Goal Space Planning with Reward Shaping

by

Kevin Roice

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Statistical Machine Learning

Department of Computing Science

University of Alberta

Abstract

Planning and goal-conditioned reinforcement learning aim to create more efficient and scalable methods for complex, long-horizon tasks. These approaches break tasks into manageable subgoals and leverage prior knowledge to guide learning. However, learned models may predict inaccurate next states and have compounding errors over long-horizon predictions. This often makes background planning with learned models worse than model-free alternatives, even though the former uses significantly more memory and computation. Methods that plan in an abstract space, such as Goal-Space Planning, avoid these typical problems of models by background planning with models that are abstract in state and time. This thesis shows how potential-based reward shaping can propagate value and speed up learning with local, subgoal-conditioned models. We demonstrate the effectiveness of this approach in tabular, linear, and deep value-based learners, and study its sensitivity to changes in environment dynamics and the chosen subgoals.

Preface

The majority of this thesis is based on two co-authored publications: a workshop paper (Roice et al., 2024), and a journal paper (Lo et al., 2024). The work was done in collaboration with Parham Mohammad Panahi, Scott Jordan, Adam White, and Martha White.

The experiments were ran by Parham and myself. Scott provided technical guidance and suggested the use of potential-based reward shaping. Adam and Martha helped with several rounds of iteration on the empirical design and provided invaluable feedback on the results and writing. The theoretical results were a product of discussions between Scott and myself. This thesis and any mistakes in it are my own fault.

"We are stories, contained within the twenty complicated centimeters behind our eyes."

– Carlo Rovelli

"This too shall pass."

– King Solomon

Acknowledgements

I first extend my gratitude to my supervisors, Adam White and Martha White. Without their guidance, this project would not have been possible. Adam showed me what greatness looks like, and what it means to be a good scientist and writer. Martha is a wizard with algorithm development, iteration, and research guidance. Together, they were instrumental to my journey. Scott Jordan helped me tremendously when tackling my first research questions in reinforcement learning. This thesis would not be complete without his unwavering patience toward my horrendous coding, and his responsiveness no matter where he was on the planet.

My interactions with Marlos C. Machado, Michael Bowling, Yaozhong Hu, Levi Lelis, Erin Talvitie, Benjamin Eysenbach, and Rich Sutton were a privilege. They have led me to shape my views on research and life.

The last two years would not have been as exciting as it was without the support and friendship of the many kind souls I had the fortune of meeting in Edmonton: Aaron Hashim, Olya Mastikhina, Brett Daley, Esraa Elelimy, David Szepesvári, Haseeb Shah, Andrew Patterson, César Salcedo, Aditya Ramesh, Abrar Fahim, Kushagra Chandak, Aakash Sasikumar, Subhojeet Pramanik, Jordan Coblin, Deep Gandhi, Erfan Miahi, Han Wang, Edan Meyer, Vlad Tkachuk, Farzane Aminmansour, Fernando H. Garcia, Roy Nath, Golnaz Mesbahi, Jacob Adkins, Prabhat Nagarajan, Khurram Javed, Abhishek Naik, Gábor Mihucz, Rohini Das, Shibhansh Dohare, Manan Tomar, Adeel Gaija, Yazeed Mahmoud, Parham Mohammad Panahi, Diego Gomez, and Anna Hakhverdyan. They built me a second home away from home. I am thankful to Paul Saunders and his family for welcoming me with open arms on my first Christmas away from home.

I also thank Nishan Sen, Aliya Edwards, Deep Shah, Vigneshwar Hariharan, Oscar Lewis, Ryan Stewart, Harry Morris, Emma Lewington, Theodore Russell, Tania Gardašević, Adam Gillett, Andrea Biju, and Scott Ward for keeping me sane despite the distance (and time).

Most importantly, I could not make it through the last 23 years of life without Amma, Appa, Caroline, Ryan, and God.

Contents

Abstract ii

Preface iii

Acknowledgements v

List of Tables ix

List of Figures x

List of Algorithms xiv

1 Introduction 1

1.1 Planning in Reinforcement Learning 1

1.2 Contributions 4

1.3 Thesis Structure 5

2 Background 6

2.1 Reinforcement Learning 6

2.1.1 The Agent-Environment Problem Formulation 7

2.1.2 The Value Function 8

2.1.3 Learning and Approximating the Value Function 9

2.1.4 Planning . . . 10
2.2 Abstractions . . . 11
2.2.1 Abstracting State . . . 12
2.2.2 Abstracting Time . . . 13
2.3 Reward Shaping . . . 14
2.3.1 Reward Shaping in Computational RL . . . 14
2.3.2 Potential Based Reward Shaping . . . 15

3 Goal-Space Planning **17**
3.1 Subgoals . . . 17
3.2 Planning Abstractly . . . 20
3.2.1 Related Works . . . 20
3.2.2 Goal Space Planning . . . 22

4 Goal Space Planning with Reward Shaping **26**
4.1 Problems with naively using $v_{g^\star}$ . . . 27
4.2 $v_{g^\star}$ as a Potential . . . 28
4.3 Theoretical Results . . . 30
4.3.1 Shaping and Q-initialization Equivalence . . . 30
4.3.2 Shaping with $v^\star$ . . . 34

5 Experiments **36**
5.1 Tabular Results . . . 37
5.2 Linear Function Approximation Results . . . 39
5.3 Deep Reinforcement Learning Results . . . 42
5.4 Robustness to Accuracy of the Learned Models . . . 44
5.5 The Role of Subgoal Selection . . . 45

5.6 Subgoal Placement and the Region of Attraction 50
5.7 Comparison with other Potentials 53

6 Conclusions and Future Works **55**

References **57**

Appendix **65**

A Learning the Option Policies 65
B Learning the Subgoal Models 66
C Putting it all together 72
C.1 GSP Pseudocode 74
C.2 Optimizations for GSP using Fixed Models 76
D An Alternative way of using $v_{g^\star}$ 76
E Errors in Learned Subgoal Models 78
F Hyperparameter Sweeps 79

List of Tables

1 Mean squared error across state-to-subgoal models used in PinBall. 79

List of Figures

2.1 A schematic of the agent-environment interaction loop. At timestep t, the agent (left) takes action $A_t = a$, and the environment (right) in turn returns the next state and reward $S_{t+1} = s'$ and $R_{t+1} = r$ respectively. 7

2.2 Computing the potential difference used to shape the reward signal $R_t \to \tilde{R}_t$. The potential function Φ must assign a real value to all states in $\mathcal{S}$. 16

3.1 Using a subgoal structure in the PinBall domain (Konidaris and Barto, 2009), as done by Lo et al.. The agent begins with a set of subgoals (denoted in teal) and learns a set of subgoal-conditioned models. **(Abstraction)** Using these models, the agent forms an abstract MDP where the states are subgoals with options to reach each subgoal as actions. **(Planning)** The agent plans in this abstract MDP to quickly learn the values of these subgoals. **(Projection)** Using learned subgoal values, the agent obtains approximate values of states based on nearby subgoals and their values. These quickly updated approximate values are then used to speed up learning. 19

