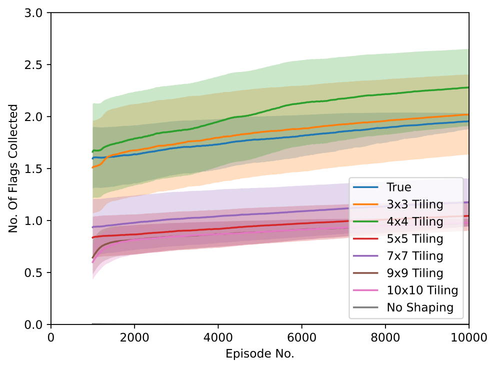
(h) Long Strips Unsmoothed



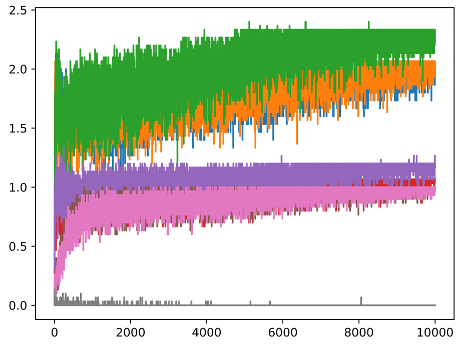
(i) High Connectivity Smoothed



(j) High Connectivity Unsmoothed



(k) Low Connectivity Smoothed



(l) Low Connectivity Unsmoothed

Figure 3.3: Reward graphs for each Flag Collection variation using different sized abstractions. Results shown on the left side are smoothed with a moving average of reward of the previous $NumberOfEpisodes/10$ episodes, with shaded regions denoting a confidence interval of 95%. On the right are the unsmoothed results.

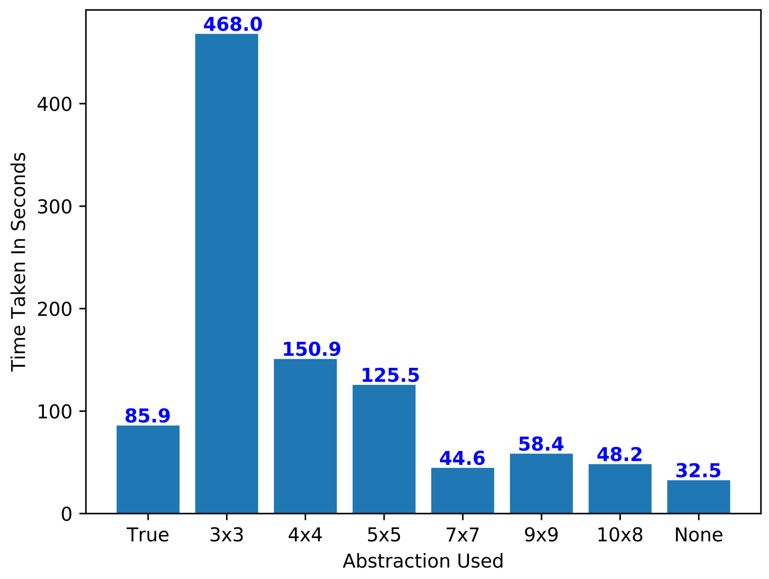


Figure 3.4: Time taken to solve each abstraction and simulate 10,000 episodes for the basic Flag Collection domain

labelled abstraction. The 5×5 tiling and hand labelled abstraction both perform similarly. This is a very positive result due to the difference in knowledge given to each agent and abstraction. The larger tilings take much less time, and although they perform worse than the hand labelled abstraction, they do not perform particularly badly. In fact, every tiling outperformed the standard $Q(\lambda)$ algorithm.

Throughout the rest of the results in Figure 3.3, this trend continues. It is worth noting that many of these variations of the flag collection problem required markedly fewer episodes in order for most of the abstractions to converge upon the optimal policy. A lot of the uniform abstractions converge more quickly than the hand-labelled abstraction.

The agent in the "Big" variation of the environment (Figure 3.3c) appears to get stuck in a local optimum and the agent's policy doesn't converge on a final flag. This is likely to do with the initial parameters, the size of the domain and also the accuracy required to achieve reward. The abstract states are also much larger in comparison to individual states — meaning that navigating to an abstract state is not as useful. It is worth noting that this largeness also applies to the hand-crafted abstraction.

Overall these results are broadly positive. They indicate that an easily created uniform partition of the appropriate size can compete with a hand-crafted abstract state-space when used for AMDP-based reward shaping. Even in the cases when they perform noticeably worse than a hand-crafted abstract state-space, shaping from any uniformly partitioned state-space is better than no shaping at all.

3.5 Issues with this Approach

Whilst this approach displays the efficacy an AMDP based on a uniformly partitioned state-space can have when utilised for reward shaping, there are a number of issues with this approach. We now address the most pressing concerns.

3.5.1 Domain Knowledge

The most glaring issue is that we provided a lot of domain knowledge about the class of environments. This domain knowledge includes the transition function, reward function, notions of the abstract actions, as well as the abstract state in which the flags reside. None of these constitute "trivial" domain knowledge, nor can they easily be applied to a vast number of other domains. For example, in a lot of domains — particularly those not based around physical, discrete navigation — the adjacency of two states does not necessarily mean an agent can transition between the two. This applies equally to abstract states. Our encoding of the transition function utilised our knowledge that for this particular class of environments adjacency through a door is enough to ensure transitionability. The AMDP's reward function implicitly encodes the approximate location of the flag, giving the abstract state in which it lies. This too is considerable domain knowledge, as the vanilla agent that we are comparing against does not have this knowledge.

This large amount of domain knowledge required is the primary reason that we cannot really claim that the current implementation of our method is actually improving on the performance of a vanilla $Q(\lambda)$ agent. However, what we can show, is that in many cases our agents performed comparably to a hand-labelled partitioning of the environment which was given the same knowledge. Even though the intrinsic abstract state structure of the domain has been lost, applying the same knowledge can give comparable or better results. If this utilised knowledge can be learnt or acquired through interaction with the environment, then we could potentially expect a tangible improvement to the vanilla agent. The crux of the matter is that we want as little external knowledge to be utilised as possible.

3.5.2 Intuitive Objections

Due to the uncertain nature of the walls within the environment and a uniform size of abstract state, it is expected that at some point in the partitioning, an abstract state will straddle one or more walls. By this, we mean that a wall will divide an abstract state and make it impossible to manoeuvre from one part of the abstract state to another. This has the potential to cause an issue. If we consider Figure 3.5, we can see that from the AMDP's perspective, the optimal path is from abstract state A to B to C. This route, of course, is not viable, since B is divided by an impassable wall. The agent has to retrieve the flag from B, then return to A and take the longer route round to C. The issue causing this to arise is that abstract state adjacency is not, in some sense, transitive. There are possible transitions from A to B and from B to C, but it is impossible to move from A to C via B. The uniformity of the state-space partitioning did not take into account the ground-level dynamics of the abstract states it created. A hand-labelled abstract state-space, on the other hand, would likely not have done this.

More complex versions of this scenario played out in our experiments with many abstract states straddling walls in this manner. This never appeared to be an issue — the AMDP is not in control of the agent and can only reward or penalise the agent for abstract transitions. If the agent does move from A to B, it is indeed rewarded and picks up the flag, however the AMDP can only encourage the agent to move from B to C if the agent actually does it. Since this is impossible from this side of B, the agent cannot be rewarded for this. The AMDP can penalise the agent for moving back to A — since the AMDP views this as a step backward in the path to the goal. However, since this is really the only option for the agent other than staying in B forever, the agent eventually overcomes its desire to avoid A. As the episodes progress, the larger overall reward of returning through A and eventually moving to the goal propagates backwards to overcome this aversion to moving back through A entirely. This scenario may cause some slowdown in the learning process compared to if this problem was not present. However, in all of the performed experiments, any potential slowdown caused by this issue was outweighed by the overall benefits of reward shaping and were still comparable to hand-labelled abstract states where this scenario does not occur. Further, this scenario highlights the attractive property of policy invariance that PBRS provides. Even though the shaping function discourages what could be an optimal solution, the ground agent will not be prevented from achieving this solution — it might just take longer (however for our agent here it does not).

3.5.3 Time Taken To Solve the AMDP

Comparing our uniform partitions of the state-space to the hand-labelled abstract states comes with the caveat that as the abstract state-space resolution increases, so too does the time taken to solve the AMDP. Again, we can look at Figure 3.4 to see an example of the increase in time taken to solve and utilise the higher resolution AMDPs. This increase in time would also happen for a higher resolution hand-labelled abstract state-space, but generally the labeller can be more efficient with space (not requiring uniform partitions) and can make better use of the domain's natural structure. However we must also remember the cost of hand-labelling such an AMDP's state-space in the first place. For the easy example of our Flag Collection tasks, envisioning the abstract state spaces as "rooms" was relatively simple, but still took time to actually encode into the environment — more time than was taken by Value Iteration to solve the highest resolution uniformly partitioned AMDP.

Overall, for many of the environment variations, one of the middling sized abstract state space resolutions performed as well as the hand-labelled state-space and took a comparable time to solve.

3.6 Conclusion

The goal of this feasibility study has been to discover how effectively AMDPs with abstract states formed by uniform partitions of the original state-space

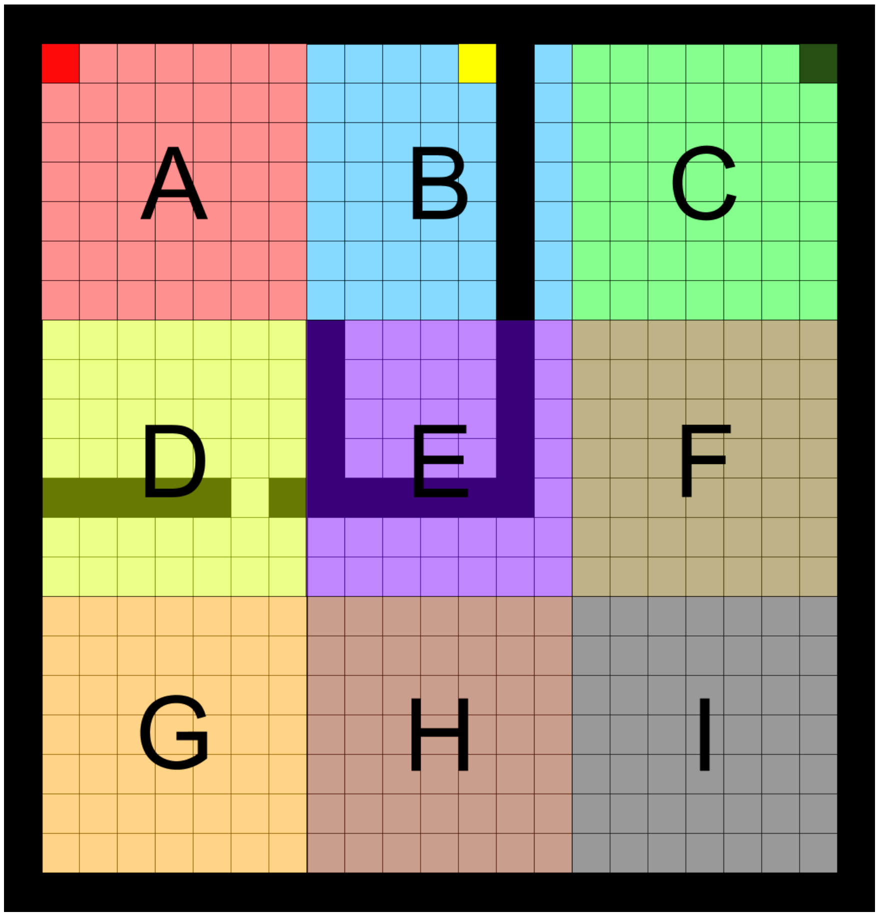


Figure 3.5: Possible scenario in which the reward shaping may slow down the learning process due to the AMDP's granularity.

can be used for potential-based reward shaping. We are especially interested in comparing the efficacy of these AMDPs to AMDPs utilising hand-crafted abstract state-spaces. We saw that, in general, the results for the uniformly partitioned AMDPs were comparable to the AMDPs with hand-crafted state-spaces — in some cases even outperforming them. This is significant as the cost of experts encoding domain knowledge into useful AMDPs is often prohibitive; removing this cost with a simple, intuitive approach that gives much of the same benefits could make AMDP-based reward shaping more applicable to a wider variety of domains. This came with the caveat that the very high resolution uniformly partitioned AMDPs can take much longer to solve with Value Iteration — but this is balanced by the time taken to encode the expert's abstract states, and in many cases, solving the uniformly partitioned AMDP is still less costly. The method outlined in this chapter would nevertheless be largely suitable for low-dimensional environments in which the dynamics are relatively simple and based on already understood heuristics — such as the requirement to move to directly adjacent states through physical space. Additionally, some notion of the location of the abstract rewards or goal should be understood beforehand to craft the AMDP. Despite these requirements, there are many types of scenarios in which they are met — for instance guiding a robot towards a tag emitting some sort of signal that the robot has access to. In the coming chapters we wish to expand the range of domains in which this approach can be applied by removing these requirements.

Now that we have seen the surprising efficacy and potential of uniform abstract state-space AMDPs, we wish to remove the onerous domain knowledge that we utilised to create the rest of the AMDP. Moving forwards, we will look at exploration methods for gleaning much of this knowledge from pure interaction with the environment. Expanding the approach to incorporate this in order to remove this domain knowledge reliance is the focus of the next chapter.

Chapter 4

Uniform State Abstraction For Reinforcement Learning

This chapter introduces an extension to the previously detailed MultiGrid Reinforcement Learning (MRL) [29]. The new method — which we will often refer to as Uniform Partition State Abstraction (UPSA) — is an improvement to MRL within the Deep Learning scenario. We show this empirically over three domains with continuous state-spaces. This extension is also an improvement over the Uniformly Partitioned State-spaces of the previous chapter and requires less domain knowledge to construct the required AMDP.

The exact differences between the two lie in when and how the abstraction is created and solved. UPSA explicitly models the abstraction and solving the abstract problem as soon as possible to produce a static reward shaping function. MRL on the other hand only implicitly models the abstraction process and is continually changing the shaping function.

Further, throughout this chapter we examine both methods in detail and in Section 4.4.1 elucidate the differences between UPSA and MRL, pointing out justifications for why UPSA is a more desirable approach for Deep Learning specifically.

4.1 Uniform Partition State Abstraction

Utilising UPSA for reward shaping proceeds broadly in the following steps. Initially, a set of abstract states is constructed simply by uniformly partitioning the state-space. Secondly, an abstract model of the domain is built up as an

AMDP by observing agent interaction with the environment as viewed through an abstract lens. In this exploration stage, an exploration policy is followed in order to maximise the diversity of states visited and actions performed. Once the allotted time for exploration has elapsed, the AMDP is constructed according to the interactions observed. Special attention is given to the abstract reward function, which we discuss thoroughly in section 4.2.3. This AMDP is then easily solved using Value Iteration; yielding abstract state values. Finally, the agent can begin interacting with the environment with the intent to maximise reward. A Potential-Based reward shaping function using a potential function based on the values of abstract states is used to shape the reward and guide the agent to more fruitious behaviour.

UPSA is intended as a conceptually simple, yet surprisingly effective augmentation for existing RL algorithms such as DQN. In the proceeding sections, we give a much more detailed breakdown of each step in the method, along with justifications and empirical results.

4.2 Constructing the AMDP

We now give a detailed view of the construction process for the AMDP. Throughout this section there are references to two levels of learning occurring. Firstly, there is the agent updating its Q-value estimates for state-action pairs in the environment — we refer to this as ground learning. The other type of learning is abstract learning, where the AMDP dynamics are learned and abstract values are computed. This nomenclature is simply how we distinguish between the different learning types.

While each of the coming sections is part of the sequential process of constructing the AMDP, we often fully construct aspects of the AMDP early on in the process, some of which are used to construct later parts. This will be detailed where it occurs and how it relates to the abstraction process overall.

4.2.1 Partitioning the State-space

The first step of the method is to partition the state-space uniformly in each dimension. The only hyper-parameter to consider is the number of partitions per dimension. More partitions per dimension will allow a greater level of guidance for the agent over that dimension. The downside to using more partitions is an increased computational cost of solving the AMDP later on. For continuous state dimensions that have no upper or lower bound on the values we must also pick suitable upper and lower limits. Any encountered values outside of this range are set to partitions capturing any such values.

This allows us to easily define a state abstraction function Z_S which maps a continuous ground state s to the partition in which it lies $Z_S(s)$. This function is quick to compute for a given s. Since a different number of partitions can be used for each dimension, this further allows more important state dimensions to be partitioned more finely, granting a greater level of guidance to the agent without increasing the granularity in less important dimensions.

Visually this partitioning process is illustrated in Figure 4.1. This figure has been adapted from a similar figure in [17] but modified to map ground states to abstract states rather than an element of a STRIPS [21] plan. This figure shows the intuition behind identifying abstract states and abstract transitions for continuous ground state-spaces and discrete abstract state-spaces.

The notion of identifying abstract transitions in this manner is useful and we will refer to this as viewing the ground transition through an "abstract lens". This view through an abstract lens allows us to construct the transition function — and thus the AMDP as a whole — without any knowledge of abstract transitions a priori.

Sometimes we may want to utilise the ground values represented by abstract states — particularly when finding criteria for abstract goals. To do this, we refer to an abstract state $Z_S(s)$ as having the values of ground state $s*$ where $s*$ is the central ground state of $Z_S(s)$.

At the end of this section, we have both our set of abstract states and the corresponding abstraction function. The set of abstract states, while almost trivial to produce is actually a key element in constructing the rest of the AMDP. The abstraction function acts as a "lens", allowing us to view agent interactions from an abstract perspective — without this, the abstract dynamics could not be constructed and abstract values could not be assigned.

4.2.2 Exploration

Since there is no prior knowledge of the dynamics of the environment the agent interacts with, the AMDP cannot be fully constructed until the agent has explored the environment sufficiently (i.e., we need to estimate a Z_P and Z_R). To do this, we allow the agent to follow an exploration policy, observing and recording the abstract transitions that are taking place.

In order to allow shaping to begin sooner it is desirable to minimise the time spent within the exploration phase. To achieve this, we do not perform any ground learning during the exploration phase. We simply focus on building up the AMDP by observing the ground agent's transitions through an abstract lens.