3.2 Original and abstract state spaces, taken from Lo et al. (2022). . . 23

5.1 The FourRooms domain. The blue square is the initial state, green square the goal state, and red boxes the subgoals. A subgoal's initiation set contains the states in any room connected to that subgoal. 37

5.2 These four plots show the action values after a single episode of updates for Sarsa with and without GSP and eligibility traces, i.e., $\lambda = 0.9$. Each algorithm's update is simulated from the same data collected from a uniform random policy. Each state (square) is made up of four triangles representing each of the four available actions. White squares represent states not visited in the episode. 38

5.3 This plot shows the average number of steps to goal smoothed over five episodes in the FourRooms domain. Shaded region represents 1 standard error across 100 runs. 39

5.4 Obstacles and subgoals for GridBall and PinBall. The larger circles show the initiation set boundaries. Subgoals are defined in position space. 40

5.5 The tile-coded value function after one episode in GridBall. Like Figure 5.2, the gray regions show the visited states that were not updated. The red circle is the main goal. 42

5.6 Five episode moving average of return in the GridBall over 200 episodes (left) and PinBall over 500 episodes (right). We performed 30 runs, and showed 1 standard error in the shaded region. All learners used linear value function approximation on their tile-coded features. 43

5.7 Investigating the behavior of GSP in the deep reinforcement learning setting in PinBall. (a) Following the format of Figure 5.6, we show the 20 episode moving average of steps to the main goal in PinBall. (b) Five episode moving average of steps to goal in PinBall for GSP with models trained with differing numbers of epochs. 44

5.8 Different subgoal configurations in the FourRooms environment with a lava pool. The purple square is the learner's starting location, the gray squares the walls, the orange squares the location of the lava pool, and the green square the goal location. The only difference between these figures are the red boxes, which indicate the states that are subgoals for that configuration. 47

5.9 This figure shows the average return (left) and average probability the agent will take the alternative path (right) from each episode. Shaded regions represent (0.05,0.9)-tolerance intervals (Patterson et al., 2020) over 200 trials. 48

5.10 The top row of this figure shows the value of $v_{g^\star}$ for each state before the lava pool, for each subgoal configuration. The second and third rows show the change in $v_{g^\star}$ after the first and 100^{th} episode, after the lava pool is introduced. 49

5.11 This figure shows the time the agent spends per episode in the bottom left and top right rooms. The lines convey the average % of time the agent spend and the shaded lines represent $(0.05, 0.9)$ tolerance intervals computed from 100 trials. 51

5.12 Five episode moving average of steps to goal in PinBall with different potential functions for $\Phi(s)$. We follow the format of Figure 5.7a. 54

A.1 Evaluation of PinBall option policies by average trajectory length. Policies were saved once they were able to reach their respective subgoal in undeer 50 steps, averaged across 100 trajectories. Subgoal 2 was the hardest to learn an option policy for, due to its proximity to obstacles. 66

B.1 State-to-Subgoal models learnt by neural models after 100 epochs. 68

B.2 Learning and using pre-trained models for GSP. 73

D.1 Five episode moving average of return in FourRooms, GridBall and PinBall. Curves are averaged over 30 runs where the shaded region is one standard error. 78

E.1 Model errors in State-to-Subgoal models used in GridBall. 78

F.1 Left Column: learning curves for five different step sizes, α, averaged over 30 runs. Right Column: sensitivity to different step sizes. Each dot represents the steps to goal averaged over 30 runs and 1000 episodes. The error bars show one standard error. The refresh rate τ increases with each row. 81

List of Algorithms

1 `Goal Space Planning` for Episodic Problems 25
2 `Planning`() 25
3 `Goal Space Planning with Reward Shaping` on DDQN 29
4 `MainPolicyUpdate`$(s, a, s', r, \gamma, a')$ 74
5 `Update_Models`(s, a, s', r, γ) 74
6 `Update_GSP_Models`(s, a, s', r, γ) 75

Chapter 1

Introduction

Imagine planning your vacation in terms of individual footsteps from your sofa to your destination and back. Or a driver who forgets how to drive every time they enter a new vehicle. It is not unusual for today's artificial intelligence to come up with absurd scenarios like this. This thesis addresses problems of this nature.

1.1 Planning in Reinforcement Learning

Reinforcement learning (RL) is a branch of artificial intelligence that studies how a learner can interact with their environment to find a behavior that maximizes the reward it accumulates. Model-based RL (MBRL) uses the concept of *planning*: the learner imagines how the environment would respond to different actions using a predictive model that it constructs and updates. These predictive models can either output the next possible state of the environment (a transition model) or some probability distribution over states (a distribution model). MBRL has achieved remarkable feats (Schrittwieser et al., 2020; Hafner et al.,

2023), but RL systems still encounter difficulties when the agent plans in the background over long horizons.

Planning with learned models in RL has the potential to improve the sample efficiency of learning. Planning provides a mechanism for the agent to imagine experiences, in the background as it interacts with the world. This lets it improve value estimates while making good use of each interaction sampled from the world. This hypothetical experience provides a stand-in for the real world; the agent can generate many experiences (transitions) in its head (via a model) and learn from those experiences. Dyna (Sutton, 1991) is a classic example of *background planning*: on each step, the agent generates several transitions according to its model, and updates its behavior with those transitions as if they were real experience.

Background planning allows learners to both adapt to the non-stationarity and exploit things that remain constant. In many interesting environments, like the real world or multi-agent games, the agents cannot learn or even represent the optimal way of behaving (a.k.a. the optimal policy). The agent can overcome this limitation, however, by using a model to rapidly update its value function (an estimate of how good each state is toward maximizing reward). Continually updating the agent's model of the world and re-planning allows it to adapt to changes in the world, helping its current decision-making. A model can capture stationary facts and adapt to non-stationaries in how the world works; planning can be used to reason about how the world works to produce better policies.

The promise of background planning is that we can learn and adapt value estimates efficiently, but many open problems remain to make it more widely useful. These include:

1. rolling out one-step predictions from transition models could lead the agent

to imagine invalid states,

2. distribution models can be complex, especially for learning probabilities in high-dimensional tasks,

3. planning itself can be computationally expensive for large state spaces.

One way to overcome these issues is to construct an abstract model of the environment and plan at a higher level of abstraction. In this thesis, we present a technique to plan abstractly that can help certain learners achieve their goals faster. A temporally-abstract model allows the agent to jump between states potentially alleviating the need to generate long rollouts. It also predicts the rewards it receives and the probabilities of where it ends up after such a jump.

However, there are issues with abstract models. Though planning can be shown to be provably more efficient (Mann et al., 2015), the resulting policy is sub-optimal, restricted to going between landmark states. This sub-optimality issue forces a trade-off between increasing the size of the abstract problem (to increase the policy's expressivity) and increasing the computational cost to update the value function. In this thesis, we investigate an abstract model-based planning method that can quickly propagate changes over the entire state space, and do not limit the optimality of learned policy.

The alternative strategy, which we shall explore, is to use the policy computed from the abstract problem to guide the learning process in solving the original problem. More specifically, the purpose of the abstract problem is to propagate value quickly over an abstract state space and then transfer that information to a value function estimate in the original problem. This approach has two main benefits: i) the smaller, abstract space can propagate value quicker and with a smaller computational cost than if we used the original space, and, ii) the learned

policy is not limited to the abstract space. Overall, this approach increases the agent's ability to learn and adapt to changes in the environment quickly.

1.2 Contributions

This work studies an enhancement of an existing abstract planning algorithm, named Goal-Space Planning (GSP) (Lo et al., 2022). We analyze the properties of this algorithm on a much wider set of learners and environments and find that modifying it to use a technique called reward shaping improves the number of instances where the algorithm can be used.

Specifically, in this thesis we:

1. introduce and analyze Goal-Space Planning with Reward Shaping, a background planning formalism for the general online RL setting,
2. carefully study its benefits in terms of value propagation, and in what instances it speeds up learning,
3. measure the robustness of this method to model accuracy. We report relatively fewer epochs of training is needed to reap the benefits of Goal Space Planning,
4. we find reward shaping to be a much more effective approach to incorporating this abstract model into a learner, making the algorithm amenable to a wider variety of learners than before.

1.3 Thesis Structure

The next chapter of this thesis will overview the relevant background needed to understand Goal-Space Planning with Reward Shaping: namely the problem setting we use to model agent-environment interactions, learning, and planning techniques. Next, Chapter 3 overviews the vanilla Goal-Space Planning algorithm. Then Chapter 4 introduces the modifications we make to this algorithm and their implications. Chapter 5 empirically examines Goal-Space Planning with Reward Shaping to identify and better understand what components of the algorithm are crucial for the speed-up in learning and value propagation. Finally, the last chapter concludes this work and highlights avenues for future works to build upon.

Chapter 2

Background

This chapter will provide the necessary background to understand the context and motivation for Goal-Space Planning with Reward Shaping. We first cover some of the fundamental concepts of the reinforcement learning setting. We then move on to the more general notion of abstraction, emphasizing its importance in simplifying complex problems and enabling efficient learning. Following this, we focus on planning and how it can help the decision-making processes. Finally, we touch upon the concept of shaping and its applications to facilitate learning. This sets the stage for the reward-shaping variant of Goal-Space Planning which we analyse in our experiments.

2.1 Reinforcement Learning

There are many ways to model the sequential decision-making problem; we will overview reinforcement learning (RL), the problem-setting we will use throughout this work.

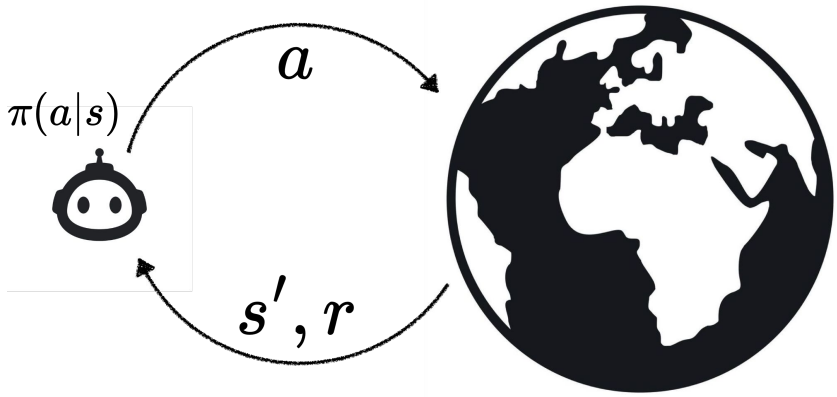


Figure 2.1: A schematic of the agent-environment interaction loop. At timestep t, the agent (left) takes action $A_t = a$, and the environment (right) in turn returns the next state and reward $S_{t+1} = s'$ and $R_{t+1} = r$ respectively.

2.1.1 The Agent-Environment Problem Formulation

We consider the setting where an agent can interact with its environment by taking actions and receiving observations.

We model the environment as a discrete-time, finite *Markov Decision Process* (MDP) (Puterman, 2014). An MDP is formalised as a 4-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$. $\mathcal{S}$ is the state space and $\mathcal{A}$ is the action space. The reward function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$, and the transition probability, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \Delta(\mathcal{S})$, describe the expected reward and probability of transitioning to a next state, for a given state and action. On each discrete timestep t, the agent selects an action A_t in state S_t, the environment transitions to a new state S_{t+1} and emits a scalar reward R_{t+1} [1]. We visualize this interaction loop in Figure 2.1. We describe a behavior using

[1]Throughout this work, we will use capitalized italic letters to represent random variables. These typically have a time index in the subscript. Lowercase letters are used to indicate the values a random variable can take. For example, s and s' in Figure 2.1 are the specific states that are the outcomes of S_t and S_{t+1}.

a policy $\pi : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{A})$. The agent's objective is to find a policy that maximizes expected *return*, the future discounted sum of reward $G_t \doteq R_{t+1} + \gamma G_{t+1}$, for some constant discount factor $\gamma \in [0, 1]$.

2.1.2 The Value Function

Since the goal of the agent is to find a policy that maximizes its return in expectation, it is crucial to evaluate how good each state or state-action pair is in terms of future rewards. The agent needs a way to reason about how 'valuable' it is to be in some state, in terms of accumulating future reward. The *value function* encodes this.

For some given policy π, its value function $v_\pi : \mathcal{S} \mapsto \mathbb{R}$, is defined as the expected, discounted sum of rewards the agent would get if it followed policy π from some state s,

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s], \tag{2.1}$$

$\forall\ s \in \mathcal{S}$. The expectation operator $\mathbb{E}_\pi[\cdot]$ is important. Agents reason in terms of expected return due to stochasticity (randomness) in the environment or even in the agent's decision-making itself, as we shall later see. Once the agent is in some state, it must also be able to reason about what action is favorable in terms of return. To do so, value functions can also be defined over the joint state and action space. We call this the *action-value function* $q_\pi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$,

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a]. \tag{2.2}$$

2.1.3 Learning and Approximating the Value Function

The process of computing a value function to evaluate how good a given policy is referred to as the *prediction* problem.

For finite state spaces, if we know the transition dynamics $p \in \mathcal{P}$ of the MDP, we can use dynamic programming (DP) techniques like *value iteration* to iteratively improve the value function and policy and eventually converge to $v^\star$ (Bertsekas, 1987). However, since we usually do not know p (the exact model of the environment), we use each experience from interacting with the environment $\langle S_t, A_t, R_{t+1}, S_{t+1}, \gamma_{t+1} \rangle$ [2] to learn $Q(s, a)$[3], an estimate of q_π from Equation 2.2. For example, the *Q-learning* algorithm (Watkins, 1989) learns $q^\star$ by applying the Bellman optimality equation in the update rule,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma_{t+1} \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t) \Big].$$

There are two things to notice about this method. Firstly, it does not rely on any model of the environment – it is a *model-free* method. Secondly, the update rule moves the $Q(S_t, A_t)$ estimate in the direction of the term in the square brackets by some scalar stepsize α. We can get different algorithms by swapping out the term in the square brackets. The *Sarsa* algorithm (Rummery and Niranjan, 1994) moves action-value estimates in the direction $\Big[R_{t+1} + \gamma_{t+1} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \Big]$. These square-bracketed terms are known as temporal difference errors, and these algorithms are called *temporal difference* (TD) learning algorithms, as we use a future state's value estimate to update the current state's estimate (a process referred to as bootstrapping).