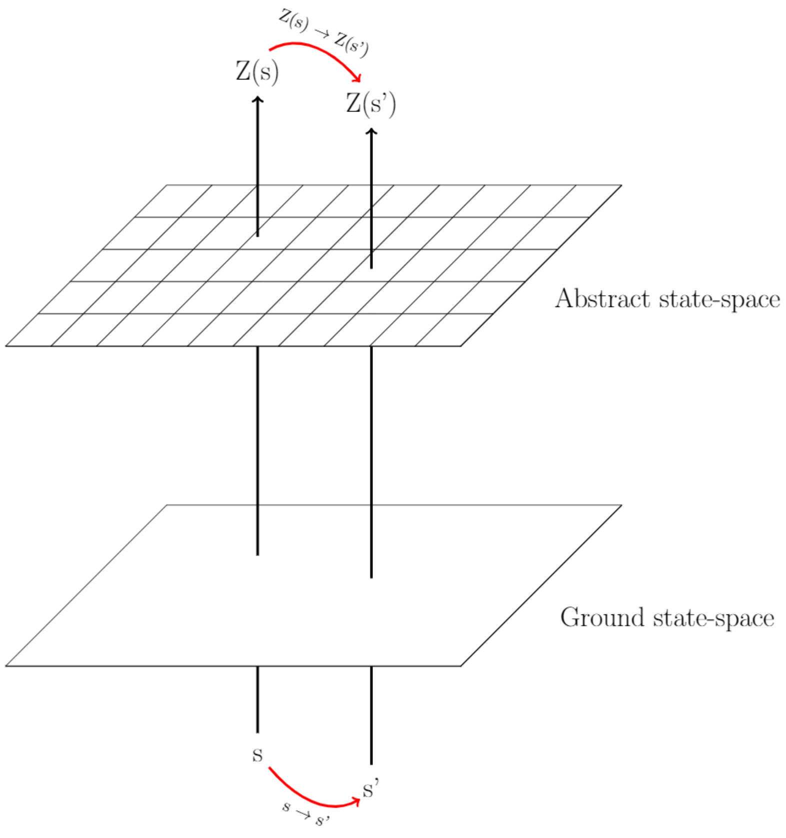


Figure 4.1: Visualisation of mapping from ground-transitions to abstract-transitions.

During the exploration phase, the agent will move between ground states and consequently abstract states. For exploration we also utilise a different level of states which we refer to as exploration states.

A dynamic exploration policy is given for the agent to follow. The exploration policy simply has the agent pick the action that has been selected the least when in its current abstract state in an ϵ-greedy fashion. This encourages the agent to explore previously unseen states earlier. For this exploration method, it turned out to be advantageous to use so-called exploration states instead of the AMDP states. By exploration states, all we mean are an even coarser uniform partitioning of the state-space (than the AMDP's abstract states). These exploration states are formed identically to how AMDP states were created — they really are just a coarser partitioning. The reason for using these very coarse exploration states with our exploration policy is to encourage the agent to explore on a macro-scale. Despite using these exploration states, it is crucial to remember that it is the AMDP state-transitions that are recorded, not the exploration state-transitions. This allows us to build up a comprehensive model the transitions available within the AMDP.

At the end of this section, we now have our approach for constructing the abstract transition function — we add each of the observed abstract transitions as available in the AMDP.

4.2.3 Abstract Reward

Whilst the transition function detailing the dynamics of the environment viewed through abstract states can easily be approximated through observation, defining a useful reward function has a few more possibilities and challenges.

For an abstract transition $\bar{s} \to \bar{s}'$, one could opt to estimate the average observed cumulative reward over that transition period. However this may not be ideal for environments where there are many ways of achieving the abstract transition, with large variances in the cumulative ground reward observed. As a consequence, the abstraction is not able to distinguish between poor sequences of ground actions and a potentially optimal sequence, if the poor sequences are more frequent. This was also observed empirically, where using observed cumulative reward values worked well for environments such as Mountain Car, but performed poorly in environments where the ground reward can vary more, e.g. Puddle World.

A simpler alternative that was found to work well empirically was to set the abstract reward $R_{\mathcal{A}}$ to -1 for every transition. Combined with the set of terminal abstract goal states, the value of an abstract state becomes the transition distance between the current abstract state and a goal. Despite our abstract states being the same size, this does not ensure that each abstract transition takes the same number of ground transitions — particularly when some state-spaces incorporate agent velocity. This means that in some cases, where the agent moves through abstract states very slowly, that the AMDP will be rewarding non-optimal behaviour. Empirically, this doesn't seem to cause much of an issue — the AMDP's extra reward and guidance being helpful most of the time is enough to improve performance. This supposed issue is exacerbated in the Catcher environment (Section 4.6.3), where one of the state dimensions is velocity — determining exactly how quickly the agent can move through the position-dimension abstract state. Despite this, we still see notable improvements to the convergence speed in Catcher.

4.2.4 Abstract Goals

UPSA requires that one or more abstract states are selected as goal states. When the value function for the AMDP is being computed, the values of abstract goal states are set to 0. This provides an endpoint for the step-penalty reward function. This usually does not require much additional knowledge of the domain — goals are often part of the task description or easy to describe.

Some environments require a single goal to be repeatedly achieved, for example the Catcher domain in Section 4.6.3. Our method handles this with no required alterations, repeatedly guiding the agent to the goal.

Our method can also be applied to environments which do not have conventional "goals" or domains with an infinite horizon. In such cases, the abstract goal can simply be set to "desirable" behaviour. This will then reward repeated completion of the abstract goal more-so than other behaviour.

4.2.5 Putting it all together

Now we have all of the constituent parts of the AMDP and we simply need to put it together. We have the set of abstract states $S_{\mathcal{A}}$ defined by the partitioning. We also have the set of observed transitions. We create an abstract action set $A_{\mathcal{A}}(\bar{s}) = \{\bar{s}' : \bar{s} \to \bar{s}' \text{ was observed}\}$. That is, if abstract transition $\bar{s} \to \bar{s}'$ was observed, then we give abstract state $\bar{s}$ an action $\bar{s}'$ representing the action that causes such a transition. We also need the transition probability function — for this we can use

$$P_{\mathcal{A}}(\bar{s}, \bar{s}') = \begin{cases} 1, \text{If transition } \bar{s} \to \bar{s}' \text{ was observed} \\ 0, \text{otherwise} \end{cases}$$

We can construct the AMDP to be deterministic, this is justified by the fact that we are only using the AMDP for reward shaping. If we are considering $\bar{s} \to \bar{s}'$

in the shaping stage, then even if the underlying environment is stochastic, we need not worry about how that affects this transition since we only consider transitions after they have occurred. We also include the constructed reward function $R_{\mathcal{A}}$. All of this together yields the AMDP $(S_{\mathcal{A}}, A_{\mathcal{A}}, R_{\mathcal{A}}, P_{\mathcal{A}})$.

The overall algorithm is given in Algorithm 5; here the resulting abstract transition function is deterministic as described above, and each abstract action is rewarded with a step penalty.

Algorithm 5 AMDP Generation

```
procedure AMDP GENERATION(MDP $\mathcal{M}$, Partition List $D$, Exploration Policy $\pi_E$)
    $S_{\mathcal{M}}$ = set of MDP states
    Partition $S_{\mathcal{M}}$ uniformly into $D_i$ bins for dimension $i$
    Denote these bins as elements of $\mathcal{S}_{\mathcal{A}} = (\times \mathbb{Z}_{D_i})$
    Create function $Z_S$ mapping state $s$ to bin containing $s$
    Initialise abstract transitions $P_{\mathcal{A}}(\bar{s}, \bar{a}, \bar{s}') := 0$
    Initialise abstract Rewards $R_{\mathcal{A}}(\bar{s}, \bar{a}, \bar{s}') := 0$

    for each exploration episode do
        $s :=$ Initial State
        while episode not complete do
            Select action $a$ according to $\pi_E$
            Perform action $a$ and observe $s', r$
            if $Z_S(s) \neq Z_S(s')$ then
                $P_{\mathcal{A}}(Z_S(s), Z_S(s'), Z_S(s')) = 1$
                $R_{\mathcal{A}}(Z_S(s), Z_S(s'), Z_S(s')) = -1$
    return $(S_{\mathcal{A}}, A_{\mathcal{A}}, R_{\mathcal{A}}, P_{\mathcal{A}})$
```

4.3 Utilising The AMDP

Now that the appropriate AMDP is constructed, dynamic programming methods such as Value Iteration can be used to compute a value $V(\bar{s})$ for each abstract state $\bar{s}$. We now can use our value function to augment an existing RL algorithm (We utilised DQN [46]) using potential-based reward shaping. Whenever the agent changes abstract state from $\bar{s} \in S_{\mathcal{A}}$ to $\bar{s}' \in S_{\mathcal{A}}$ we consider this an abstract transition $\bar{s} \rightarrow \bar{s}'$. If such an abstract transition happens from our ground-level observations — that is, we observe $s \rightarrow s'$, such that $s \in \bar{s}$, $s' \in \bar{s}'$ — the agent is given additional reward $F(\bar{s}, \bar{s}') = \omega(\gamma V(\bar{s}') - V(\bar{s}))$ for suitable scaling factor ω.

Intuitively, this shaping rewards the agent for moving towards more promising abstract states (i.e., abstract states that have a higher potential). It is worth noting that despite DQN itself having no convergence guarantees, utilising potential-based reward shaping with DQN in this manner will uphold the sacrosanctity of policy invariance guarantees that PBRS provides; the proof given in [51] still holds for deep learning agents.

The exploitation policy utilises an ϵ-greedy approach where the ϵ value is annealed over the course of the total number of episodes (For details see the hyperparameter table 4.1). It is worth mentioning that this is an entirely different ϵ than is used in the exploration phase — both the exploration and exploitation policies both employ an ϵ-greedy action selection, but they do not share the same ϵ-variable.

It is possible that during the exploration phase some abstract state $\bar{s}$ is missed. If $\bar{s}$ is then encountered in this exploitation phase, then the AMDP has no appropriate value $V(\bar{s})$. In this case, we return a value 0 and then resolve the AMDP with value iteration. The time to resolve the AMDP is very short due to the small size of the AMDP. In all of our test domains this did not occur even once.

4.4 MultiGrid Reinforcement Learning

Before we look at the results yielded by our uniform state abstraction approach we should overview MultiGrid Reinforcement Learning (MRL) [29] — the method our result extends and improves upon.

In MRL, the same partitioning method is used as UPSA. However, in MRL it is used twice, once to discretize the continuous state-space and produce a set of ground states and a second time to produce a set of abstract states the same way that we do. It is worth mentioning that MRL was developed before the onset of deep learning based RL methods that we see today — that is why the ground states are produced by a discretization.

MRL proceeds like tabular Sarsa, updating Q-values using the appropriate update rule. However, there is an additional reward function $reward_v(r)$ representing some form of external knowledge, usually representing as domain expert's hand-coded function encapsulating a goal or desired behaviour. This is used to update a function V using a Temporal Difference (TD) update. Finally, the potential function for shaping is $\phi(s) = V(s)$.

4.4.1 Differences

Disregarding properties that are simply a product of their time, such as ground state representation and the use of on-policy Sarsa, the chief difference between MRL and our approach is the explicit building of the model. MRL does not construct an AMDP, it simply performs TD updates on its abstract states. This difference has many consequences and is something of a double-edged sword.

The first result of this model-free approach to abstract learning is that MRL can begin shaping immediately — whereas our approach requires an exploration stage. However, at the beginning of training, shaping has the potential to hinder learning since the abstract values are initialised randomly and have not had time to converge on suitable values. On the other hand, once the exploration stage is complete, our method can quickly solve the AMDP using Value Iteration and provide a final static shaping function with very close to the exact values of the abstract states given our model.

If we try to compare the performance of MRL and our UPSA (As we do in Section 4.6), the first issue we encounter is how do we make the comparisons fair? In the original MRL paper, it was evaluated on just the Mountain Car environment, using discretized ground states. Comparing that against our DQN-based approach is unfair.DQN is able to generalise on a global level and each weight update can improve the policy over many separate states. MRL, on the other hand, uses Sarsa on a discretized ground state-space, and thus each update can only improve the policy for a small, local region of the state-space. This ultimately leads to DQN being able to converge on a near-optimal policy far faster than MRL.

To address this issue, a slight alteration is made to MRL to allow easier comparison. The ground state representation is replaced with a neural network. This is likely what the authors would have elected to do if DQN had been feasible at the time. Introducing this ground state representation, however, reveals a flaw in utilising this model-free approach. As we see in the upcoming results, MRL generally performs less well than UPSA, despite all else being the same. The mixing of two model free methods of very different sample complexities prevents shaping from being very useful. That is, DQN converges on a solution before the TD-updates have had a chance to make V into a useful shaping function. Our method does not suffer from this; in UPSA DQN does not even start learning until the abstract values are finalised by the exploration stage. However, we cannot simply opt not to use Deep Learning for the ground state representation in MRL, the improvements in convergence time and speed in these domains are too large to ignore.

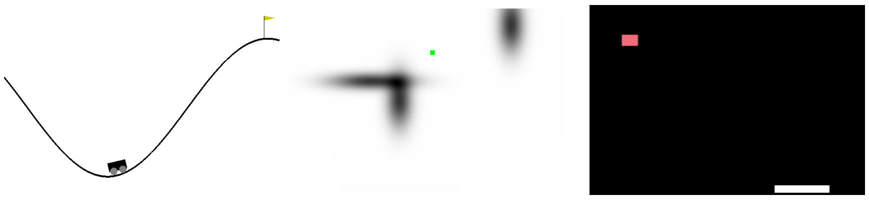
4.5 Alternative Approaches

The method proposed in this chapter (and the preceding chapter) share a similarities with function-approximation-based approaches. Both types of approach are, at their core, trying to reduce complex (and possibly continuous) state-spaces into something smaller that captures the "essence" of the states and is less costly to learn. Function approximation approaches such as Tile Coding (which we saw in Section 2.3.2) and N-tuple Neural Networks [15] typically use some method of feature extraction (for Tile Coding, it is the boolean mask of which tiles the state lies in) and then learns directly on the extracted features. For coarse feature extractions, however, this can lead to a loss of accuracy due to inability to distinguish between states that map to the same features. The proposed method from this chapter, UPSA, does not have the same disadvantage, since the abstract state — the extracted feature, if you will — is used to inform the learning process of an agent training on the ground state-space. The use of abstraction in this way, doesn't "blind" an agent to differences in ground states belonging to the same abstract state. However, we must remember that dealing with large or continuous state-space directly, such as the environments we will use to evaluate UPSA, is a relatively recent possibility due in a large part to the advancement in processing power from recent years. Many function-approximation approaches were developed due to these limitations and comparing these against a modern implementation of a deep RL algorithm isn't a particularly fair comparison.

4.6 Experiments And Results

We now examine the results attained when we compare DQN, DQN augmented with MRL and DQN augmented with UPSA. We evaluated these across three domains: Mountain Car, Catcher and Puddle World. These domains are all commonly used benchmark environments. Additionally, each domain has intuitive state transition dynamics, enabling easy conception of what a "good" abstraction may look like — it is important that our chosen environments do not have optimal abstractions for AMDP-based PBRS that just happen to line up with our proposed uniform partitions.

Figure 4.2 shows a visual depiction of each environment. We will now give a brief overview of each environment.



(a) Mountain Car (b) Puddle World (c) Catcher

Figure 4.2: The environments used to evaluate our method.

4.6.1 Mountain Car

In the Mountain Car environment, the agent is positioned somewhere inside a valley and must reach the top of the right hand side. The state-space consists of the x-position and velocity of the car. The agent has three actions, $left$, *neutral* and *right*, indicating an amount of force to apply in the designated direction. A reward of -1 is received after each action the agent makes. An episode terminates upon reaching an x-position of 0.6 or after 200 steps have elapsed. It is important to note that the car cannot reach the top of the hill simply by moving to the right — the car needs to build up momentum first by swinging back and forth. Any abstract state with a central x-position greater than 0.5 was considered an abstract goal. The implementation of Mountain Car used in our experiments was from OpenAI's gym suite [11].

4.6.2 Continuous Puddle World

In the Continuous Puddle World environment the agent is situated on a two-dimensional plane, ranging on values from $(0, 0)$ to $(1, 1)$. The agent begins in the bottom left quadrant and must reach very close to $(1, 1)$. There are five actions available to the agent: the agent may move in any cardinal directions (by a randomly determined, but bounded amount), as well as standing still. There are puddles occupying certain areas of the plane (shown in the depiction). The agent receives a reward of -1 for each step, as well as additional negative reward based on how deep into the puddle the agent is. Ideally we want the agent to move to the top right corner receiving as large a reward as possible. An episode terminates on reaching (close to) $(1, 1)$ or if 250 steps elapse. The optimal strategy is therefore to move to the top right via the shortest route while avoiding the puddles. Any abstract state with both an x and y value of greater than 0.9 was considered an abstract goal. For our experiments we used the Gym-Puddle implementation of Puddle World [19].

4.6.3 Catcher

In the catcher game, the agent embodies a one dimensional, horizontal line. Small squares fall from above the agent, perpendicular to the agent's axis of movement. This agent has three actions, $left$, *neutral* and *right* which moves the agent in the corresponding direction. The agent's goal is to move itself to intercept the falling square. The state-space consists of the agent's x position and velocity, as well as the square's x and y position. For each square the agent intercepts, it receives a reward of 1. For each square that it misses it receives a reward of -1. After 3 misses in total, the episode ends. The episode also terminates after 500 steps in order to prevent episodes becoming inordinately long as the agent improves. This allows the collection of approximately 15 balls. An abstract state is considered an abstract goal if the x positions of the paddle and square are within ten pixels of each other and the velocity of the paddle is less than three pixels per step. Our experiments utilised an implementation of catcher from the PyGame Learning Environment suite of domains [71].