[2] Following the convention of White (2017), we include the discount factor, γ_{t+1}, in each transition to serve two roles: discounting and episode termination, if any.

[3] Somewhat abusing notion, Q is a table that stores action-value estimates for each state-action pair.

Since the state and/or action spaces are often too large to enumerate as a table of estimates, value estimates are typically maintained using some function approximator, $\hat{q}_\pi$. In RL, this is almost always a parameterized function $\hat{q}_\pi(\cdot, \cdot; \mathbf{w})$[4], like a linear model or an artificial neural network. In the parameterized setting we use $\mathbf{w}$ to denote the parameters of our function approximation. A linear action-value function approximation would take the form $\hat{q}_\pi(s, a; \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(s, a)$, where $\boldsymbol{\phi}(s, a)$ is a *feature vector* used to represent a state-action pair.

Most TD learning algorithms have semi-gradient counterparts for the parameterized setting.[5] For example, semi-gradient Sarsa:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \Big[R_{t+1} + \gamma_{t+1} \hat{q}_\pi(S_{t+1}, A_{t+1}; \mathbf{w}) - \hat{q}_\pi(S_t, A_t; \mathbf{w}) \Big] \nabla_{\mathbf{w}} \hat{q}_\pi(S_t, A_t; \mathbf{w}).$$

2.1.4 Planning

In the field of artificial intelligence, agents that learn and plan using some model of the world are referred to as *model-based* learners. In the RL setting, *planning* refers to any computational process that uses a model to improve the agent's policy (Sutton and Barto, 2018).

The TD learning algorithms described in the previous section learn using experience directly from the environment. These model-free learning algorithms have no direct estimate of the environment transition dynamics $p(s'|s, a)$. DP methods are model-based, as they require the true dynamics for their convergence guarantees.

[4]We use the semi-colon (;) to separate a function's arguments from its parameters. The parameters appear on the right of the ;. We will denote vectors with lower-case boldface letters throughout this thesis.

[5]We call these *semi*-gradient methods as we do not fully apply the chain rule when differentiating the least squares error (Barnard, 1993). See Sutton and Barto (2018) for an in-depth overview of ignoring the effect of our parameters on the target.

Other model-based approaches either try to learn or are given an approximation of how the environment works. There are many ways this can be done. The simplest is a state-transition model, which predicts S_{t+1} given some $S_t = s$. Models can either be given to an agent, or learned from its own experience of what happens after being in state s.

Dyna (Sutton, 1991) is a family of MBRL algorithms that use real-world experiences to learn both value estimates and estimates of the world dynamics (i.e. a world model), while also carrying out background planning with the learned model.

Besides transition models, some models can also make reward predictions. These models answer questions like: given the agent is in state s and ends up in state s', how much return can it expect to receive? Notice how such models would act on arbitrary time scales: s and s' could be really close, or really far away in time under the behavior policy. Such models are amenable to abstractions in time, which we will review in Section 3.2.2.

2.2 Abstractions

In complex environments like the real world, managing the granularity of both states and actions is crucial for efficient decision-making and learning. The world is much bigger and more complex than the agent (Javed and Sutton, 2024). Reasoning in terms of individual states and timesteps, as in Figure 2.1, becomes cumbersome for our computationally bounded agent. Abstraction is essential for an intelligent system to transform raw observations into more useful and interpretable constructs for decision-making.

2.2.1 Abstracting State

In an MDP, the state is sufficient to fully describe the configuration of the environment. Depending on how complex the environment is, this often results in the state carrying a lot of information – most of which may be redundant to our decision-making agent seeking to maximize return. Consider Atari 2600 games — a common benchmark for deep reinforcement learning algorithms (Bellemare et al., 2013). Each second, an agent playing these games would receive 60 frames of 210×160 pixels that can each take on 128 possible colors. That totals to over 6 million scalars each second. These raw pixel values along with the internal state of the simulator are often far too unstructured to reason with – decision-making becomes intractable. A very rewarding state and a very bad state may have nearly identical pixel values. It becomes very difficult to reason about actions and have a value function defined at this level of granularity.

Determining what information to keep and what to ignore is an open problem in the field of representation learning – the theory of which is beyond the scope of this thesis [6]. In many cases, we can define a value function over a more abstract space than $\mathcal{S}$. The simplest way to do so is *state aggregation*: treating similar states to be the same. *Tile coding* (Sutton and Barto, 2018) takes this one step further by partitioning $\mathcal{S}$ into non-overlapping tiles, to form a tiling. Multiple of these tilings are offset and stacked on top of each other, and used to represent a state[7]. Artifical neural networks can also be used for state abstraction, as is commonly done in deep RL. We tested Goal-Space Planning with each of these forms of state abstraction in this thesis.

[6] Refer to Abel (2022) for a more detailed study on the theory of state abstraction.

[7] Kumaraswamy (2022) provides an excellent description of how tiling coding is set up.

2.2.2 Abstracting Time

Instead of having to reason about what action to take at every single time step, there are many situations where an agent would want to perform a routine of actions spanning multiple timesteps.

One way to represent such temporal abstractions is using the *options* framework (Sutton et al., 1999). Formally, these are defined as a policy, π_ω, coupled with some initiation set $\mathcal{I}_\omega \subseteq \mathcal{S}$, and termination probability $\beta_\omega : \mathcal{S} \mapsto [0, 1]$. We represent options with the triple $\omega = \langle \mathcal{I}_\omega, \pi_\omega, \beta_\omega \rangle$, and Ω denotes the space of possible options. Options are a generalization of single timestep actions (sometimes called *primitive actions*). They can be thought of as executing a temporally extended action π_ω from some state $s \in \mathcal{I}_\omega$. For all timesteps this option is active, we sample actions according to the option policy π_ω. This allows the agent to "jump" between two states that may be distant in time (Wan and Sutton, 2022). These could encode skills such as a robot arm lifting a cup, or a person travelling to Italy, while abstracting away the lower-level actions like the robot's voltage signals or human muscle twitches. Our usual learning algorithms can be run on an augmented space of primitive actions and useful options, $\mathcal{A} \cup \Omega$.

Instead of simply executing options[8], their true power becomes apparent when the agent uses them to reason in time. We can construct *option models* to do so. This is effective for long-horizon planning, and reasoning about goals in the distant future, as we shall see in the next section. Most recently Sutton et al. (2023) showed how options that respect the reward from the environment can be useful for planning. The discovery and use of such options for planning has been referred to as the *STOMP progression* (**S**ub**T**ask, **O**ption, **M**odel, **P**lanning), and this thesis can be regarded as one instance of this framework. This "temporally-

[8]Sometimes called the *call-and-return* execution model of options (Veeriah, 2022).

abstract cognitive structure" made with the options framework facilitates reasoning at a higher level of abstraction (Sutton et al., 2023). When computation is limited, this allows the agent to reason and compose more intricate and reusable behaviors than if we were to reason in terms of primitive actions.

2.3 Reward Shaping

Shaping is a form of operant conditioning, a type of learning where the reinforcing signal depends on the learner's behavior (Sutton and Barto, 2018). The causality between action and reward in operant conditioning provides a core component in how many reinforcement learning algorithms learn good policies.

Despite the name, shaping in psychology [9] is distinct from reward shaping in RL. This section will describe a form of reward shaping, which we later show to be invaluable to improving GSP.

2.3.1 Reward Shaping in Computational RL

While good shaping can be a powerful technique for learning systems (Peterson, 2004), it became apparent that ill-defined shaping terms can lead to very undesirable policies. Shaping was first investigated in computational reinforcement learning by Randløv and Alstrøm (1998) when training an agent to ride a bicycle in a simulator. In this setting, an agent drove a bicycle and tried to reach a fixed goal state. There was a reward of -1 per step, but the designers chose to add a positive bonus every time the agent moved towards the goal. Amusingly, the agent found that it would get much more return per episode if it moved in a circle of the right size, so the positive reward of moving towards the goal outweighed

[9] We refer the reader to Skinner (1958) for an in-depth treatment of shaping in psychology.

the -1 per step. Clearly, unconstrained additions to the reward signal can lead to behaviors far from desirable. In Randløv and Alstrøm's case, the optimal policy changed. We call this the policy variance problem (Behboudian et al., 2022) - the transformation we applied to the reward signal changed the optimal policy in the new MDP.

2.3.2 Potential Based Reward Shaping

This problem was fixed a year later, with a concept called *Potential Based Reward Shaping* (PBRS) (Ng et al., 1999). The motivation was to find a way to constrain the shaping term we add to the reward, to preserve the optimal policy and rankings of policies. PBRS constrains what we can add to the reward signal. This aims to address the policy variance issue we saw in the bicycle example. In order to do this, we first need to have a potential function $\Phi : \mathcal{S} \mapsto \mathbb{R}$ over the entire state space. Φ can be any function that assigns a real value to every state in the state space - creating a landscape over the state space that encodes some domain knowledge. For example, the value functions can be potential functions.

This shaping constraint can be understood by considering potential energy from classical mechanics, as the name suggests. If we have an object in 3D space, and we move it through some closed loop such that it exactly returns to its initial state (position and velocity), it should not have gained any energy. This prevents situations like Randløv and Alstrøm's problem of getting infinite return.

Φ is used to shape the reward signal as follows:

1. When the agent transition from state S_{t-1} to S_t, we call Φ at each of these states.

2. We discount the potential of the next state, and add the difference $\gamma\Phi(S_t) -$

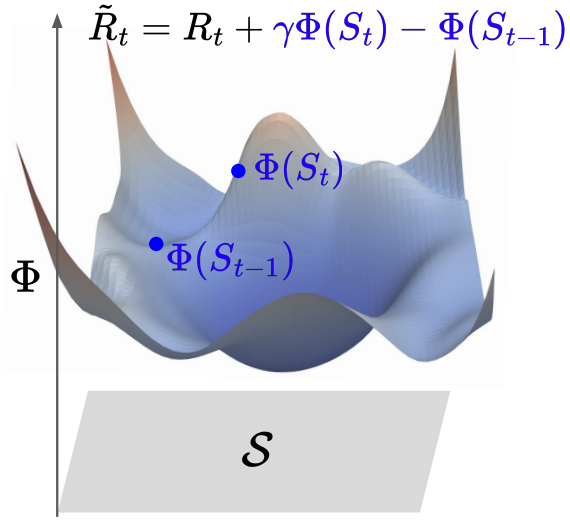


Figure 2.2: Computing the potential difference used to shape the reward signal $R_t \rightarrow \tilde{R}_t$. The potential function Φ must assign a real value to all states in $\mathcal{S}$.

$\Phi(S_{t-1})$ to the reward signal.

In this thesis, we exploit this reward transformation in the context of model-based reinforcement learning. Specifically, we set Φ, to be a learned model of the environment's reward and transition dynamics, and analyze how the policy invariance[10] properties of shaping can help a larger variety of learners (tabular, linear function approximation, and deep RL algorithms) to use the local models of GSP.

[10]Refer to Theorem 1 of Ng et al. (1999) for a proof of how such a reward transformation preserves the optimal policy.

Chapter 3

Goal-Space Planning

This chapter sets the groundwork for the Goal-Space Planning algorithm. We first formalize the notion of subgoals in a state space, and then move on to using this structure to create an abstract MDP to plan with. We then give an overview of abstract planning techniques before directing our attention to the vanilla Goal-Space Planning (GSP) algorithm (Lo et al., 2022).

3.1 Subgoals

Finding a subgoal structure to an underlying problem involves a combination of state and time abstractions described in Section 2.2. While discovery is a fascinating problem in itself, this thesis will focus on algorithms to leverage given subgoal structures.

In this work, we formalize this subgoal structure by defining a *goal-space*, $\mathcal{G}$. Despite its name, we treat $\mathcal{G}$ as a finite, discrete set of subgoals, $\mathcal{G} = \{g_1, g_2, \ldots, g_n\}$.

Similar to options (Sutton et al., 1999), each subgoal has two components: a set of goal states, and a set of initiation states. These two sets are defined by the indicator functions

$$m(s, g) = \mathbb{1}_{s \in g},$$
$$d(s, g) = \mathbb{1}_{s \in \mathcal{I}_g},$$

with m specifying the states in the subgoal (this could be something as simple as state aggregation), and d specifying states in the subgoal's initiation set. We say that a state s is a member of subgoal g if $m(s, g) = 1$, and we only reason about reaching a subgoal g from states s in its initiation set $\mathcal{I}_g$ (such that $d(s, g) = 1$). This constraint is key for locality: to learn and reason about a subset of states for a subgoal. We assume the existence of a (learned) *initiation function* d to indicate when the agent is sufficiently close in terms of reachability. We discuss some approaches to learning this initiation function in Appendix B.

We expect a complete, general-purpose agent to discover these subgoals on its own, including how to represent these subgoals to facilitate generalization and planning. As mentioned before, in this thesis, we first focus on how the agent can leverage reasonably well-specified subgoals.

While subgoals could represent a single state, they can also describe more complex conditions that are common to a group of states. For example, g could correspond to a situation where both the front and side distance sensors of a robot report low readings—what a person would call being in a corner [1]. As another example, in the Abstraction step of Figure 3.1, we simply aggregate certain states into nine subgoals—which correspond to regions with a small radius. For

[1]For the curious reader, the idea of identifying subgoals as states where certain observation signals or features are high is a concept called *feature attainment*. Sutton et al. (2023) first formally coined this term in the reinforcement learning setting.