4.6.4 Experimental Detail

The hyper-parameters for each agent and environments are in table 4.1. The shared hyper-parameters were chosen empirically in order to optimise the performance of the baseline DQN agent. In order to achieve this, many variations of hyper-parameters were tested manually to identify the highest performing policy. The hyper-parameters that were only used by our method were chosen empirically (In the same way as above) in order to maximise the performance against time.The total number of states in each AMDP is given by the product of the partition sizes for each state dimension. These sizes can be found in Table 4.1 (Abs. Size). The total number of abstract states in each environment was 2500. This ensured that each AMDP captured a sufficient level of detail of the environment whilst keeping the optimal policy quick to solve (less than ten seconds) with Value Iteration.

All of the approaches utilised the same network architecture. This is a simple feed-forward network with fully connected layers. The architecture was also the same across each environment, only different in the input size and output size for the number of actions. A visual representation of the architecture is given in Figure 4.3. The input to the network is a vector representation of the environment's state perceived by the agent — the exact size of this will depend on the environment. Each available action has a corresponding output node in the final layer, the output of an action a's node on a given input state s represents the learned value $Q(s, a)$.

For MRL in each environment, the sum of the ground rewards received during the transition between coarse aggregations was used to update the coarse value

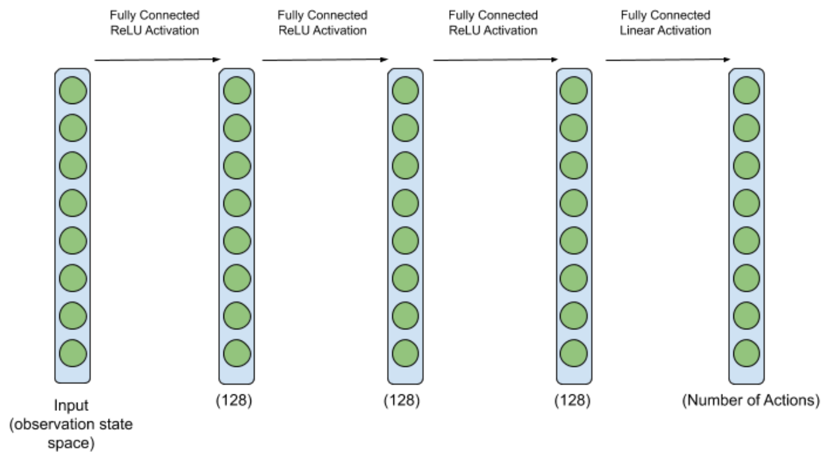

Fully Connected ReLU Activation
Fully Connected ReLU Activation
Fully Connected ReLU Activation
Fully Connected Linear Activation
Input (observation state space)
(128)
(128)
(128)
(Number of Actions)

Figure 4.3: Neural network architecture for each of the agents utilised in this chapter. The architecture is shared across environments also, changing only the number of input dimensions and output actions.

function, as defined in the original MRL paper.

The abstract reward function used for Mountain Car was the sum of received ground rewards during the abstract transition. This was chosen to enable an easier comparison with MRL (Mountain Car was the only domain used for evaluation in the original MRL paper [29]). For both Puddle World and Catcher we used an abstract reward function of -1 for each abstract transition as this was found to perform better for UPSA.

We show our empirical results for each domain. In each of the below results, the mean reward of each agent is plotted, with confidence intervals of 95% shaded. Our augmented agent is shown in grey against the unaugmented DQN agent in blue and DQN augmented with MRL in green. Since UPSA initially follows an exploration policy to construct the AMDP - it uses more episodes than the others. For a fair comparison, we opt to compare the agents by plotting reward against elapsed time — including the exploration phase. This also fairly accounts for any extra computation our method uses to solve the AMDP or compute the extrinsic reward. The end of the exploration phase is indicated by the vertical dotted black line. We ran the three agents for a set number of episodes, meaning the agents varied in the amount of time taken to complete the task. Since we are more interested in comparing performance against time, the shortest time taken to complete all of the episodes was taken as a benchmark and the other agents had their results truncated to that time-step. Comparing the agents episodically directly is somewhat misleading, this is because our method completes a lot of episodes very quickly during the exploration phase. However

Parameter	Mountain Car	Puddle World	Catcher
α	$1e-3$	$5e-4$	$1e-5$
γ	0.995	0.99	0.95
τ	$1e-2$	$1e-2$	$1e-2$
ω	1	1	1
ϵ	$0.1 \rightarrow 0.01$	$0.2 \rightarrow 0.05$	$0.1 \rightarrow 0.01$
Abs. Size	$(50, 50)$	$(50, 50)$	$(20, 10, 20, 10)$
Exp. Size	$(5, 5)$	$(5, 5)$	$(10, 5, 10, 5)$
Episodes	500	1000	1000
Exp. Episodes	500	1000	500
Action Rep.	64	64	16

Table 4.1: Hyper-parameters used for experiments

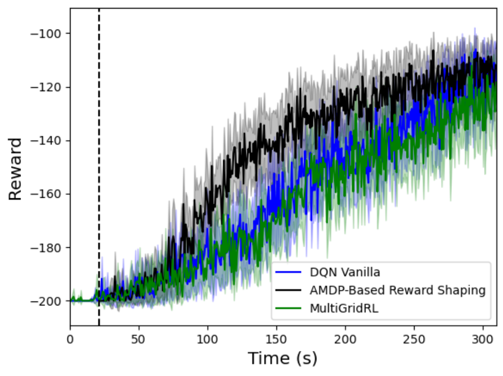
if we offset for the exploration episodes we see similar results to those shown for time.

In each domain we can see that our approach beats both other approaches with a significant margin. The exploration phase takes very little time and gives a large performance boost very quickly. MRL is more of a mixed-bag, improving the learning performance of DQN mildly in Catcher and Puddle World, while somewhat hindering it Mountain Car. It is notable however that in none of the environments did MRL out perform our approach.

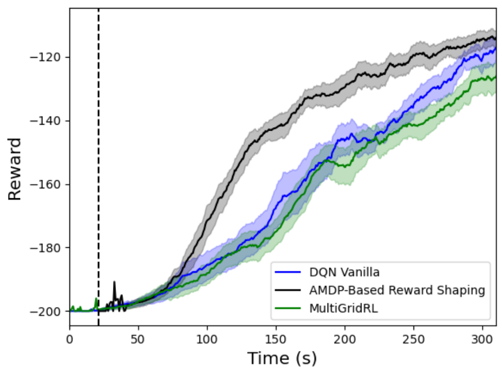
Overall these results show that MRL is ill-suited for Deep Learning due to the large amount of interactions required for learning its abstract value function. The results further show that our extension of MRL is an improvement for the Deep Learning setting, and that DQN augmented with UPSA significantly outperforms Vanilla DQN, without the need to provide much domain knowledge.

4.7 Limitations and Next Steps

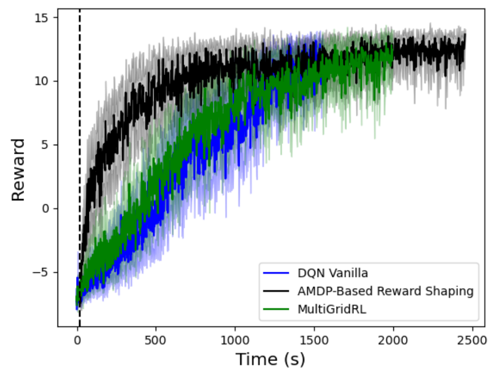
Whilst the UPSA approach has proven itself a useful tool for continuous-state-discrete-action tasks there are some innate limitations that should be noted. The simplest limitation is that of requiring a goal. Due to the fact that the abstract reward selected is typically -1 — that is, they are step-penalties — the abstract states that constitute goals are given as a (limited) form of domain knowledge. These goal specifications need not be overly precise as shown in our results, and constructing goal specifications for tasks like Mountain Car, Catcher and Puddle World is simplistic. The limitation comes from interacting with unknown or very complex environments, where desirable behaviour is not necessarily known a priori. This could potentially be solved by introducing a



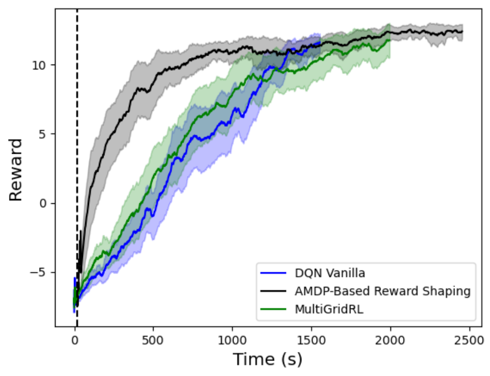
(a) Mountain Car Unsmoothed



(b) Mountain Car Smoothed



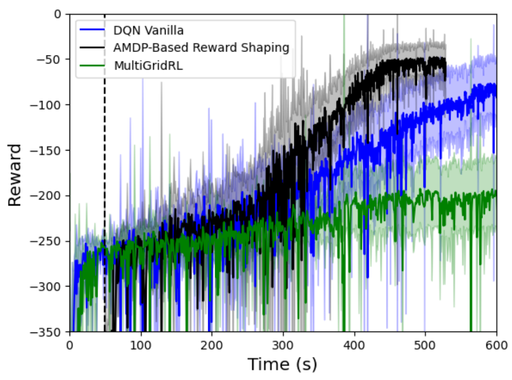
(c) Catcher Unsmoothed



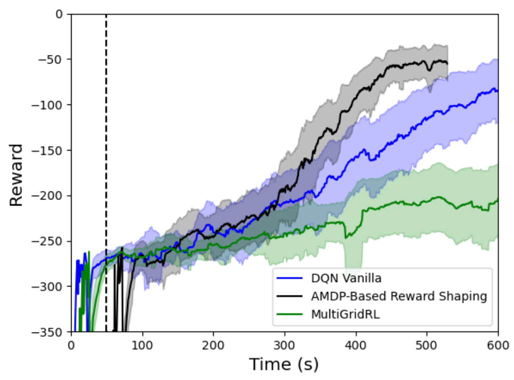
(d) Catcher Smoothed

Figure 4.4: Results Comparing DQN, MRL and Uniform State Abstraction for each environment The end of the exploration phase is denoted with a dotted vertical line. We show both the raw mean results as well as a smoothed figure for visual clarity. In the smoothed figure a moving mean window is applied with size 25 in order to smooth the curves. We plot the unsmoothed results for each episode before this point, explaining the oscillation observed in the figure.

8



(e) Puddle World Unsmoothed



(f) Puddle World Smoothed

Figure 4.4: Results Comparing DQN, MRL and Uniform State Abstraction for each environment The end of the exploration phase is denoted with a dotted vertical line. We show both the raw mean results as well as a smoothed figure for visual clarity. In the smoothed figure a moving mean window is applied with size 25 in order to smooth the curves. We plot the unsmoothed results for each episode before this point, explaining the oscillation observed in the figure.

more complex abstract reward function based upon the agent's experiences. — although further work would be needed to verify the efficacy of this approach.

The second limitation (and one that also applies to MRL) is that the number of abstract states grows exponentially with the number of state dimensions. Suppose the domain has a d-dimensional state representation, and that each dimension is split into n partitions. The total number of states is then n^d. This is — again — the Curse of Dimensionality. For environments with large state representations this poses an issue with the tractability of computing (or even storing) the values of abstract states within the AMDP. To compound this, many domains of interest within the RL community at the time of writing consist of very high-dimensional state-spaces such as controlling agents based on pixel input. One such domain is that of Atari games - where each game-play frame consists of 210×160 pixels, each consisting of three sub-pixels denoting colour. This is essentially a $210 \times 160 \times 3 = 100,800$ dimensional state representation. Even with appropriate cropping, downsampling and greyscale conversions, this is far too many dimensions to handle with the method at current.

The next steps then are to try and address these issues. The greater limitation is that of the exponential growth of the abstract state-space. Since this will be the case for any discrete partition of state-spaces —not just those that are uniformly partitioned — "natively" utilising a continuous abstract state-space would be the ideal solution. Of course, this raises questions about what a continuous abstract state-space would look like and how we would go about constructing state abstraction functions, as well as how we learn abstract models. The next chapter explores this question in detail.

4.8 Conclusion

This chapter has introduced a new method — UPSA — for speeding up Reinforcement Learning agents with very little external domain knowledge. This method employs an exploration stage to build up an abstract model of its environment, when viewed through the “abstract lens” of uniformly partitioned abstract states. The agent builds up its knowledge of which transitions are available and computes a distance (in abstract transitions) to its pre-specified goal. Value Iteration is used to produce a reward shaping function that helps guide the agent toward more fruitious behaviour. The agent can then shift to an exploitation stage, where it utilises the shaping function.

In terms of external domain knowledge, we have reduced the required knowledge down from the transition function, reward function and abstract state size required in the previous chapter to just the abstract state size and a goal for the agent. The transition function and general reward function represent the vast majority of external knowledge that is typically required to utilise reward shaping in this manner. Our newer method therefore requires far less knowledge than that of the previous chapter.

For the domains that we applied this method to, the decrease in convergence time was significant, even allowing for the time spent in the exploration phase. We compared our approach to an older approach with a similar intent — Multi-Grid Reinforcement Learning (MRL) — and found that our newer method outperforms both Vanilla DQN and DQN augmented with MRL.

Finally, we saw the need to extend our method to be more suitable to higher-dimensional tasks. In order to achieve that, we will need to break away from the discrete abstract states that we have so far used to model our abstraction of the environment. The next chapter will focus on this as well as other aspects of dealing with high-dimensional state-spaces.

Chapter 5

Latent State Abstraction

This chapter builds and improves upon the Uniform Partition State Abstraction (UPSA) approach to constructing a shaping function. The aim here is to significantly extend the range of domains for which the method can be effective. We achieve this by removing two of the assumptions made by UPSA. These assumptions are that abstract states are discrete and also that the abstractions originate from uniform partitions over the ground state-space.

The combination of discrete abstract states and the fact that these states originate from uniform partitions over the ground state-space is problematic for learning due to the aforementioned Curse of Dimensionality [6]— the number of abstract states grows exponentially with the number of dimensions in the state representation. To overcome this we introduce the idea of latent state-space based abstractions, and demonstrate how these can be used for reward shaping to the same effect as the method from the previous chapter. This chapter culminates in the introduction of a new method, which we will refer to as Latent Property State Abstraction (LPSA).

Before we can begin to introduce our method, there are a few prerequisite techniques and terminologies that need to be understood. In the following sections we will introduce these prerequisites.

5.1 High Dimensional State-Spaces

Many high dimensional state-spaces contain vast amounts of redundant information. This is seen often in “visual” domains, where instead of a concise state description, the state of the domain is represented by a visual depiction of the scenario. Even low-resolution domains such as Atari games contain a vast number of pixels to convey the current state. This yields an overly large number of state dimensions, one for each possible colour for each pixel. This often gives millions of state dimensions.

To mitigate this somewhat, convolutional layers are nearly always used when working with visual domains within RL — identifying salient features from the images and essentially reducing the number of dimensions. The reduction achieved by this method can be vast, however the convolutional layers must have their weights learned and thus are not instantly effective. The benefit from using convolutional layers occurs later during the training process as the layers begin to represent useful abstractions.

5.2 Continuous Abstract States

Constructing continuous abstract state-spaces is not as intuitive as the uniform partition approach that we used in prior chapters. Our previous abstract state-spaces were just discretizations of the existing ground state-space. Obviously if we take this discretization to the “continuous limit”, we are just left with the original ground state-space. This is of no help for creating a continuous abstract state-space.

Instead of trying to abstract each state dimension individually, we look to abstract the states as a whole. For most high-dimensional environments — particularly environments represented as visual information where there is a lot of redundant information — the individual state dimensions are not inseparable. This allows us to reduce the number of dimensions when abstracting the state space. We opt to base our abstractions on the hidden variables within state-spaces that determine the values of large numbers of states. These are often referred to as “latent states”. The next section gives more explicit details about latent states.

5.3 Latent State-Spaces

Oftentimes environments contain properties that are not explicitly observed or part of the state representation. Such properties are referred to as latent. Despite being unobserved, latent properties can contain valuable information and can determine the values of many states. For example, if working with a pixel representation of a simple maze environment, the agent’s position is not directly available to the agent as part of the state-space representation and is thus latent. Crucially, the agent’s position within the maze can be inferred from the state-space — if one knows what to look for. It is extremely useful for the agent to know its position within a maze environment. Further, from a latent position variable, the pixels surrounding the position (in the shape of the visual depiction of the agent) can have their values determined by the agent's occupation of that position — supposing the agent takes up physical space.

If the agent understands the rules of an environment's pixel representation, then from a relatively small amount of appropriate latent properties, much of the original, visual state-space can be reconstructed. It can be a much simpler problem for the agent to learn policies on state-spaces consisting only of these latent properties. The ability of these policies to perform well will depend on the extent to which the original state-space can be reconstructed from these latent properties — this corresponds to how well the agent's abstract model matches its "reality".

When the state-space of an environment is altered to consist of latent variables, we will refer to this as a latent state-space, the individual states consisting of latent variables we will call latent states.

The question remains of how an abstraction function to map states to latent counterparts will be learnt. The next few sections will delve into the preexisting topic of autoencoders that we will adapt for this purpose. We will also take a look at how latent state-spaces and auto-encoders have previously been used within RL before moving on to how we will utilise them for reward shaping and the LPSA approach.

5.4 Auto-Encoders

An auto-encoder [28] is a neural network architecture consisting of an encoder-decoder pair. The purpose of an auto-encoder is to simultaneously learn both an encoder and decoder by feeding the combined pair training examples (x, x) — that is, the network tries to learn the identity function. After training has completed, the encoder and decoder can be split and used in a modular fashion. Typically auto-encoders bottleneck in size towards the middle of the network — the end of the encoder and beginning of the decoder. This forces the network to learn a compressed representation of the data. The idea being that the encoder then compresses the data into a smaller form and the decoder then decompresses this smaller form into the original data.

We can think of the encoded representation of the states as a latent state representation. Each variable in the compressed form can correspond to a latent variable — assuming the network learns to reconstruct the original data accurately then these latent variables are certainly sufficient. The latent variables learnt, however, may not correspond to any intuitive properties that a human

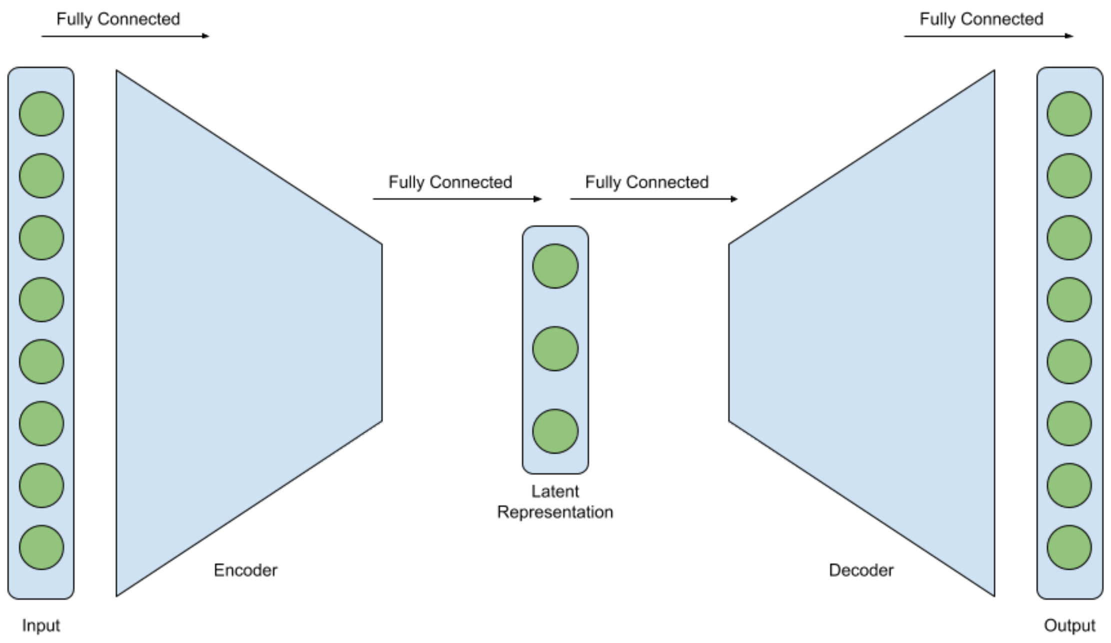


Figure 5.1: Neural network architecture for an auto-encoder

domain expert may identify.

Figure 5.1 displays the basic auto-encoder design, highlighting the bottleneck into a compressed representation consisting of latent variables. The standard auto-encoder utilises fully-connected layers throughout the architecture.

The auto-encoder is trained using standard back-propagation on the reflective training pairs (x, x) drawn from a training set. More formally, if we have a neural network for encoding and one for decoding representing functions $e_\theta : X \to Z$ and $d_\phi : Z \to X$, then the loss function is based on the error between the original input $x \in X$ and the reconstruction $d_\phi(e_\theta(x)) \in X$. For example, the loss function using mean squared error for a basic auto-encoder would be:

$$\mathcal{L}(x) = |x - d_\phi(e_\theta(x))|^2$$

Back-propagation with gradient descent can then find the weights θ and ϕ to minimise this loss, and thus find the best possible encoder-decoder pair.

5.4.1 Variational Auto-encoders

Many alterations and improvements can be made to the basic auto-encoder design. One such improvement are auto-encoders that can utilise convolutional layers in the encoder The Decoder can correspondingly use so-called "de-convolutional" layers which attempt to undo the previously applied convolutions. This allows the auto-encoder as a whole to handle higher dimensional data and learn latent representations for entities such as images.

The main problem with auto-encoders is the generative capability — the latent spaces learned by the encoder can often be discontinuous or sparse, causing the decoder to struggle to generate meaningful examples from much of the latent space.

A solution to this comes in the form of Variational Auto-Encoders (VAEs) [56][35]. A VAE differs from an ordinary auto-encoder by splitting the output in the last layer of the encoder. Instead of the encoder outputting a vector of n dimensions, two n-dimensional vectors are output. Rather than outputting an exact vector in the latent space, the encoder now gives a vector of means and a vector of standard deviations $\boldsymbol{\mu} = [\mu_1, \mu_2, ..., \mu_n]$, $\boldsymbol{\sigma} = [\sigma_1, \sigma_2, ..., \sigma_n]$. A single n-dimensional vector is then sampled from each dimension distribution: $\boldsymbol{z} = [\mathcal{N}(\mu_1, \sigma_1), ..., \mathcal{N}(\mu_n, \sigma_n)]$. This $\boldsymbol{z}$ is then fed to the decoder to produce an output. Intuitively, this adds some noise to the input of the VAE and forces a higher level of generalisation to take place.

More formally, if we have a neural network for encoding and one for decoding representing functions $e_\theta : X \to Z$ and $d_\phi : Z \to X$, then a derivation using Bayes' laws and basic probability theory [36] shows us that we should select θ and ϕ such that:

$$\theta, \phi = \underset{\theta,\phi}{\arg\min} \Big(-\underset{x \sim e_\theta(z|x)}{\mathbb{E}} \big[\log d_\phi(x \mid z) \big] + \mathbb{KL}(e_\theta(z \mid x), d_\theta(x)) \Big)$$

Where $\mathbb{KL}(p(x), q(x))) = \mathbb{E}_{x \sim p(x)}[\log(\frac{q(x)}{p(x)})]$ is the Kullback-Leibler divergence, a measure of how different two probability distributions are from each other. The equation $-\mathbb{E}_{x \sim e_\theta(z|x)} \big[\log d_\phi(x \mid z) \big] + \mathbb{KL}(e_\theta(z \mid x), d_\theta(x))$ can then be used as a loss to minimise with standard gradient descent approaches, in order to find θ and ϕ.

5.5 Using Auto-encoders and Latent State-spaces with Reinforcement Learning

In recent years auto-encoders have seen some use within the field of RL. We will briefly review some of the most influential work in which they appear and highlight their function and the benefits they bring to the learning process. It is important to additionally note that both auto-encoders and latent state-spaces see use outside of RL, but in the interest of relevancy and space we limit this section to applications or techniques that directly utilise RL.

5.5.1 World Models

In the paper "World Models" [30], the authors train a VAE to reconstruct frames from the OpenAI Gym [11] Car Racing environment. This is used in tandem with a Recurrent Neural Network to select actions.

The network model for this approach consists of three major components. These are the vision, memory and controller components. The vision component is a VAE that maps visual information into 32-dimensional latent states. The memory component learns to predict a distribution of future latent states based on the latent state history, action and current latent state. This component uses a mixture density network RNN [61] to do so. The controller is a single layer perceptron with an identity activation function. The controller receives a concatenation of the latent state and the distribution of the predicted future state. From this the controller selects an action.

After each time step, the agent receives a new observation frame from the environment. This is passed to the encoder and the latent state is identified. This is then passed to both the memory model and the controller. The memory model takes the latent state, the previous action performed, and its own previous output, and assigns probabilities to future latent states, which are then passed to the controller. Finally the controller selects a new action based on both the current latent state and the output of the memory component. This procedure can then repeat until the episode terminates.

At the time this paper was written (and it appears to still be the case at the time this thesis was written) this approach yielded state-of-the-art performance on the Car Racing domain, reaching the criteria for "solving" the domain.

The same paper then goes on to show how VAEs can be used to simulate environments, namely ViZDoom [33] (an RL domain adapted from the first-person shooter "Doom"), and train entirely within these "dreams" — the RNN repeatedly predicts the next latent state and this prediction is used to select an action. These “dream” policies are then shown to transfer well to the real environment.

A few downsides to this approach are that on occasion the agent learned to exploit the dream representation in ways that do not transfer to the environment. Further, the RNN is learning the dynamics of the dream environment and allowing the agent to view these inferred hidden states which may give more access to hidden states than the original observations. The authors note these limitations in the paper.

5.5.2 Robotics

Another VAE based approach is found in [22], where the authors attempt to tackle a domain based around manipulating a physical robotic arm in order to complete a variety of tasks.

Here the auto-encoder is used to bring down the state-space of webcam frames down to a more manageable size, this latent representation of the images is combined with non-visual state dimensions such as joint angles and velocities. Further, the algorithm utilises “feature presence” on the latent space which essentially filters out latent features which are not highly “activated” in one area of the image. The idea behind this feature presence concept is to remove aberrations from lighting or camera anomalies, it also smooths out latent trajectories. The final algorithm then uses model-based RL to train on the present latent states and non-visual state dimensions.

The approach was evaluated on a robotics-based task, where the agent’s action-space consists of a seven dimensional, continuous value, where each dimension corresponds to a torque value to be applied at a robotic arm’s specified joint. There are a number of tasks for the robotic arm to complete, ranging from sliding a Lego block to a designated position to scooping up a bag of rice with a spatula and transferring it to a bowl. The VAE-based approach was far more capable than an agent learning without any visual information using the same model-based RL algorithm. Further, this approach also outperformed other auto-encoder-based approaches at the same task.

There is an appealing conceptual contrast in the approach that the prior two approaches utilise their auto-encoders. The World Models approach aimed to use the auto-encoder to make the state-space a lower dimension and to train on this smaller state-space directly. The robotics approach, on the other hand, sought to augment an existing low-dimensional state-space with salient features identified from visual data. This contrast further emphasises the range of possible methods and approaches provided by auto-encoders and latent state-spaces.

5.5.3 Structured Latent State-spaces

In “Plannable Approximations to MDP Homomorphisms: Equivariance under Actions” [72], the authors introduce a method for constructing latent state-spaces that retain a structure closely resembling that of the original state-space. This approach attempts to create an “abstract” MDP that is arbitrarily close to a homomorphism with the MDP of the original domain. To distinguish this type of abstract MDP from the AMDPs we introduced earlier we will refer to these as “homomorphic MDPs” or “HMDPs”.

From the original, ground-level deterministic MDP $M = (S, A, R, T)$, an HMDP $H = (\bar{S}, \bar{A}, \bar{R}, \bar{T})$ is desired to be constructed. Functions Z and Y denote respective mappings from states and actions to their latent-space counterparts. To achieve this homomorphic quality of the latent representation, two properties are required. The first, and most important of these is action equivariance. This is a property of a mapping denoting that transitions in the state-space match those in the latent state-space. More formally, action equivariance is represented by

$$Z(T(s, a)) = \hat{T}(Z(s), Y(a))$$

Secondly, it is also required that

$$R(s, a, s') = \bar{R}(z, y, z')$$

for $z = Z(s), z' = Z(s'), y = Y(a)$.

Distance functions for the above property equations are included as terms in the loss function for learning the latent representations of states, actions, transition, and reward functions. Also included is a term attempting to maximise the distance between non-adjacent states in the latent state-space to avoid learning trivial representations. The authors prove that at a loss of zero, the learned HMDP is an exact homomorphism of the original MDP.

Once the HMDP has been constructed, it is discretised to allow Value Iteration to learn optimal Q-values (and thus an optimal policy) within this discretized setting. The agent can then interact with the original state-space, observe state s, map s to $z = Z(s)$ and identify $Q(z, a)$ for each $a \in A$ by interpolating the discretized Q-values. This can then be repeated until an episodes termination, giving a final policy as close to optimal in original, ground domain as the discretization granularity and homomorphic loss allow.

The policy from the HMDP is used directly in the ground MDP rather than informing a ground policy or designating macro-actions to be taken. The smaller size of the latent representation, in theory, makes the HMDP easier or quicker to find an optimal solution. The homomorphic property of the HMDP then ensures that the learned HMDP policy is also suitable for the original MDP.

The method was evaluated on a few simple visual domains. This approach outperformed other baselines utilising latent state-spaces including an adaption of the World Models [30] algorithm. Principal Component Analysis (PCA) visualisations also showed that the latent representations learned by their approach appear cleaner and more similar to the original state-space than other latent representation methods. However, this doesn't necessarily tell the whole story as the 50-dimensional latent space is reduced to 3-dimensions by PCA. Much of the ground-space structure of the original state-space could have been kept in the other latent representations and been lost or distorted in the PCA visualisation.

Further, the environments used and the general outline of the approach imply a deterministic MDP. Many complex environments do not have deterministic transition functions. The authors state that the approach can be adapted for stochastic MDPs however.

A possible future avenue of work could be to leverage structured latent state-spaces for a reward shaping approach similar to the core idea that has been used throughout this thesis. In this approach, a much larger loss could be allowed for creating the HMDP. The HMDP could then be thought of as a much broader approximation to the real MDP. Reward Shaping could be used in the same manner that we saw in previous chapters, using Potential Based Reward Shaping and setting the potential function $\phi(s) = V(Z(s))$. The advantage to this type of approach is that the HMDP does not have to be very accurate which could take a lot of computation to achieve, yet, through reward shaping would still provide a useful shaped reward. While this idea is not explored further in this thesis (due to other approaches appearing more promising) it does seem worthy of consideration by the RL community.

5.6 Latent State-spaces as Abstraction

A common theme running through the papers overviewed in the previous section is the use of a network to reduce the size of the representation. This is where the "abstraction" occurs. At the core of it, the networks mapping from ground state-spaces to latent state-spaces are an abstraction. While this may seem self-evident, explicitly reframing these networks as such as opposed to "compression" or "encoding" can remind us of the other utilities that these networks and latent state-spaces may have for improving RL performance.

It turns out that we have seen applications of latent state-spaces already in this thesis. In a CNN the convolutional layers are mapping the high-dimensional visual observation into a lower-dimensional latent representation of salient features. This is then a small enough representation to be handled by fully-connected layers. In fact, the only difference between the encoding component of a convolutional auto-encoder followed by fully-connected layers of a neural network and a CNN is the training technique.

This abstraction philosophy of reducing the representation size of states is distinct from those typically seen in AMDPs. Reducing the representation size doesn't seek to create an "abstract" instantiation of the domain, representing a high-level overview of the task. Instead, they try to recreate the domain as closely as possible in a smaller latent representation. The intuition being that in these smaller domain spaces it will be easier or quicker to find an optimal solution.

The research so far into latent state-spaces and RL has focused on training directly on a latent state space and transferring or using the learned policy; little-to-no research appears to have been done on using autoencoders for the hierarchical approaches to abstraction, such as reward shaping from AMDPs or Hierarchical Deep Q-Learning. This is the direction in which we will orient our research, focusing specifically on AMDPs.

5.7 Utilising Latent State-spaces

We can think of the structures that map state-spaces to latent variables as an abstraction mapping and the space contrived from latent variables output as an abstract state-space. We will refer to the state-spaces contrived from latent state dimensions specifically as latent state-spaces for the remainder of this thesis to help distinguish the types of abstract state-spaces we have utilised.

Many methods could be used to create suitable latent state-spaces. This work will focus on utilising variational auto-encoders to achieve this; they are easy to train and can reduce the size of state-spaces effectively.

Taking this approach we can use latent state-spaces for reward shaping. To do this there are a few things which we require. We need an auto-encoder, trained to encode a visual representation of an environment into a smaller number of dimensions. We also need a neural network architecture that takes as input the encoded state and outputs a learned value denoting that encoded state's value — this is essentially a deep AMDP. Finally we also require an ordinary deep Q-network for the environment we wish to train and augment with reward shaping.

For this approach to be "worthwhile" from an efficiency perspective, we need our approach to reach the same or higher performance as a Vanilla DQN in the same amount of time (or similarly, reach the optimum performance in less time). However, since this approach has hitherto not been used for shaping, even just a proof of concept is a novel contribution, showing that agents learning latent states can be used to shape rewards effectively. It is with this proof of concept that this chapter culminates. In the proceeding sections, a brief overview of the method is given followed by a more detailed look at each component of the method.

5.8 Method Overview

Here we give a very broad overview of the Latent Property State Abstraction (LPSA) method before detailing the individual steps in the proceeding sections. The first step is to train the auto-encoder; random rollouts of episodes from the environment produce samples of high-dimensional states. The auto-encoder trains in a self-supervised manner using these states.

Once the allotted training time for the auto-encoder has elapsed, the encoder and decoder are split and function as a state abstraction function and its (approximate) inverse.

Now that the abstraction function is available (the encoder), the neural network representing the AMDP's Q-function can be trained using a suitable RL algorithm (such as DQN) on the environment but first passing the high-dimensional states from the environment through the encoder to yield abstract states. The AMDP's Q-function is then learning to associate states from the latent state-space to Q-values for each action.

Once the AMDP is trained, we can use it for shaping in the same manner as UPSA (from Chapter 4) and other shaping methods; the ground network is trained on the desired environment as normal, except that an additional reward is given to the agent based upon the discounted difference in the values of the abstract states it has just moved between.

As long as the learned abstraction function is mapping similar states to similar areas of the latent state-space — requiring the auto-encoder to be "good" — and the abstract network is learning to perform "acceptably" on the latent state-space then the reward shaping can boost the ground agent's learning, particularly early on in the training process. A diagrammatic overview of the method is given in Figure 5.2.

The two-step "pre-processing" can be computationally costly. Obviously if the cost of pre-processing is larger than the reduction in convergence time that the method provides then the approach is not "worth it" from an efficiency perspective. This will depend on the environment used and how robust the auto-encoder and abstract policy need to be to attain a suitable shaping function. Unfortunately in the domain we used here (car racing), the cost outweighs the gain. Nevertheless, the pre-processing is something that, in principle, can be made far more efficient through parallelization that was not leveraged in this project due to the added complexity of implementation. Instead, this approach is modelled as a proof of concept and demonstrates that the shaping function provided by the self-learned abstraction is useful for improving the performance when compared to an unshaped agent.

We will explore these stages in more detail in the coming sections, but first we introduce the domain we will use to evaluate our method. Introducing the domain first allows us to explain some of the finer details of the method with reference to a concrete environment example.

5.9 Experimental Domain

Due to the fact that this approach for abstraction is able to handle higher dimensional input we would like to move away from the simple experimental domains of the previous chapters. The domain we will primarily use to evaluate this approach is a discretized version of OpenAI Gym's Car Racing domain [11]. A visual depiction of the environment is given in Figure 5.3.

In this environment the agent exerts control over a car, and must guide it around a procedurally generated track. The agent must try and keep the car within the track. The car receives a reward of -0.1 after each time step as well as $1000/N$ for each segment of track visited for N the total number of track segments.

The agent perceives a 96×96 pixel representation of the environment, including visual cues for the vehicle's speed, ABS (anti-lock braking system) sensors and the angle of the steering wheel.

In the original environment the action-space is essentially $[0, 1]^3$, the accelerator, brakes and steering each take a continuous value between 0 and 1 denoting the extent to which the control is activated.