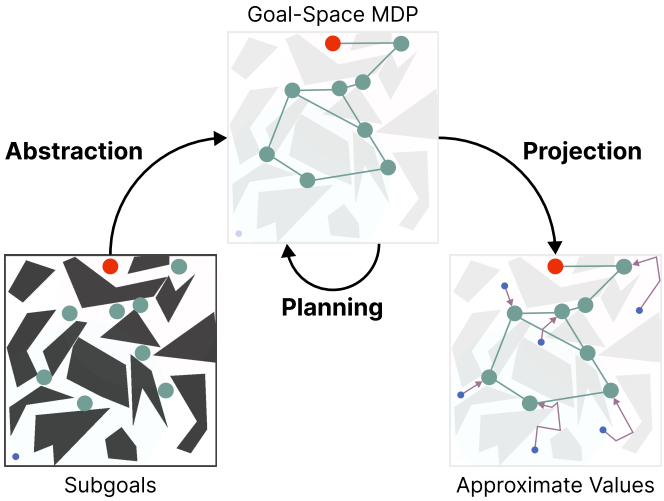


Figure 3.1: Using a subgoal structure in the PinBall domain (Konidaris and Barto, 2009), as done by Lo et al.. The agent begins with a set of subgoals (denoted in teal) and learns a set of subgoal-conditioned models. **(Abstraction)** Using these models, the agent forms an abstract MDP where the states are subgoals with options to reach each subgoal as actions. **(Planning)** The agent plans in this abstract MDP to quickly learn the values of these subgoals. **(Projection)** Using learned subgoal values, the agent obtains approximate values of states based on nearby subgoals and their values. These quickly updated approximate values are then used to speed up learning.

a concrete example, we visualize subgoals for our experiments in Figures 5.1 and 5.4. Essentially, our subgoals define a new state space in an abstract MDP, and these new abstract states (subgoals) can be represented in different ways, just like in regular MDPs.

In practical model-based reinforcement learning, it is often unnecessary to capture every detail of the environment (Arumugam and Van Roy, 2022; Rodriguez-Sanchez and Konidaris, 2024). Instead, designing smaller, abstract MDPs that focus on essential behaviors can facilitate effective planning while respecting the computational limitations of the agent.

3.2 Planning Abstractly

Using transition models that make one-step predictions can be problematic. These models have several issues. Transition models may predict invalid states. Errors in one-step models can compound over time when rolling out the model for long-horizon predictions. A variety of approaches have been developed to handle issues with learning and iterating one-step models. Several papers have shown that using forward model simulations can produce simulated states that result in catastrophically misleading values (Jafferjee et al., 2020; van Hasselt et al., 2019; Lambert et al., 2022).

3.2.1 Related Works

An emerging trend is to avoid approximating the true transition dynamics, and instead learn dynamics tailored to predicting values on the next step correctly (Farahmand et al., 2017; Farahmand, 2018; Ayoub et al., 2020). This trend is also implicit in the variety of techniques that encode the planning procedure into neural network architectures that can then be trained end-to-end (Tamar et al., 2016; Silver et al., 2017; Oh et al., 2017; Weber et al., 2017; Farquhar et al., 2018; Schrittwieser et al., 2020). We similarly attempt to avoid issues with iterating models, but do so by considering a different type of model.

Current deep model-based RL techniques plan in a lower-dimensional abstract space where the relevant features from the original high-dimensional experience are preserved, often referred to as a *latent space*. MuZero (Schrittwieser et al., 2020), for example, embeds the history of observations to then use predictive models of values, policies, and one-step rewards. Using these three predictive models in the latent space guides MuZero's Monte Carlo Tree Search without

the need for a perfect simulator of the environment. Most recently, DreamerV3 demonstrated the capabilities of a discrete latent world model in a range of pixel-based environments (Hafner et al., 2023). There is growing evidence that it is easier to learn accurate models in a latent space.

Temporal abstraction has also been considered to make planning more efficient, through the use of hierarchical RL and/or options. MAXQ (Dietterich, 2000) introduced the idea of learning hierarchical policies with multiple levels, breaking up the problem into multiple subgoals. A large literature followed, focused on efficient planning with hierarchical policies (Diuk et al., 2006) and using a hierarchy of MDPs with state abstraction and macro-actions (Bakker et al., 2005; Konidaris et al., 2014; Konidaris, 2016; Rodriguez-Sanchez and Konidaris, 2024). [2]

There has been some work using options for planning using only one level of abstraction and restricting planning to the abstract MDP. Hauskrecht et al. (2013) proposed to plan only in the abstract MDP with macro-actions (options) and abstract states corresponding to the boundaries of the regions spanned by the options, which is like restricting abstract states to subgoals. Bagaria et al. (2021) discover skills to construct discrete graph abstractions of continuous state and action spaces with subgoal nodes and option policy edges. The most similar to our work is LAVI, which restricts value iteration to a small subset of landmark states (Mann et al., 2015).[3] These methods also have similar flavors to using a hierarchy of MDPs, in that they focus planning in a smaller space and (mostly) avoid planning at the lowest level, obtaining significant computational speed-ups.

[2] See Gopalan et al. (2017) for an excellent summary.

[3] A similar idea to landmark states has been considered in more classical AI approaches, under the term bi-level planning (Wolfe et al., 2010; Hogg et al., 2010; Chitnis et al., 2022). These techniques are quite different from Dyna-style planning—updating values with (stochastic) dynamic programming updates—and so we do not consider them further here.

3.2.2 Goal Space Planning

One such approach for planning with an abstract model of the environment is Goal-Space Planning (GSP) (Lo et al., 2022), which this thesis will build upon. In this section, we will first outline the technical definitions of subgoal-conditioned models, and then overview how they are used in vanilla GSP.

For planning and acting to operate in two different spaces, Lo et al. (2022) defined four models: two used in planning over subgoals (subgoal-to-subgoal) and two used to project these subgoal values back into the underlying state space (state-to-subgoal). Figure 3.2 visualizes these two spaces.

The state-to-subgoal models are $r_\gamma : \mathcal{S} \times \bar{\mathcal{G}} \to \mathbb{R}$ and $\Gamma : \mathcal{S} \times \bar{\mathcal{G}} \to [0, 1]$, where $\bar{\mathcal{G}} = \mathcal{G} \cup \{s_\perp\}$ if there is a terminal state, $s_\perp$, (episodic problems) and otherwise $\bar{\mathcal{G}} = \mathcal{G}$.

An option policy $\pi_g : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{A})$ for subgoal g starts from any s in its initiation set, and terminates in g — in $\tilde{s}$ where $m(\tilde{s}, g) = 1$. The reward-model $r_\gamma(s, g)$ is the discounted rewards under option policy π_g:

$$r_\gamma(s, g) = \mathbb{E}_{\pi_g}\left[\sum_{k=0}^{\infty}\left(\prod_{k'=0}^{k}\gamma_{t+k'+1}\right)R_{t+k+1}\middle|S_t = s\right], \tag{3.1}$$

where the discount is zero upon reaching subgoal g.

The discount-model $\Gamma(s, g)$ reflects the discounted probability of reaching subgoal g starting from s, in expectation under option policy π_g

$$\Gamma(s, g) = \mathbb{E}_{\pi_g}\left[\sum_{k=0}^{\infty}\left(\prod_{k'=0}^{k}\gamma_{t+k'+1}\right)m(S_{t+1}, g)\middle|S_t = s\right]. \tag{3.2}$$

These state-to-subgoal models will only be queried for (s, g) where $d(s, g) > 0$: they are local models. Notice how Equation 3.1 and Equation 3.2 have an analogous form to the value function defined in Equation 2.1. When we generalize the value function to look at discounted sums of scalars which are not just the reward signal, they become *general value functions* (GVFs) (Sutton et al., 2011). They can still be learned using the usual value function learning algorithms, like TD learning.