In our version of the environment we gave the car 20 discrete actions corresponding to different applications of the accelerator, brakes and steering. The reason for doing so is to evaluate agents with a focus on the high-dimensional state-space rather than a more complex action space. We are trying to show that utilising a reward shaping function derived from an agents interactions with an abstract environment can improve learning speed — the continuous action space is not necessary to demonstrate this.

Since the original environment uses a continuous action space, our agents may not reach the same level of performance overall as those that can utilise the continuous actions — however, we can still highlight the improvement to learning speed that our method brings compared to Vanilla DQN on the same discretised

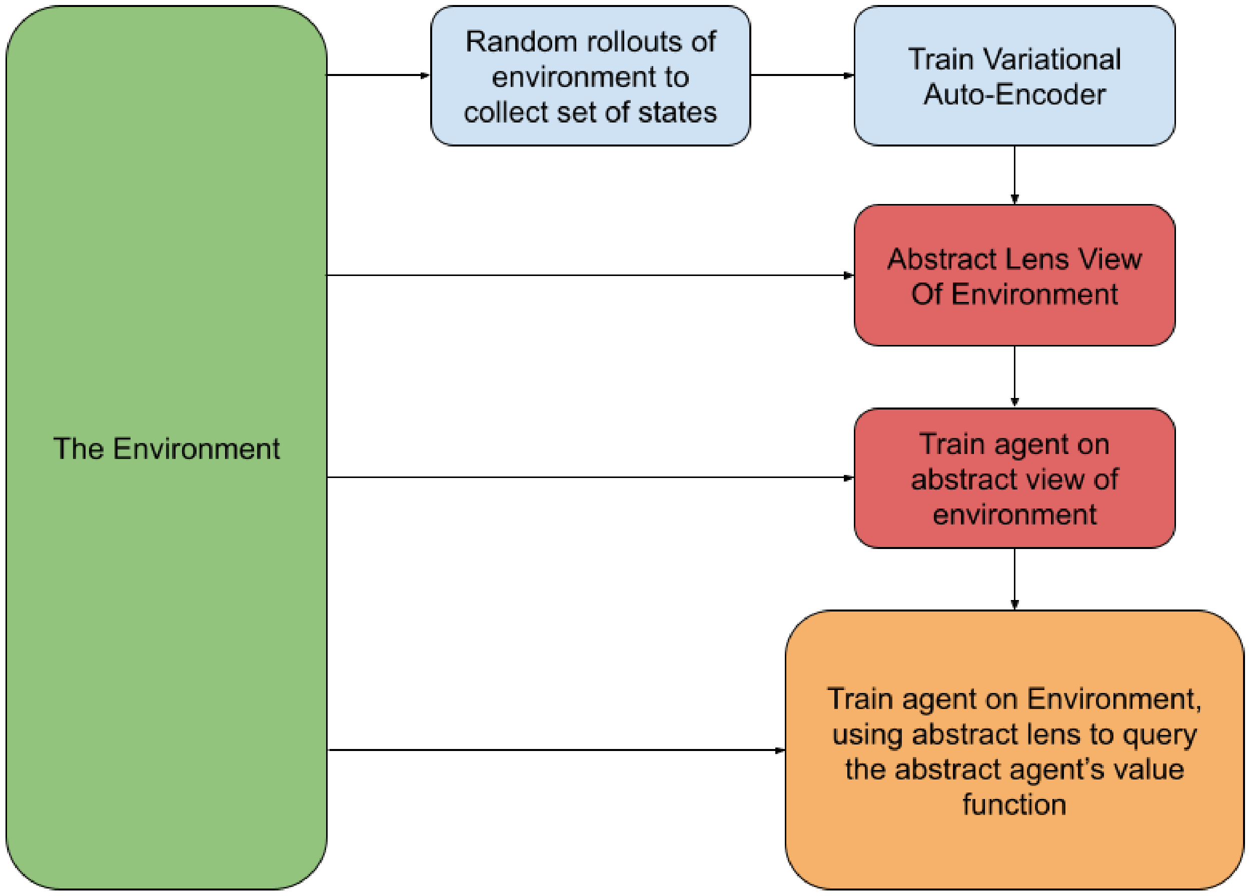

The Environment
Random rollouts of environment to collect set of states
Train Variational Auto-Encoder
Abstract Lens View Of Environment
Train agent on abstract view of environment
Train agent on Environment, using abstract lens to query the abstract agent's value function

Figure 5.2: A broad overview of the Latent Property State Abstract method. Arrows denote the flow of informational dependencies. Stages coloured blue correspond to the auto-encoder, red to the abstraction process, orange to the ground agent and green to the environment.

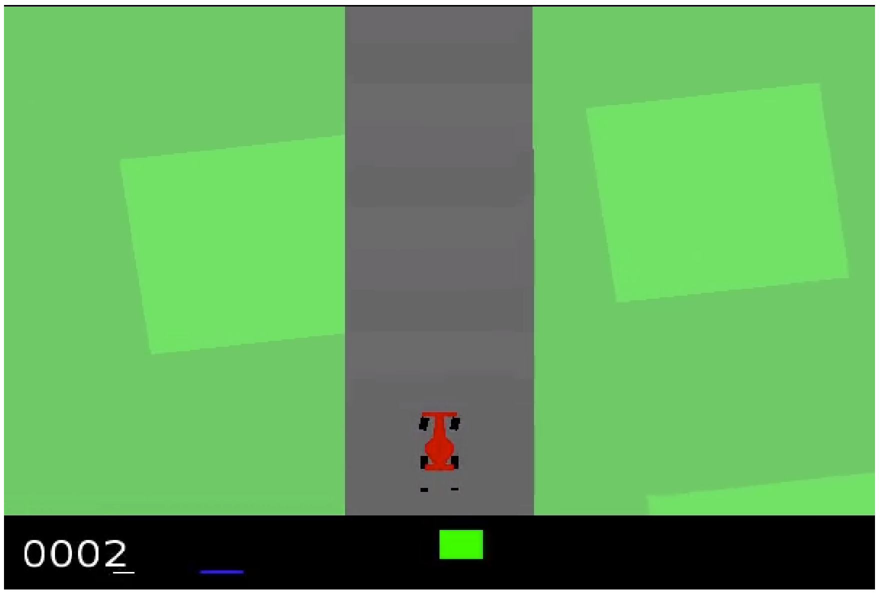


Figure 5.3: Visualisation OpenAI Gym's Car Racing domain, image captured from the domain running with visualisation on. This is what the agent "perceives" as an array of RGB pixel values.

environment.

In order to speed up the learning for the agents, we utilise an early stopping method for the environment; if the agent fails to enter a new track segment in 12 time-steps (and therefore experience a positive reward) the episode terminates. This is necessary as otherwise the environment only terminates if the car completely leaves the playing field. This can take a long time for very little benefit. Note that we do not employ this for our implementation of the *World Models* approach which we compare our algorithm against. The reasons for this are given in Section 5.13.1.

5.10 Training the Auto-encoder

The encoder is the crux of the LPSA method. The quality of the abstraction function will determine how well the abstract agent can learn a policy for the latent state-space as well as how applicable the shaping is to the current state. It is therefore imperative that the auto-encoder has minimal reconstruction loss.

There are important decisions to make for training the autoencoder, the two most important are that of architecture and the number of latent state dimensions.

We opt to use a variational auto-encoder primarily for its ability to create a densely populated latent state space — this will be pertinent for encoding states outside of those encountered by the autoencoder during training. The exact architecture is in figure 5.4, where the separate encoder and decoder components are detailed. Combining the output of the encoder to the input of the decoder constitutes the full auto-encoder architecture.

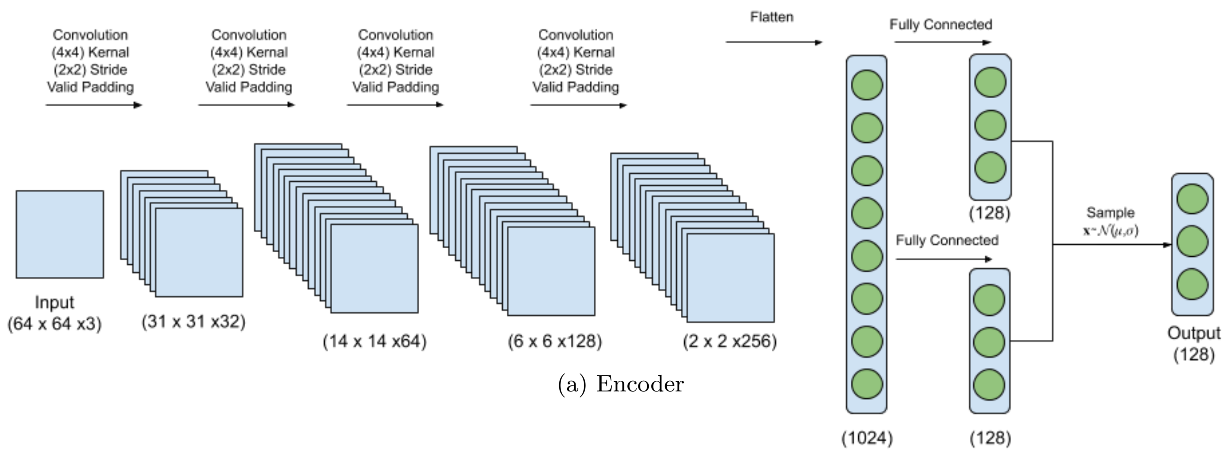
To begin, we downsample the original $96 \times 96 \times 3$ image from the environment into a $64 \times 64 \times 3$ image. This reduction in resolution helps reduce the amount of information that needs to be compressed without drastically altering the quality of the image. The number of latent state dimensions will affect the amount of information that can reasonably be encoded. A higher number of latent state dimensions will intuitively allow for a lower final reconstruction loss with diminishing returns. On the other hand, increase the number of dimensions will add more parameters to the abstract model and make it take longer to train, as well as reducing the degree to which the environment is “abstracted”.

An agent interacts with the environment using a random policy for a specified number of time-steps. This policy favours acceleration in order to visit more states. Additionally, random start positions are used to again maximise the diversity of states visited. For each training step, a batch of states are drawn uniformly from the training set, the loss is calculated and the weights are updated using the loss function and procedure outlined in Section 5.4. Once enough episodes have elapsed the training ends. This will unfortunately depend on the environment and needs to be found empirically.

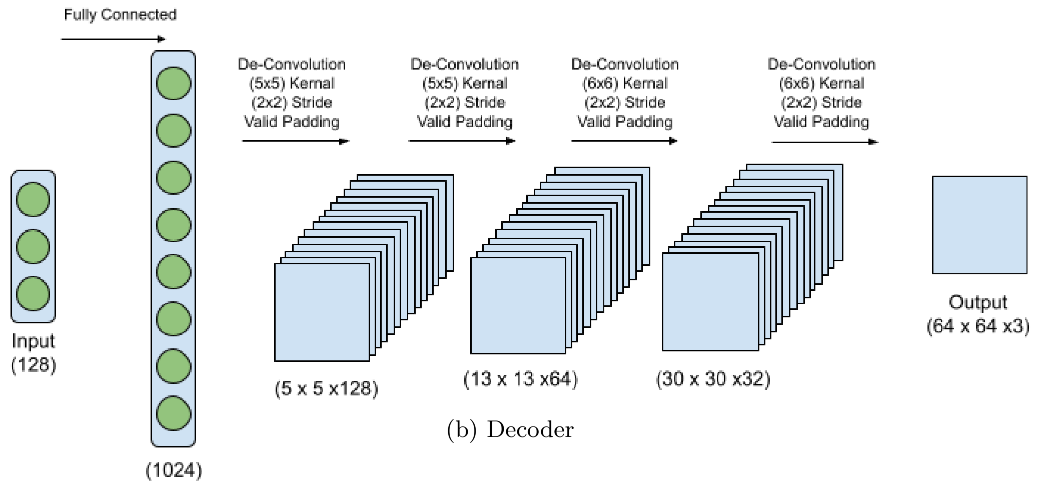
For our experiments with the Car Racing domain a batch size of 128 was used for each training step and 5000 batches were used for training. Further, 128 dimensions were utilised for the latent state-space. The auto-encoder and rollout policy implementation from the GitHub repository: [13] was utilised to train our auto-encoder. This repository itself was based on the author of [13]’s initial attempts to implement the aforementioned *World Models*[30] paper, and contained an auto-encoding architecture adapted for the Car Racing domain. Alterations to the exact architecture are made to better suit our purposes.

Figure 5.5 shows the typical change in VAE reconstruction loss over the course of the training procedure. As expected, the loss initially rapidly decreases before slowing down at a relatively stable value.

It is necessary to highlight that VAEs (like many deep generative models) do not typically perform as well or as reliably when the underlying data distribution changes [73]. Even within the same environment, this poses issues if the distribution of states during the random rollouts is different from the distribution of states during training and evaluation. This highlights the importance of



(a) Encoder



(b) Decoder

Figure 5.4: The encoder-decoder components of the auto-encoder architecture.

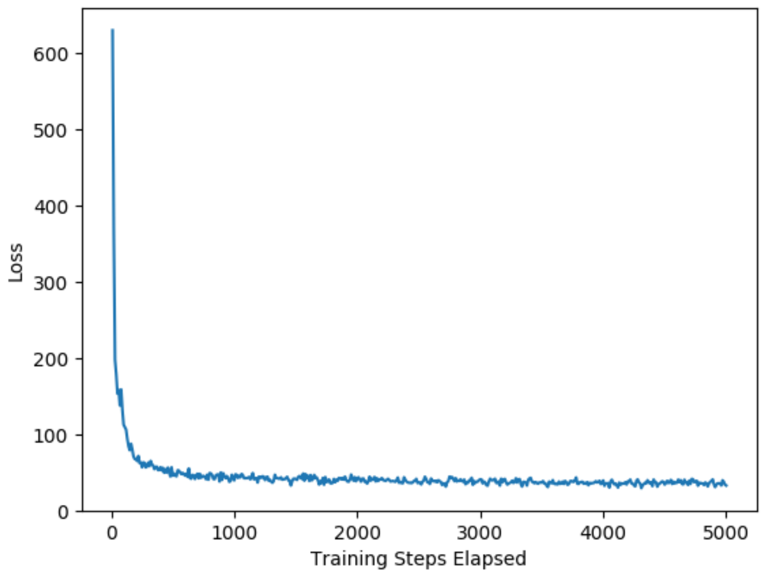


Figure 5.5: A typical plot of loss against the number of training steps for our Variational Auto-encoder when trained on frames from the Car Racing environment.

a strong exploration policy for the rollout phase. In our approach we used the randomised rollout policy that favoured acceleration as well as using randomised track starting positions. These help to ensure that the auto-encoder is being trained on a distribution as close as possible to that encountered during evaluation of a fully trained agent. However, this does pose a potential limitation for environments where this is not feasible or where exploration is very tricky. More sophisticated approaches may be necessary for such domains.

5.10.1 Auto-encoder Reconstruction

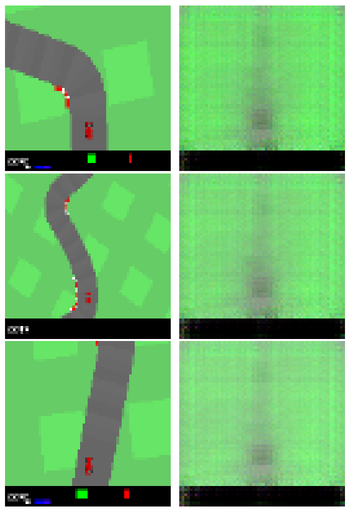
While there are no theoretical guarantees for ensuring that the autoencoder will reach an acceptable level of loss, or that this level of loss will transfer well to visual construction, it is easy to intuitively see approximately how well the autoencoder is learning by comparing the input images and the reconstructed output.

Figures 5.6 and 5.7 show some of the reconstruction attempts from throughout the training process; highlighting the auto-encoders reconstructive capabilities after 50, 500 and 5000 auto-encoder training epochs. Each epoch corresponds to an update to the auto-encoder's weights utilising a batch size of 128 samples.

At the point when 50 epochs have elapsed, The reconstructed images largely look the same and there is very little resemblance to the original image. The auto-encoder does seem to be beginning to capture that there is usually a track segment somewhere in the centre of the image, indicated by the grey blob. After 500 steps the reconstructions are beginning to resemble the original image far more. However, zoomed out portions of the track and certain edge cases still reconstructed unsatisfactorily. Finally, once training has completed and 5000 epochs have passed, the replications are more accurate. Track edges appear more defined. The network still struggles with extremely zoomed out frames due to the high level of detail required to represent these. However the other frames are replicated with a high degree of accuracy. While they are not perfect reconstructions and perhaps a little blurry, the auto-encoder has certainly managed to capture a decent approximation of the current frame. Recall that our purpose of using the autoencoder is not perfect reconstruction — we want to utilise the autoencoder as an abstraction function. This will typically require less-than-perfect reconstructions due to the very nature of abstraction.

5.11 Training the Abstract Network

Now that the auto-encoder has concluded its training, we can utilise the encoder portion of the auto-encoder to act as a state abstraction function. We denote


(a) 50 training steps

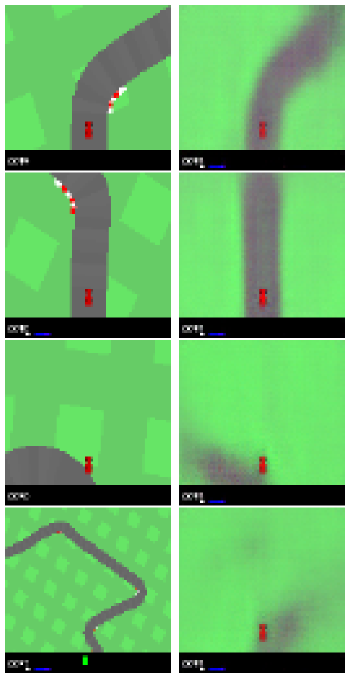

(b) 500 training steps

Figure 5.6: A sample of environment frames and their reconstructions after 50 and 500 training epochs

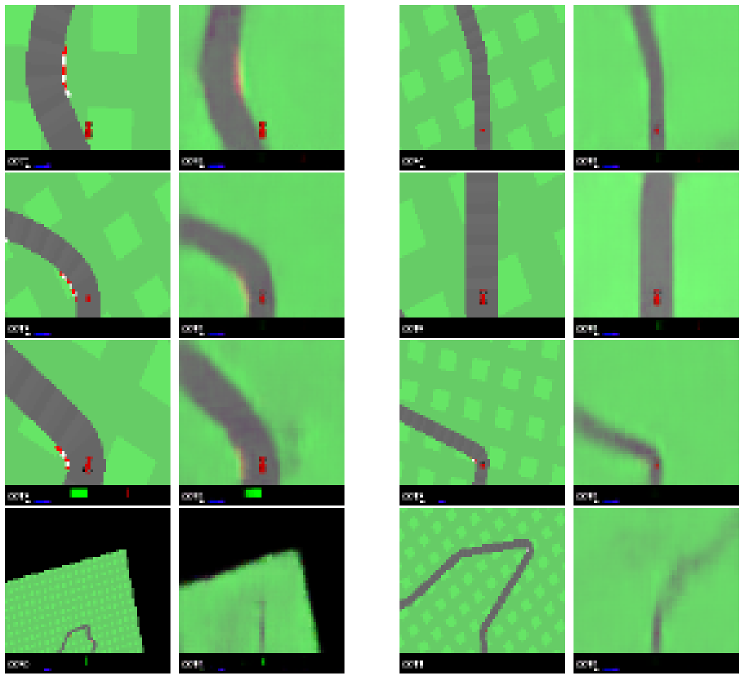


Figure 5.7: A sample of environment frames and their reconstructions after 5000 training steps

this encoder as a function Z_S, mapping elements of the ground state-space to the latent state-space.

We train a neural network using DQN on the environment with the latent state-dimensions as the input. Each action is repeated three times by the agent to reduce the computation time required — updates and queries to the neural network are then performed only every third frame, rather than after each frame. This essentially triples the amount of episodes we can train in a given time-frame while having little effect on the performance of an agent due to the fact that thirty frames are occurring every second, so the impact of an individual action is relatively low. Further, the observation received by the agent is a stack of four latent states from when an action was last selected. Therefore the shape of each observation is (128×4) and consists of latent states from time-steps $t = 0, -3, -6, -9$.

After each observation, The agent adds the following to its experience replay: (o, a, r, o') for current observation o, action a, new observation o' and immediate reward r. Note that o and o' are stacks of the latent states as described above. It is the observations that are added to the experience replay instead of the ground states as in ordinary DQN. From here on, DQN proceeds as normal to learn a policy over the latent state-space with the ground actions available.

It is important to note that even though the abstract agent is learning to associate stacks of latent states with values, the agent is interacting with the ground environment viewed through an abstract "lens". This means that no abstract transition function needs to be learned or provided to the agent — the abstract transition function of the environment is the ground transition function with states mapped into latent space.

Slightly more formally, assuming that the ground transition function of the environment is $P_{\mathcal{M}}$, then the abstract transition function will be $P_{\mathcal{A}}(Z_S(s), a, Z_S(s')) = P_{\mathcal{M}}(s, a, s')$. Of course, this is not known to the agent at the point of learning since $P_{\mathcal{M}}$ is hidden, however this is the transition function that the abstract agent is subject to. Similarly, for rewards, the ground reward is given to the agent, but associated with latent values. $R_{\mathcal{A}}(Z_S(s), a, Z_S(s')) = R_{\mathcal{M}}(s, a, s')$.

Once a predetermined number of episodes or steps have elapsed, training finishes and the Q-Network represents the function mapping latent states and actions to values — that is, the abstract Q-function. For our experiments we chose to train both the ground agent and abstract agent for the same number of episodes — 8000. A single episode took less time for the abstract agent however, owing to the smaller neural network requiring fewer calculations to update the weights, as well as the abstract agent generally not surviving as long in the environment in a given episode.

Since the abstract agent is capable of interacting with the ground environment (because of the identical action-space), we can evaluate its performance directly. It is worth noting that we do not expect the abstract agent to perform spectacularly, its perception is limited by only receiving latent states. However, we do not require that the abstract agent performs well, only that the value function it learns is useful for reward shaping, this could take the form of discouraging behaviour that it has identified as “bad”.

For completion we will include the abstract agent’s performance in results to verify that training directly on latent states is not inherently superior to Vanilla DQN. This will further support our hypothesis that the “abstractness” of the shaping is providing the boost to learning speed, rather than access to a simpler environment model.

5.11.1 Differences to Uniform Partition State Abstraction

There are some other important properties of LPSA that differ from the UPSA method proposed in the previous chapter. The first and most obvious is that of abstract state-shape. No longer is the state abstraction function restricted to uniform partitions along each state dimension. This allows for more flexibility in the state abstraction function, as well as allowing redundant information to be minimised. Further, the abstract state-space in the latent-state-based method is continuous, allowing for more nuance in categorising abstract space.

A further difference in the approaches is that in the previous chapter, the AMDP was explicitly constructed and then solved through Value Iteration. Now that we are in a continuous state-space, this is not so feasible. Instead we opt to use a model-free RL approach to ascertain the values of abstract states — this is the task of the abstract network. This approach is similar to that employed by Multigrid Reinforcement learning (MRL) [29] (see Section 2.6.4 for an overview) for learning the abstract policy — although due to the difference in abstract problem scales, our method uses DQN to achieve this rather than SARSA. In this way the method detailed in this chapter can also be viewed as an extension of MRL, but in a different direction than the prior uniform state abstraction.

A final difference is that in UPSA we constructed a new abstract action set, denoting the states reachable from the current abstract states. Both MRL and LPSA, on the other hand, retain the ground action set. For LPSA specifically, the continuous nature of the abstract state-space make it infeasible to have an abstract action for each reachable abstract state. However, we can utilise the existing ground actions as abstract actions within the latent state-space. Keeping the ground actions also allows our abstract agent to interact with the environment fully and provides a means of evaluating the abstract agent’s performance.

Table 5.1 gives an overview of the dimensions in which each of these methods differ.

Property	MRL	UPSA	LPSA
Abstract state-space	Discrete	Discrete	Continuous
State-dimension aligned?	Yes	Yes	No
Model-based AMDP?	No	Yes	No
Abstract Actions	Ground Actions	Adjacent Abs. States	Ground Actions

Table 5.1: Comparison of properties of MultiGrid-RL, Uniform Partition State Abstraction and Latent Property State Abstraction

Parameter	Ground Agent	Abstract Agent
α	4×10^{-4}	4×10^{-4}
γ	0.95	0.95
τ	1×10^{-2}	1×10^{-2}
ω	2.5×10^{-3}	N/A
ϵ	$0.1 \rightarrow 0.01$	$0.1 \rightarrow 0.01$
Batch Size	128	128
Episodes	8000	8000
Warm up Steps	10000	10000

Table 5.2: Hyper-parameters used for experiments

5.11.2 Architecture and Hyper-parameters

A feed forward neural network is used to train the abstract network. Figure 5.8 gives the exact architecture for this network. Hyper-parameters are given in Table 5.2.

5.12 Utilising the Abstraction

The abstraction is utilised in exactly the same way as in the previous chapter. To reiterate, a DQN agent is learning on the environment as normal, the difference comes from the modified reward function. As the agent makes transition $s \rightarrow s'$, the abstraction function Z_S, which is now a VAE, maps s and s' to their abstract, latent states $Z_S(s)$ and $Z_S(s')$.

The abstract Q-function then maps these abstract states to their values $V(s) = \max_{a \in A_{\mathcal{M}}} Q(Z_(s), a)$.

The reward given to the agent for this transition $s \rightarrow s'$ is then the observed reward from the environment plus the discounted difference in abstract state

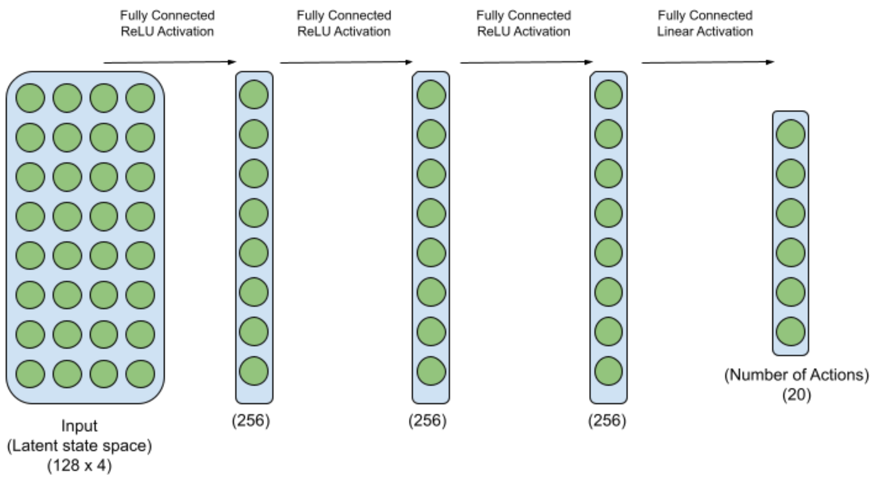


Figure 5.8: Abstract Agent Neural Network Architecture

values: $r + \gamma V(Z_S(s')) - V(Z_S(s))$. The experience

$$(s, a, s', r + \gamma V(Z_S(s')) - V(Z_S(s)))$$

is added to the agent's experience replay for training.

5.12.1 Training the Ground Network

Apart from the modified reward function to utilise reward shaping and architectural differences, the compared agents are identical. Here we detail the network architecture and training procedure, along with all relevant hyperparameters.

Again there is a small amount of pre-processing performed to the environment states in order to aid the learning process. This time the image is converted to grey-scale from colour, where each pixel can now take a value between 0 and 255. Each action is repeated three times by the agent to reduce the computation time required (following the same reasoning as for the abstract agent, repeating the same action three times reduces the number of times the neural network needs to be updated and queried). Once again the observation received by the agent is a stack of four frames from when an action was last selected. Therefore the shape of each observation is $(96 \times 96 \times 4)$ and consists of the frames from time-steps $t = 0, -3, -6, -9$.

The ground network used consisted of two convolutional layers interleaved with Max Pooling followed by a flattening and two fully-connected layers, with an

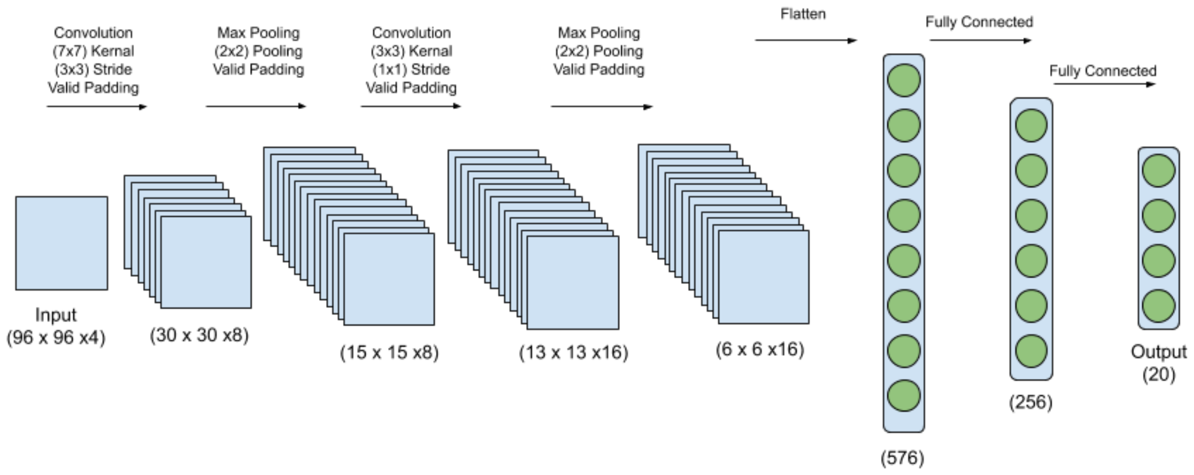


Figure 5.9: Ground Agent Neural Network Architecture

output node for each available action. The exact architecture is given in figure 5.9.The hyperparameters of note are listed in Table 5.2.

5.13 Results

Here we give some results evaluating the performance of our approach against benchmark agents (notably DQN and World Models). We end this section with the final results comparing the Vanilla DQN against our DQN augmented with LPSA. We also compare both of these methods against learning directly on the latent state-space (the AMDP) to ensure that the latent representation is not simply easier to learn on overall.

We do not compare against MRL or UPSA primarily due to the infeasible amount of memory that would be required. Each state dimension corresponds to one pixel. Each pixel can take a value between 0 and 255. Further, there are $96 \times 96 = 9216$ pixels in total. If we wanted to just split each state dimension into two equally sized partitions, there would be a total of 2^{9216} abstract states. Whilst this is far fewer than the original state-space of 256^{9216} states it is still far too many to handle discretely or tabularly — especially since it is estimated that there are approximately 10^{80} atoms in the universe. It was therefore determined that successfully implementing these algorithms for such a large domain was somewhat out of scope for a PhD thesis.

5.13.1 LPSA Results

In figure 5.10 we can see that utilising reward shaping provides a significant boost to the initial learning performance when compared to Vanilla DQN. This is kept up until the agent converges on the near-optimal score for the Car Racing environment.

Both LPSA and DQN eventually converge on similar policies with similar scores, however it is clear that our LPSA approach performs better than DQN after any fixed number of episodes.

We also see that the abstract agent can quickly converge to a sub-optimal policy. However it is also clear that the abstract agent will not converge on the same near-optimal policy as the other two methods — at least in a feasible amount of time. The abstract agent is seemingly limited by the granularity of the abstract environment.

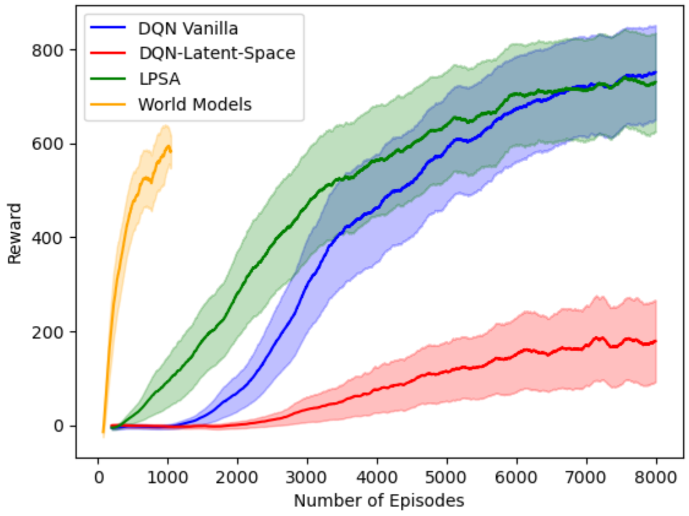
It is therefore clear that it is the shaping function derived from the abstract agent that is giving LPSA the boost to learning performance and that it is not the case that the abstract representation of the environment is inherently easier and that we are passing the abstract agent's solution to the ground agent.

The abstract agent's solution to the abstraction of the environment is capturing key skills that are transferable to the ground environment. This makes it apparent that LPSA's use of an auto-encoder is successfully capturing important abstract aspects of the environment.

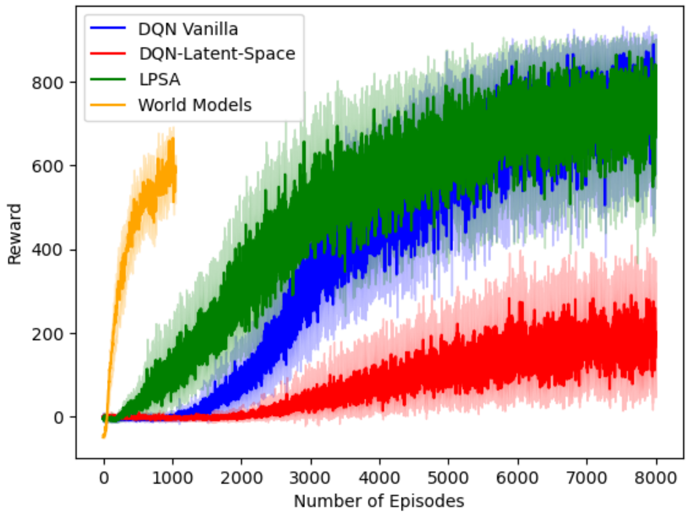
The aforementioned *World Models* paper [30] at the time of writing claims to have the highest performance on the Car Racing domain and is the current state of the art. We also compare our results against their method [1]. Their approach also utilises auto-encoders and was outlined in Section 5.5.1.

The *World Models* approach takes far fewer episodes than LPSA to converge on a strong policy, however each individual episode takes vastly more time to complete. Whilst their approach does allow for an ultimately more stable policy able to perform to a near-optimal level, the time taken to achieve this becomes disproportionately prohibitive with diminishing returns. We ran the *World Models* approach for a approximately the same amount of time as the other agents (roughly 24 hours) and report the results for the amount of episodes that elapsed. At the end of the set training time, *World Models* has a marginally

[1]The method used here was adapted by Sajjad Kamali Siahroudi from L3S, University of Hanover, from an existing code-base [24] as part of an as-yet unpublished paper based on this chapter



(a) Smoothed Results



(b) Unsmoothed Results

Figure 5.10: The average reward received by an agent plotted against the number of elapsed episodes. Each agent was trained for approximately 24 hours. The averages are based on 25 repetitions of the learning process for each agent and the shaded regions represent a 95% confidence interval. The smoothed agent used a moving average of window size 20 for World Models and 200 for the other agents (Due to the difference in the number of episodes).

worse performance than either Vanilla DQN or LPSA. Additionally it is worth noting that since the action-space in our environment is discrete, the *World Models* approach may perform differently than reported in the original paper.

The results given here highlight the difficulties that may occur when trying to compare methods that are very different. Since we may not wholly rely on comparisons by episodes or time alone a more detailed analysis is required, utilising elements from both.

The Car Racing domain is not ideal for comparing time taken — an episode varies wildly in length depending on performance. Due to this, we opt to plot the mean reward against the number of elapsed episodes while separately noting the average number of agent steps per second. This way, we can compare the performance fairly over a set period of time, without disadvantaging the agent for learning to perform well early on. Table 5.3 lists the average number of steps performed per second by each agent.