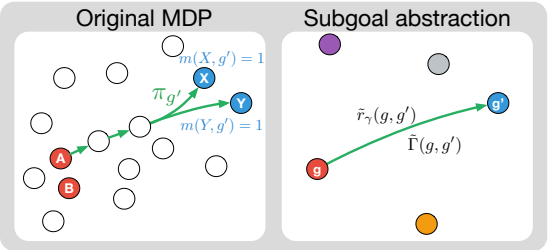


Figure 3.2: Original and abstract state spaces, taken from Lo et al. (2022).

The state-to-subgoal models are used to define subgoal-to-subgoal models, $\tilde{r}_\gamma : \mathcal{G} \times \bar{\mathcal{G}} \mapsto \mathbb{R}$ and $\tilde{\Gamma} : \mathcal{G} \times \bar{\mathcal{G}} \mapsto [0, 1]$. [4] For each subgoal $g \in \mathcal{G}$, we average $r_\gamma(s, g')$ for all s where $m(s, g) = 1$.

$$\tilde{r}_\gamma(g, g') \doteq \frac{1}{z(g)} \sum_{s:m(s,g)=1} r_\gamma(s, g') \quad \text{and} \quad \tilde{\Gamma}(g, g') \doteq \frac{1}{z(g)} \sum_{s:m(s,g)=1} \Gamma(s, g') \tag{3.3}$$

for normalizer $z(g) \doteq \sum_{s:m(s,g)=1} m(s, g)$. In words, the subgoal-to-subgoal models take an average of the state-to-subgoal models, where the starting state is a member of a subgoal.

Planning involves learning $\tilde{v}(g)$: the value for different subgoals. This can be achieved using an update similar to value iteration, for all $g \in \mathcal{G}$,

$$\tilde{v}(g) = \max_{g' \in \bar{\mathcal{G}}: \tilde{d}(g,g') > 0} \tilde{r}_\gamma(g, g') + \tilde{\Gamma}(g, g')\tilde{v}(g'). \tag{3.4}$$

The value of reaching g' from g is the discounted rewards along the way, $\tilde{r}_\gamma(g, g')$,

[4] The first input is any $g \in \mathcal{G}$, the second is $g' \in \bar{\mathcal{G}}$, which includes $s_\perp$. We need to reason about reaching any subgoal or $s_\perp$. We do not reason about starting from it to reach subgoals.

plus the discounted value in g'. If $\tilde{\Gamma}(g, g')$ is very small, it is difficult to reach g' from g—or takes many steps—and so the value in g' is discounted by more. With a relatively small number of subgoals, we can sweep through them all to quickly compute $\tilde{v}(g)$. With a larger set of subgoals, we can instead do as many updates possible, in the background on each step, by stochastically sampling g.

This update can be interpretted as the standard value iteration update in a new MDP, where the set of states is $\mathcal{G}$, and the actions from $g \in \mathcal{G}$ are state-dependent, corresponding to choosing which $g' \in \bar{\mathcal{G}}$ to go to in the set where $\tilde{d}(g, g') > 0$. The rewards are $\tilde{r}_\gamma$ and the discounted transition probabilities are $\tilde{\Gamma}$. [5] Under this correspondence, Lo et al. (2022) showed that the above converges to the optimal values in this new Goal-Space MDP. This idea of approximate value iteration over a more abstract MDP is not novel (Sutton et al., 1999), and the GSP framework can be regarded as an instance of the STOMP progression (Sutton et al., 2023).

Now let us examine how to use $\tilde{v}(g)$ to update our main policy. The simplest way to decide how to behave from a state is to cycle through the subgoals, and pick the one with the highest value. In other words, we can set

$$v_{g^\star}(s) \doteq \begin{cases} \max\limits_{g \in \bar{\mathcal{G}}: d(s,g) > 0} r_\gamma(s, g) + \Gamma(s, g)\tilde{v}(g) & \text{if } \exists\ g \in \bar{\mathcal{G}} : d(s, g) > 0, \text{ (projection step)} \\ \text{undefined} & \text{otherwise,} \end{cases} \tag{3.5}$$

and take action a that corresponds to the action given by π_g for this maximizing g. Note that some states may not have any subgoals nearby. $v_{g^\star}(s)$ is undefined for that state. We show the vanilla GSP algorithm in Algorithm 1. For conciseness, we refer the reader to Appendix C.1 for the full pseudocode from Lo et al. (2022).

[5] The transition probabilities are typically separate from the discount, but it is equivalent to consider the discounted transition probabilities.

Algorithm 1 `Goal Space Planning` for Episodic Problems

Assume given subgoals $\mathcal{G}$ and relevance function d
Initialize base learner (e.g. $\mathbf{w}, \mathbf{z} = \mathbf{0}, \mathbf{0}$ for Sarsa(λ)[6]), model parameters $\boldsymbol{\theta} = (\boldsymbol{\theta}^r, \boldsymbol{\theta}^\Gamma, \boldsymbol{\theta}^\pi), \tilde{\boldsymbol{\theta}} = (\tilde{\boldsymbol{\theta}}^r, \tilde{\boldsymbol{\theta}}^\Gamma)$
Sample initial state s_0 from the environment
for $t \in 0, 1, 2, ...$ **do**
    Take action a_t using q (e.g., ϵ-greedy), observe $s_{t+1}, r_{t+1}, \gamma_{t+1}$
    Choose a' from s_{t+1} using q (e.g. ϵ-greedy)
    `Update_Models`$(s_t, a_t, s_{t+1}, r_{t+1}, \gamma_{t+1})$ (see Algorithm 5)
    `Planning`() (see Algorithm 2)
    `MainPolicyUpdate`$(s_t, a_t, s_{t+1}, r_{t+1}, \gamma_{t+1}, a')$ (see Algorithm 4)

Algorithm 2 `Planning`()

for n iterations, for each $g \in \mathcal{G}$ **do**
    $\tilde{v}(g) \leftarrow \max_{g' \in \bar{\mathcal{G}}: d(g,g') > 0} \tilde{r}_\gamma(g, g'; \tilde{\boldsymbol{\theta}}^r) + \tilde{\Gamma}(g, g'; \tilde{\boldsymbol{\theta}}^\Gamma)\tilde{v}(g')$

[6] Sarsa(λ) has two sets of parameters to initialize: its action-value function weights $\mathbf{w}$, and its eligibility trace vector $\mathbf{z}$ (Rummery, 1995).

Chapter 4

Goal Space Planning with Reward Shaping

We consider three desiderata for when a model-based approach should be effective:

1. The model should be feasible to learn: we can get it to a sufficient level of accuracy that makes it beneficial to plan with that model.