Further, we cannot easily make useful comparisons between Vanilla DQN or LPSA against *World Models*. The reason for this is due to the sheer difference in approaches. *World Models* relies on an evolutionary approach known as CMA-ES (Covariance Matrix Adaptation - Evolution Strategy) [31]. Under this approach, multiple members of the “population” are evaluated concurrently, each having their performance scored over a number of episodes (in our case, 4 episodes) taking the average of these scores to assess each member. After each member has been evaluated, the next generation is created based on properties and strategies favoured by the most fit members of the previous generation. The complete details are beyond the scope of the thesis and given in [31].

In order to maximise computational throughput, we opt to disable the early-stopping mechanism for the *World Models* approach. This way, each member of the population can always be learning, rather than waiting after potentially stopping early — the majority of this time will have to be spent anyway as certain agents begin to excel and would not trigger the early-stopping mechanism (recall the members of the population are evaluated concurrently). There is a slight trade-off here as even the best agents don’t always use the maximum number of steps in this domain, but this seems the best choice when there is a large number of members in the population.

This is all relevant to our evaluation because each episode now takes many more steps to complete — at least initially in the training procedure. The effects of this are two-fold — first, each episode takes longer as it consists of more steps, and second, more insight can potentially be gleaned from each episode (again, there are more steps).

Another interesting point of comparison is the way in which the methods, LPSA and *World Models*, differ in their approach to creating a model. *World Models*

Vanilla DQN	AMDP	LPSA	World Models
18.45	22.42	13.38	10.63

Table 5.3: Average number of steps per second performed by each agent over the 25 training runs (rounded to two decimal places).

attempts to learn a model of the environment that map states to latent states as well as being able to predict the next latent state from the current latent state and action. On the other hand, LPSA aims to learn just a map from ground to latent states — A sort of pseudo-model. As a result, *World Models* contains a more capable model at the cost of a more complex architecture that continues to use computation during training. Comparatively, LPSA uses a simpler architecture (just the VAE) but cannot predict future latent states. Instead, LPSA leverages the existing environment and trains the abstract agent using the real environment viewed through an “abstract lens”. The benefit of this approach is that once the shaping function has been learnt, the simpler ground agent’s architecture can be exploited more efficiently than *World Models’* more complex structure. Having some kind of model can potentially bring many benefits, such as allowing the agent to estimate the effects of its actions or calculate counterfactual situations.

All of these factors together make for a difficult comparison. However, we do not need to decide that one approach is inherently better than the other — what this really shows is that LPSA is able to utilise reward shaping from an automatically learned shaping function to perform on a similar level to the *World Models* approach.

5.13.2 Caveats

Both LPSA and *World Models* include warm-up stages that are not included in our comparison — For LPSA this is training the auto-encoder and abstract agent, for *World Models* this is training the auto-encoder and recurrent neural network for the environment model. In the case of our experiments, at present, it is not “efficient” to do this approach. However, this can, in principle, be addressed with parallelization of the pre-processing stages. This was not done in this thesis due to the added implementation complexity and other issues. It is also worth mentioning that the same downside also applied to *World Models* in our experiments, which took longer to converge to a strong policy than Vanilla DQN. This is likely due to the simplified (discretized) action space of our Car Racing environment, which is easier to learn in. That being said, this is acceptable as our goal has not been to create a practicable RL algorithm, but rather to demonstrate that suitable abstractions can be created using auto-encoders for the state abstraction function and that these abstractions can be utilised to improve an agent's learning rate through reward shaping. This we have indeed demonstrated as the difference in agent performance during the early stages of training is significant.

5.13.3 Comparing Abstraction Functions

Now that we have seen the effect that utilising LPSA can give, we give a detailed analysis comparing properties of the abstraction functions used by UPSA and LPSA.

The primary advantage UPSA has over LPSA is its simplicity and understandable nature. This rises from the fact that it preserves not just the state-adjacencies, but also (assuming each state dimension has a linear ordering of its elements) the orderings of states within their relative state dimensions. That is, for $s, s' \in \mathcal{R}^N$, it follows that $s'_i \geqslant s_i \implies Z_S(s')_i \geqslant Z_S(s)_i$. It's worth noting that a stricter version of the reverse is also the case: $Z_S(s')_i > Z_S(s)_i \implies s'_i > s_i$, however in the case where $Z_S(s')_i = Z_S(s)_i$ we can not say anything about the ordering of s'_i and s_i.

While this may be a trivial property from the perspective of an RL agent, it makes the abstract states and transitions more understandable to human experts, particularly when the ground state transitions are easily understood to humans also. Having abstractions that are easily understandable can help give insight to the decisions that the agent's are making — which in turn can help improve or verify the agent's policies. This is obviously limited to a rather small subset of environments — for larger domains the complexity on both ground and abstract levels quickly grows beyond that which humans are capable of fully "grokking" or comprehending.

There isn't much that we can say about the distances between states. Firstly a distance function needs to be selected. Secondly, many commonly used distance functions do not scale particularly well into a large number of dimensions — they tend to define states as close to uniformly separated [8].

However, even for simple distance measures in low-dimensional environments, there is little useful information that we can deduce. Consider the scenario outlined in Figure 5.11, here there are four abstract states, representing each quadrant of this two dimensional grid. Ground states x and z are closer together than x and y by most sensible distance metrics despite x and y sharing an abstract state. It is clearly not the case then that sharing an abstract state ensures closer proximity to other ground states within that abstract state. We may be able to provide an upper bound for the distance between two ground states sharing an abstract state. For example, with the Euclidean distance function over $\mathcal{R}^n$, where abstract states have partition length l, we can say that for states s, s', $Z_S(s) = Z(s') \implies d(s, s') = \sqrt{\sum_{i=1}^{n}(s_i - s'_i)^2} \leqslant \sqrt{\sum_{i=1}^{n} l^2} = l\sqrt{n}$.

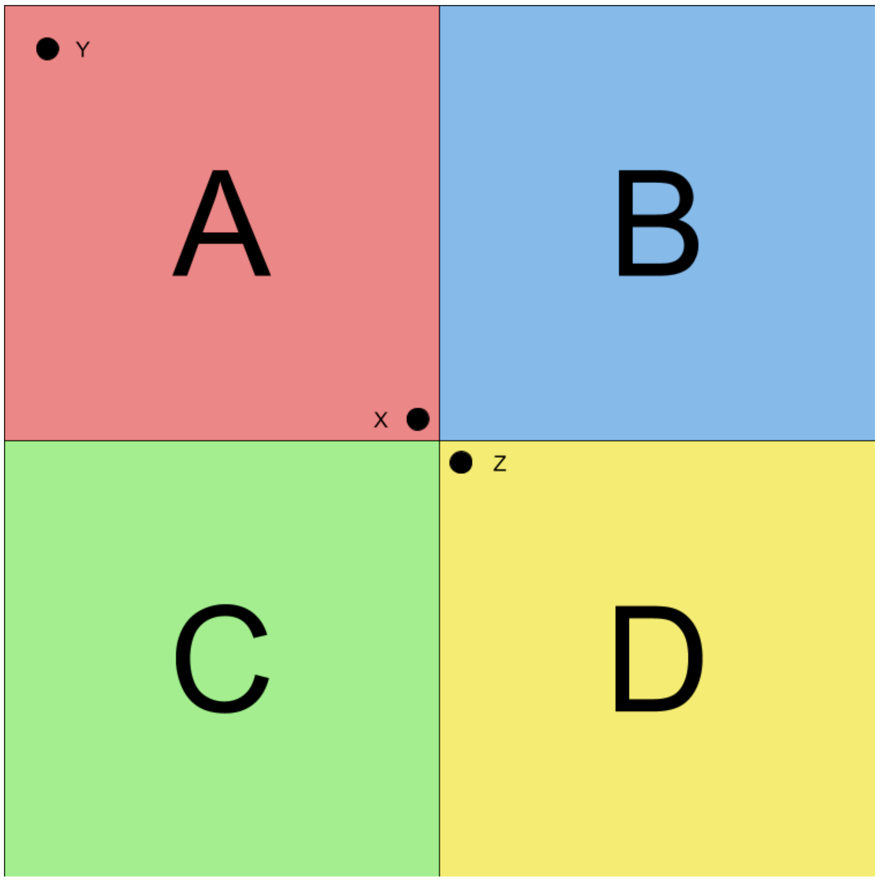


Figure 5.11: Simple domain demonstrating that two states, X and Y, sharing an abstract state can be further away from each other than a state Z in a separate abstract state.

However this provides only limited utility, particularly in higher dimensional space. This also raises the question of how distance between abstract states can be measured. Returning to Figure 5.11, what is the abstract distance between A and D? Does Euclidean distance make sense with discrete states? If we opt to take the Manhattan distance instead then the distance from A to D is two. But does it make conceptual sense for X and Z to be further away in the abstract state-space than X and anything in C? This problem only becomes more pronounced in higher dimensional space. There are no entirely satisfactory solutions to this, and ultimately we cannot make any useful guarantees about ground-distance or how this relates to abstract-distance.

On the other hand, LPSA does not have the state ordering property. First of all, the number of dimensions in both state-spaces usually do not match, So any attempt at preserving the ordering over each state dimension is simply folly. Even in the case where the state-spaces do share the same dimensionality, the autoencoders employ many levels of non-linear nodes in fully-connected manner, removing any guarantee of order preservation. It is therefore quite difficult to understand exactly what the state-abstraction function has learned to do. It will also suffer from any biases or distributional anomalies that are present in the training set — even though autoencoders are self-supervised, there is the question of how we obtain the states and the importance or frequency we assign each one. Both of these issues, understandability and data-set bias, are not only present in this thesis but a large part of neural network research as a whole. Work is being done to make deep learning more explainable and interpretable [52] [12] [4], but is still largely in its infancy.

The lack of the order-preserving property in LPSA, while making it less understandable to a human, allows LPSA to learn a more generalised state abstraction function. For our state abstraction function we want to map "similar" ground states to "similar" abstract states. By similar we mean in terms of agent trajectory and overall behaviour. We don't care how different the ground states are if they are capturing "similar" scenarios and behaviours. It is therefore necessary for the order preserving property to be removed to allow for potent state abstraction functions. This is the case precisely because similar scenarios can play out over very different areas of the ground state-space. For an intuitive example of this, in the Car Racing domain, shifting both the car and a straight track to the left of the screen changes nothing about the overall scenario or the quality of the agent's behaviour, however such a shift can cause the ground states to be very different — hundreds of pixels now take on many different values. Equally the reverse is also true, if two ground states are very near to each other, that does not imply a behavioural or scenario similarity. Again in the Car Racing domain, it only takes a few pixel changes in order to drastically change what the car will do in the next few steps — for example applying steering.

What we lose in human understandability in LPSA, we gain in the generalisability of our abstractions across the domain. This trade-off between understandability and generalisability is certainly not to be taken lightly, however, and will depend on the application and the extent to which it is safety-critical. For our purposes, we are aiming to demonstrate that state abstraction can be performed effectively by autoencoders and then utilised by reward shaping to improve and agent's learning speed. We therefore prioritise generalisability in the automated construction of state abstraction functions due to the additional abstraction potency provided.

5.14 Conclusion

In this chapter we have introduced a new method that extends on both MRL and UPSA. This new approach — LPSA — removes the limitation of constructing abstract states to be of a uniform size and aligned with each state dimension. LPSA is capable of improving an agent's learning performance through reward shaping by utilising auto-encoders to create an abstract state space. LPSA is capable of handling high dimensional state-spaces — vastly larger in size than MRL or UPSA. We saw that LPSA outperforms Vanilla DQN in the Car Racing domain — with the caveat that the shaping function and auto-encoder have been learned a priori. Nevertheless, this is a significant contribution, as it shows an entirely automatic process for an agent to create its own abstraction and for this abstraction to give a pronounced boost to early learning performance.

Chapter 6

Conclusion And Future Work

Now that the primary contributions of this thesis have been presented, we can conclude the thesis with a summary of the contributions, reflection upon the limitations of the approaches as well as potential avenues for future work.

6.1 Summary Of Contributions

This thesis has focused primarily on finding new methods to generate abstractions for use with reward shaping. Reducing the required external domain knowledge to create these abstractions has been an overarching goal throughout the research presented. This has been achieved in various ways throughout the research and will be highlighted in the coming subsections.

6.1.1 Comparison of uniform partitions and hand-labelled examples.

In Chapter 3, we investigated the performance of three agent types. The first agent type was a simple, unshaped agent applying the $Q(\lambda)$ algorithm. The second and third agent types utilised the same $Q(\lambda)$ algorithm, but were augmented with a shaped reward function constructed from an Abstract Markov Decision Process (AMDP). The AMDPs in the second and third agent type differed in that the second used an AMDP with a hand-labelled abstract states and the third utilised an AMDP where the states were formed from a uniform partitioning of the state-space. These agents were evaluated in the Flag Collection domain.

We found that in many cases, the AMDPs constructed according to uniformly partitioned ground state-spaces were able to compete with their hand-labelled counterparts in their shaping capabilities, occasionally outperforming them. This, of course, depended on the partitioning resolution, or "granularity" of the AMDP's abstract states. As expected, as the abstract states decrease in size, the more nuanced the shaping function can be — allowing more useful knowledge to be passed to the ground-level agent. Conversely, it took longer to compute the value function of the AMDPs using finer-resolution uniform partitions when compared to the hand-labelled AMDP. With the uniformly partitioned AMDPs there is a clear trade-off between the utility of the AMDP and the time required to solve the AMDP using Value Iteration. We found that an approximate balance can be struck where the uniformly partitioned AMDP requires roughly a similar time to solve and yields a similar benefit to the hand-labelled AMDP. If more time is available for solving the uniformly partitioned AMDP, then they can often give a better shaping benefit to the ground agent.

It is worth noting again that each uniformly partitioned AMDP took less time to solve than encoding the abstract state-space manually. This is where the real significance of the research lies. Encoding AMDP states for even simple environments such as the flag collection domain can take a non-trivial amount of time for a human expert to perform. Further, even the agents utilising very coarse-grained resolutions performed better than unshaped agents.

To expand briefly on the cost of labelling the abstract states of an AMDP, *Amazon's Mechanical Turk* charges (at the time of writing) $870 per 1000 image segmentations [2] (the closest analogy to partitioning a maze-based AMDP). Were we to attempt to extend this work to deal with environments with randomised room layouts and need thousands of AMDPs, the cost here would be far too large and it would be far too onerous to perform manually. Additionally, for more complex environments, such as the Car Racing domain we saw in Chapter 5, encoding AMDP states becomes not simply time consuming, but also difficult. It is unclear how to manually create a state abstraction function for such a domain, this is further complicated by the continuous nature of both the ground and abstract state-spaces. Some recent data-set corpora (independent of RL) have begun to include the cost of labelling [23] and the costs can be high. A more thorough analysis of data-set annotation cost can be found in [47]. Overall it is clear that automated solutions have an edge here in terms of speed, price and desirability.

6.1.2 Learning abstract transition functions

In Chapter 4 we devised an extension of Multi-Grid Reinforcement Learning (MRL) to incorporate deep learning. This extension utilised the same abstract transition function learning as MRL, and the abstract states were constructed in the same way (identically to the uniform partitionings).

Our extension differed from MRL by constructing an explicit abstract model of the environment and fully solving that to produce a static reward shaping function. Our approach outperformed both Vanilla DQN and MRL (extended naively to deep learning) over a range of continuous control domains.

Both MRL and our approach reduce the required domain knowledge to an abstract reward function. However, MRL's is unspecified in the general case, whereas our approach simply signifies an abstract goal and reduces the abstract reward function to a distance measure to the abstract goal in terms of the number of required abstract transitions.

Our approach is an improvement over MRL due to its performance in the deep learning scenario as well as its simplified and specified abstract reward function that utilises only a small amount of domain knowledge that is easy to obtain or often understood a priori.

6.1.3 Latent property state abstraction

Chapter 5 introduced the idea of using latent properties as an abstract method for reward shaping. This method we call latent property state abstraction (LPSA).

Here an auto-encoder was used to identify the required latent properties to represent a high-dimensional state in a low-dimensional abstraction. Utilising an auto-encoder in this manner allows the abstraction of far more complex domains that our previous methods.

We then demonstrated that agents interacting with the environment through these low-dimensional abstractions are capable of learning abstract policies that provide a good boost to learning speed when used with reward shaping. We saw that our reward shaping approach outperformed a DQN agent learning on the ground environment when the AMDP was available a priori within the Car Racing domain. Further, we saw that our method converges on a strong policy in fewer episodes than DQN as well and is comparable to the *World Models*[30] approach.

The significance of this research is that it demonstrates that the structure of the latent representation learned by the auto-encoder is suitable structure for performing RL with reward shaping in mind. Previous work, overviewed and analysed in Section 5.5, had shown a spattering of techniques for using latent state-spaces to increase learning speed. However, nothing previously had shown that the structure of the learnt latent representation is suitable for learning useful reward shaping value functions.

6.2 Limitations

Due to the nature of the thesis, many of the later chapters work on addressing limitations of the previous chapters. For example, a limitation of creating AMDPs from uniform partitions that was developed in Chapter 3 was that domain knowledge was utilised extensively to create a transition and reward function. This is the reason that this approach was not a definitive improvement over the existing $Q(\lambda)$ algorithm, but rather a proof of concept. However these limitations were addressed in Chapter 4 where we utilised MRL's approach to learning the transition function, as well as reducing the knowledge required for the reward function to an abstract goal state. Similarly, a limitation of the method from Chapter 4 was that the uniformity of the abstract state-space caused the number of abstract states to grow exponentially in the number of state dimensions, rendering the approach intractable for large-scale tasks. Again, this limitation was addressed in the following chapter, Chapter 5, where the concept of latent state-space abstractions was introduced.

The limitations we describe in this section then, will mostly refer to the limitations of the method outlined in Chapter 5 as well as any issues that affect each of the methods that have not been addressed elsewhere.

A limitation of this final method in particular is that the process for training an abstract network now takes a significant amount of time, particularly when compared to the prior methods of earlier chapters. Of course, this is to be expected, it is a more complex method, utilising deep learning rather than tabular methods and able to abstract tasks themselves are also correspondingly more complex. However, training both an auto-encoder to act as a state-abstraction function, as well as training DQN over this abstract environment can take longer than the speed up benefit provided by the shaping compared to vanilla DQN. However, producing a practical algorithm, ready for deployment was not the aim of this chapter or thesis as a whole; instead the aim was to demonstrate that latent properties of the state-space can be identified to reduce the state-space to a manageable size and then crucially that reward shaping based on the learned values of these abstract states could improve agent performance. All of this was also achieved in a fully automated manner. The only real knowledge required from a human expert is the number of latent state dimensions to allow the auto-encoder to identify. The amount required will clearly depend on the complexity of the environment, but will be easily empirically identifiable before any RL on the task begins.

The second limitation of this final approach is that the latent states require that the environment is amenable to auto-encoder reconstruction. This means that environments with very fine visual details that may be lost by the convolutional layers in the autoencoder may be less suited to this method than other environments. This is not to say that latent state-space abstraction based methods are not applicable to these finely detailed environments, however, these environments would instead possibly require an alternative method for identifying salient latent properties.

6.3 Generality

A quintessential requirement for many machine learning approaches is their generality; what range of domains do these techniques apply well to? It is well worth discussing the expectations of generality for the three core approaches proposed in this thesis. Broadly, as the thesis progresses and the methods become more complex, they also become more generalisable. If we consider the (unnamed) approach from Chapter 3 (which was intended more as a feasibility study of uniform state abstractions) we see limited applicability to environments other than Flag Collection — too much domain knowledge was used. However, it must be said that there are lots of "Flag-Collection-Like" domains that have practical use. Examples include warehouse or bomb disposal robotics, where there are clearly defined spatial state-spaces and knowledge of the goal or "flag" location is known a priori. Uniform Property State Abstraction from Chapter 4 extends the generalisability of the previous method by learning abstract transitions. This approach could, in theory, be applied to a far wider range of domains. The main limitations to this come from requiring a state-space with relatively few dimensions (due to using Value Iteration and the Curse of Dimensionality), as the environment needing to be easy to explore in order to build up an accurate AMDP. This limits the use of UPSA mostly to continuous-state control problems. Finally, Latent Property State Abstraction, from Chapter 5, is the most generally applicable. Not only can LPSA handle large numbers of state-dimensions (possibly pixel-based images), it also learns its own abstraction function, and leverages this with the existing environment to interpret ground transitions as abstract transitions. This allows LPSA to learn a strong abstract policy over the abstract states. The primary aspects of LPSA that would prevent its use on other RL domains would be, again, requiring a good level of prior exploration to build up the set of states for training the auto-encoder. There are some domains where this is simply infeasible. Additionally, some environments with pixel-based state-spaces may contain very small details that are easily lost by the auto-encoder. These domains may prove a challenge, though it may be possible to explore alternatives to auto-encoders for extracting the key information and then create an abstract state in a different manner, Overall, however, LPSA is applicable to a large variety of environments from across the RL spectrum, domains similar to Atari games are a prime example, as well as scenarios where an agent is interacting with the real world through a camera.

6.4 Future Work

As with any large body of research, the number of possible directions are seemingly exponential — the Curse of Dimensionality strikes again. Here we will outline some of the more promising areas for investigation.

Broadly speaking, the topics of most promise are abstraction width, abstraction height and abstraction integration. We will briefly outline each of these in the following sections.

6.4.1 Abstraction Integration

In order to fully reap the benefits of reward shaping from latent state-spaces the additional time taken caused by training the abstract agent needs to be addressed. The most obvious approach to doing this would be to reduce the time required to train the agent operating on the latent state-space. However this may not be feasible for complex tasks where the latent dynamics are also complex.

Another possible approach would be to interweave the training procedures for the ground agent and abstract agent. The two agents could either alternate control after a fixed number of episodes, or perhaps the abstract agent could learn entirely from the ground agent's experiences. Either way, some form of parallel learning could potentially give the required speed-up.

The ideal approach for achieving this isn't immediately clear, but once achieved would provide a deployable, efficient algorithm able to use its own abstraction to increase its learning rate with almost no human expertise.

6.4.2 Abstraction Height

As the tasks we utilise RL for become more and more complex, abstraction in some form becomes more crucial to learning a satisfactory policy in a timely manner. However, as the problem domains grow in complexity, the abstractions themselves must grow in complexity in order to capture the essence of the task and be useful abstractions. This can lead to the abstraction itself requiring a lot of computational time to attain a satisfactory abstract policy or values for use with shaping.

There are two possible solutions to this issue. The first is to increase the capabilities of the abstract representation. We saw an example of this during this thesis; the AMDPs based on uniform partitions of the ground state-space are less "abstractly powerful" than the latent state-space based abstractions. It is not immediately clear what changes to the latent state-space based abstractions would increase its abstracting power. On the other hand, stacking abstractions could be beneficial to the training process. In this approach, a very "coarse" abstraction would be trained and shape a "finer-grained", more complex abstraction. This second-level abstraction would then be used to shape the ground-level task. This could allow the training of inherently more complex tasks than current methods allow. This concept of abstraction stacking appeared in the paper [72] highlighted in Section 5.5.3, but integrating these "abstraction stacks" with LPSA could allow for some very potent agents.

Increasing what we have termed "abstraction height" (i.e., the height of an abstraction stack) could also be combined with the prior section on abstraction integration. If these multiple levels can learn their own policies and shape more grounded agents concurrently, then further time could be saved by removing the need for abstract "pre-training". This, of course, would need to be balanced against the shaping benefit that is lost by shaping before the abstract policy is complete.

6.4.3 Abstraction Width

The final proposed research direction is that of improving "abstraction width". By abstraction width we mean the breadth of domain variations that a single abstraction can be applied to and shape.

Over the course of this thesis, we have typically only considered domains that remain very similar each episode. For instance, the Flag Collection domains, Puddle World, Mountain Car and Catcher from Chapters 3 and 4 were essentially identical in every episode. Car Racing from Chapter 5 did consist of a racing track that was generated randomly from track segments, however there were only a few segments and they could almost be tackled independently of other parts of the track.

Many real-world domains are not so. Tasks can often have variations and differences that occur within each episode. An example of this within one of the smaller environments that we have used would be an alternate version of Mountain Car where the profile of the Mountain is randomised or the car has a varying mass or drive capabilities. To overcome these types of environments, the agent has to learn a more "general" solution to the problem based on its perceptions.

The work in this thesis could be extended to this type of problem. At present the abstractions constructed are verifiably useful for only the domain for which they were constructed in. However due to the nature of abstraction, it is possible that it may be easy to create abstractions that can be useful for solving many variations of the same task.

Slightly more formally, our tasks can be drawn from a distribution $\mathcal{D}$ and each episode draws a new environment d from this distribution: $d \sim \mathcal{D}$. We would then like our abstractions to encapsulate $\mathcal{D}$ and be useful for shaping agents that are also drawing their environments from $\mathcal{D}$ in the same manner.

Inevitably, some of the tasks will be more difficult than others. We also want to balance the range of tasks within $\mathcal{D}$ that an agent can perform in against the performance in each individual task. Psychometrics provides an interesting way of evaluating tasks of this sort. Item Response Theory (IRT) [20] outlines methods of statistically scoring each task d from $\mathcal{D}$ a "difficulty" score h, as well as evaluating an agent's performance over the task domain as difficulty increases. This is referred to as an Agent Characteristic Curve (ACC). IRT goes on to derive some notions of performance from this curve. The two most notable are generality and capability — these give notions of the range of domains the agent learns to perform over as well as the overall performance of the agent over the task distribution set as a whole.

IRT has recently seen use within machine learning [43] [44], where it has been used to evaluate the generalisation power of a large number of AI agents on Atari games as well as classifiers on various data-sets.

The further work to be proposed is an investigation of the trade-offs between generality and capability (when defined in this manner) against abstraction utility. A low-capability, high-generality abstraction could perhaps shape agents to be evaluated on $\mathcal{D}$ more effectively than one with low-generality and high-capability. On the other hand, high-generality abstractions used with shaping could lead to higher-generality ground agents.

This proposed research is potentially of import due to the current nature of very specialised learning algorithms that markedly worsen when even small changes to the environment are made [76]. Overcoming this overspecialisation of agents and algorithms is necessary for the eventual rise of Artificial General Intelligence.

6.5 Concluding Remarks

In recent years RL has become something of a juggernaut in the machine learning community. Exploration into the potential of this field is abound in every direction. Much effort from the community has gone into showing the power of imparting prior knowledge to agents, and that this knowledge can be based on abstractions of the task domain. We have seen more recent work beginning to automate this process of abstraction creation. As with many aspects of RL — and life in general — there is not always only one correct method to achieve something. It is among these more recent abstraction creation methods in which this body of work resides. We have tried to keep the methods intuitive and (relatively) simple, whilst also attempting to keep the abstractions that arise from these methods robust, useful and respecting a notion of an abstract hierarchy. The work here is hopefully to be used and improved upon by those inhabiting the future.

Bibliography

[1] Cambridge dictionary. https://dictionary.cambridge.org/dictionary/english/abstract. Accessed 2020-04-27.

[2] Mechanical turk pricing. https://www.mturk.com/pricing Last Accessed: 2021-05-10.

[3] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay, 2017.

[4] Plamen Angelov and Eduardo Soares. Towards explainable deep neural networks (xdnn), 2019.

[5] Kai Arulkumaran, Nat Dilokthanakul, Murray Shanahan, and Anil Anthony Bharath. Classifying options for deep reinforcement learning. *CoRR*, abs/1604.08153, 2016.

[6] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 2010.

[7] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 41–48, New York, NY, USA, 2009. ACM.

[8] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In Catriel Beeri and Peter Buneman, editors, *Database Theory — ICDT'99*, pages 217–235, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[9] Steven J. Bradtke and Michael O. Duff. Reinforcement learning methods for continuous-time markov decision problems. In *Advances in Neural Information Processing Systems*, pages 393–400. MIT Press, 1994.

[10] Steven J. Bradtke and Michael O. Duff. Reinforcement learning methods for continuous-time markov decision problems. In *Proceedings of the*

7th International Conference on Neural Information Processing Systems, NIPS'94, page 393–400, Cambridge, MA, USA, 1994. MIT Press.

[11] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[12] Shan Carter, Zan Armstrong, Ludwig Schubert, Ian Johnson, and Chris Olah. Activation atlas. *Distill*, 2019. https://distill.pub/2019/activation-atlas.

[13] Dario Cazzani. World-models-tensorflow. https://github.com/dariocazzani/World-Models-TensorFlow, 2018. Accessed 2020-05-13.

[14] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, Feb 1996.

[15] J.R. Doyle. Supervised learning in n-tuple neural networks. *International Journal of Man-Machine Studies*, 33(1):21–40, 1990.

[16] Kyriakos Efthymiadis, Sam Devlin, and Daniel Kudenko. Overcoming incorrect knowledge in plan-based reward shaping. *The Knowledge Engineering Review*, 31(1):31–43, 2016.

[17] Kyriakos Efthymiadis and Daniel Kudenko. A comparison of plan-based and abstract mdp reward shaping. *Connect. Sci*, 26(1):85–99, January 2014.

[18] Kyriakos Efthymiadis and Daniel Kudenko. Knowledge revision for reinforcement learning with abstract mdps. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '15, pages 763–770, Richland, SC, 2015. International Foundation for Autonomous Agents and Multiagent Systems.

[19] ElsanEl. Gym-puddle. https://github.com/EhsanEI/gym-puddle. Accessed: 2020-3-6.

[20] S. E. Embretson and S. P. Reise. *Item response theory for psychologists*. L. Erlbaum, 2000.

[21] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189 – 208, 1971.

[22] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Learning visual feature spaces for robotic manipulation with deep spatial autoencoders. *CoRR*, abs/1509.06113, 2015.

[23] Nicholas FitzGerald, Julian Michael, Luheng He, and Luke Zettlemoyer. Large-scale QA-SRL parsing. *CoRR*, abs/1805.05377, 2018.

[24] David Foster. World models. https://github.com/AppliedDataSciencePartners/WorldModels, 2018. 2020-012-16.

[25] Mohsen Ghafoorian, Nasrin Taghizadeh, and Hamid Beigy. Automatic abstraction in reinforcement learning using ant system algorithm. pages 9–14, 01 2013.

[26] Behzad Ghazanfari and Matthew E. Taylor. Autonomous extracting a hierarchical structure of tasks in reinforcement learning and multi-task reinforcement learning. *CoRR*, abs/1709.04579, 2017.

[27] Sertan Girgin, Faruk Polat, and Reda Alhajj. Improving reinforcement learning by using sequence trees. *Machine Learning*, 81(3):283–331, Dec 2010.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[29] Marek Grześ and Daniel Kudenko. Multigrid reinforcement learning with reward shaping. In Véra Kůrková, Roman Neruda, and Jan Koutník, editors, *Artificial Neural Networks - ICANN 2008*, pages 357–366, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[30] David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018.

[31] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 312–317, 1996.

[32] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Comput. Surv.*, 50(2), April 2017.

[33] Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. Vizdoom: A doom-based AI research platform for visual reinforcement learning. *CoRR*, abs/1605.02097, 2016.

[34] Ghorban Kheradmandian and Mohammad Rahmati. Automatic abstraction in reinforcement learning using data mining techniques. *Robotics and Autonomous Systems*, 57(11):1119 – 1128, 2009.

[35] Diederik P. Kingma and M. Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2014.

[36] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *CoRR*, abs/1906.02691, 2019.

[37] A.H. Klopf and Air Force Cambridge Research Laboratories (U.S.). *Brain Function and Adaptive Systems: A Heterostatic Theory.* Special Reports. Air Force Cambridge Research Laboratories, Air Force Systems Command, United States Air Force, 1972.

[38] Ramnandan Krishnamurthy, Aravind S. Lakshminarayanan, Peeyush Kumar, and Balaraman Ravindran. Hierarchical reinforcement learning using spatio-temporal abstractions and deep neural networks. *CoRR*, abs/1605.05359, 2016.

[39] Tejas D. Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Joshua B. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *CoRR*, abs/1604.06057, 2016.

[40] Andrew Levy, George Konidaris, Robert Platt, and Kate Saenko. Learning multi-level hierarchies with hindsight, 2017.

[41] Shie Mannor, Ishai Menache, Amit Hoze, and Uri Klein. Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 71–, New York, NY, USA, 2004. ACM.

[42] Bhaskara Marthi. Automatic shaping and decomposition of reward functions. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 601–608, New York, NY, USA, 2007. ACM.

[43] Fernando Martínez-Plumed and José Hernández-Orallo. Analysing results from ai benchmarks: Key indicators and how to obtain them. *ArXiv*, abs/1811.08186, 2018.

[44] Fernando Martínez-Plumed, Ricardo B.C. Prudêncio, Adolfo Martínez-Usó, and José Hernández-Orallo. Item response theory in ai: Analysing machine learning classifiers at the instance level. *Artificial Intelligence*, 271:18 – 42, 2019.

[45] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[46] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[47] James F. Mullen, Franklin R. Tanner, and Phil A. Sallee. Comparing the effects of annotation type on machine learning detection performance. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 855–861, 2019.

[48] Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. In *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016)*, Singapore, May 2016.

[49] Sanmit Narvekar, Jivko Sinapov, and Peter Stone. Autonomous task sequencing for customized curriculum design in reinforcement learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, Melbourne, Australia, August 2017.

[50] Sanmit Narvekar and Peter Stone. Learning curriculum policies for reinforcement learning. *CoRR*, abs/1812.00285, 2018.

[51] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[52] Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. The building blocks of interpretability. *Distill*, 2018. https://distill.pub/2018/building-blocks.

[53] Emilio Parisotto, Jimmy Ba, and Ruslan R. Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *CoRR*, abs/1511.06342, 2016.

[54] Martin L. Puterman. Chapter 8 markov decision processes. In *Stochastic Models*, volume 2 of *Handbooks in Operations Research and Management Science*, pages 331 – 434. Elsevier, 1990.

[55] Jette Randløv and Preben Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pages 463–471, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[56] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1278–1286, Bejing, China, 22–24 Jun 2014. PMLR.

[57] Melrose Roderick, Christopher Grimm, and Stefanie Tellex. Deep abstract q-networks. *CoRR*, abs/1710.00459, 2017.

[58] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[59] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988.

[60] G. Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[61] Mike Schuster. Better generative models for sequential data problems: Bidirectional recurrent mixture density networks. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 589–595. MIT Press, 2000.

[62] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[63] Satinder Singh, Richard Sutton, and P. Kaelbling. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22, 11 1995.

[64] Richard Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. 08 1996.

[65] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1 edition, 1998.

[66] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2 edition, 2018.

[67] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181 – 211, 1999.

[68] Richard Stuart Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, 1984. AAI8410337.

[69] Maxwell Svetlik, Matteo Leonetti, Jivko Sinapov, Rishi Shah, Nick Walker, and Peter Stone. Automatic curriculum graph generation for reinforcement learning agents, 2017.

[70] Nasrin Taghizadeh and Hamid Beigy. A novel graphical approach to automatic abstraction in reinforcement learning. *Robotics and Autonomous Systems*, 61(8):821 – 835, 2013.

[71] Norman Tasfi. Pygame learning environment. https://github.com/ntasfi/PyGame-Learning-Environment, 2016.

[72] Elise van der Pol, Thomas N. Kipf, Frans A. Oliehoek, and Max Welling. Plannable approximations to MDP homomorphisms: Equivariance under actions. *CoRR*, abs/2002.11963, 2020.

[73] Ziyu Wang, Bin Dai, David Wipf, and Jun Zhu. Further analysis of outlier detection with deep generative models. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 8982–8992. Curran Associates, Inc., 2020.

[74] Christopher Watkins. Learning from delayed rewards. 01 1989.

[75] Eric Wiewiora, Garrison Cottrell, and Charles Elkan. Principled methods for advising reinforcement learning agents. 07 2003.

[76] Chenyang Zhao, Olivier Sigaud, Freek Stulp, and Timothy M. Hospedales. Investigating generalisation in continuous deep reinforcement learning, 2019.

[77] E. Çilden and F. Polat. Toward generalization of automated temporal abstraction to partially observable reinforcement learning. *IEEE Transactions on Cybernetics*, 45(8):1414–1425, Aug 2015.