2. Planning should be computationally efficient, so that the agent's values can be quickly updated.

3. Finally, the model should be modular—composed of several local models or those that model a small part of the space—so that the model can quickly adapt to small changes in the environment. These small changes might still result in large changes in the value function; planning can quickly propagate these small changes, potentially changing the value function significantly.

At a high level, the vanilla GSP algorithm focuses planning over a given set of

abstract subgoals to provide quickly updated, approximate values to speed up learning to address these desiderata. In order to do so, the agent first learns a set of subgoal-conditioned models. These models then form a temporally abstract MDP, with subgoals as states, and options to reach each subgoal as actions. Finally, the agent updates its policy by mixing these subgoal values into the TD target to speed up learning. Namely, vanilla GSP used a TD error of

$$\delta_t = R_{t+1} + \gamma_{t+1}(\beta v_{g^\star}(S_{t+1}) + (1-\beta)\hat{q}(S_{t+1}, A_{t+1}; \mathbf{w})) - \hat{q}(S_t, A_t; \mathbf{w}),$$

where $\beta \in [0, 1]$ was a hyper-parameter.[1] In this thesis, we instead use $v_{g^\star}$ as a potential, and find that it can extend the vanilla GSP to a wider range of value-based learning algorithms. PBRS with $v_{g^\star}$ was found to outperform vanilla GSP. This chapter outlines some issues with vanilla GSP outlines two theoretical properties of using GSP with Reward Shaping, relevant to the empirical results in the following chapter.

4.1 Problems with naively using $v_{g^\star}$

There are two other critical issues with this approach. Policies are restricted to go through subgoals, which might result in suboptimal policies. From a given state s, the set of relevant subgoals g may not be on the optimal path. This limitation is expressly one we early mentioned we wished to avoid, and is a key limitation of Landmark Value Iteration (LAVI) developed for the setting where models are given (Mann et al., 2015). Second, the learned models themselves may have inaccuracies, or planning may not have been completed in the background,

[1]The target of this TD error resembles the Mixed Monte Carlo update (Ostrovski et al., 2017), however instead of mixing in a multi-step return, we mix in a projection of the subgoal's value.

resulting in $\tilde{v}(g)$ that are not yet fully accurate.

4.2 $v_{g^\star}$ as a Potential

When incorporating $v_{g^\star}$ into the learning process, we want to ensure the optimal policy remains unchanged and that $v_{g^\star}$ guides the agent by helping evaluate the quality of its decisions. A simple way to satisfy these requirements is to use potential-based reward shaping (Ng et al., 1999). PBRS defines a new MDP with a modified reward function where the agent receives the reward $\tilde{R}_{t+1} = R_{t+1} + \gamma\Phi(S_{t+1}) - \Phi(S_t)$, where $\Phi\colon \mathcal{S} \to \mathbb{R}$ is any state-dependent function. Ng et al. show that such a reward transformation preserves the optimal policy from the original MDP. We propose using $\Phi = v_{g^\star}$ to modify any TD learning algorithm to be compatible with GSP. For example, in the Sarsa(λ) algorithm, the update for the weights of the action-value function $\hat{q}$ would use the TD-error

$$\delta_t \doteq \underbrace{R_{t+1} + \gamma_{t+1} v_{g^\star}(S_{t+1}) - v_{g^\star}(S_t)}_{\tilde{R}_{t+1}} + \gamma_{t+1}\hat{q}(S_{t+1}, A_{t+1}; \mathbf{w}) - \hat{q}(S_t, A_t; \mathbf{w}). \quad (4.1)$$

This modified TD error can also be incorporated into more complex value-based learning algorithms. We show how GSP with PBRS can be layered onto Double DQN (a popular value-based learning algorithm that uses non-linear value function approximation) in Algorithm 3.

PBRS rewards the agent for selecting actions that result in a state transition that increase Φ. Consider the case when Φ represents the *negative* distance to a goal state. When $\Phi(S_{t+1}) > \Phi(S_t)$, then the agent has made progress towards getting to the goal, and it receives a positive addition to the reward. When Φ is an estimate of the value function, one can interpret the additive reward as

Algorithm 3 `Goal Space Planning with Reward Shaping` on DDQN

Initialize base learner parameters $\mathbf{w}, \mathbf{w}_{\text{targ}} = \mathbf{w}_0$, $n_{\text{updates}} = 0$, target refresh rate τ,
set of subgoals $\bar{\mathcal{G}}$, relevance function d, model parameters $\boldsymbol{\theta} = (\boldsymbol{\theta}^r, \boldsymbol{\theta}^\Gamma, \boldsymbol{\theta}^\pi), \tilde{\boldsymbol{\theta}} = (\tilde{\boldsymbol{\theta}}^r, \tilde{\boldsymbol{\theta}}^\Gamma)$
Sample initial state s_0 from the environment
for $t \in 0, 1, 2, ...$ **do**
  Take action a_t using q (e.g., ϵ-greedy), observe $s_{t+1}, r_{t+1}, \gamma_{t+1}$
  Add experience $(s_t, a_t, r_{t+1}, s_{t+1}, \gamma_{t+1})$ to replay buffer D
  `Update_GSP_Models`() (see Algorithm 6)
  `Planning`() (see Algorithm 2)
  for n mini-batches **do**
    Sample batch $B = \{(s, a, r, s', \gamma)\}$ from D
    if $d(s, \cdot), d(s', \cdot) > 0$ **then**
      $v_{g^\star}(s) = \max_{g \in \bar{\mathcal{G}}: d(s,g)>0} r_\gamma(s, g; \boldsymbol{\theta}^r) + \Gamma(s, g; \boldsymbol{\theta}^\Gamma)\tilde{v}(g)$
      $v_{g^\star}(s') = \max_{g \in \bar{\mathcal{G}}: d(s',g)>0} r_\gamma(s', g; \boldsymbol{\theta}^r) + \Gamma(s', g; \boldsymbol{\theta}^\Gamma)\tilde{v}(g)$
      $\tilde{r} = r + \gamma v_{g^\star}(s') - v_{g^\star}(s)$
    else
      $\tilde{r} = r$
    $Y(s, a, r, s', \gamma) = \tilde{r} + \gamma q(s', \text{argmax}_{a'} q(s', a'; \mathbf{w}); \mathbf{w}_{\text{targ}})$
    $L = \frac{1}{|B|} \sum_{(s,a,r,s',\gamma) \in B} (Y(s, a, r, s', \gamma) - q(s, a; \mathbf{w}))^2$
    $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} L$
    if $n_{\text{updates}} \% \tau == 0$ **then**
      $\mathbf{w}_{\text{targ}} \leftarrow \mathbf{w}$
    $n_{\text{updates}} = n_{\text{updates}} + 1$

rewarding the agent for taking actions that increase the value function estimate and penalizes actions that decrease the value. In this way, using $\Phi = v_{g^\star}$, the agent can leverage immediate feedback on the quality of its actions using the information from the abstract value function about what an optimal policy might look like.

For intuition as to why potential-based reward shaping does not bias the optimal policy, notice that $\sum_{t=0}^{\infty} \gamma^t (\gamma \Phi(S_{t+1}) - \Phi(S_t)) = -\Phi(S_0)$, which means the relative values of each action remain the same.[2]

[2]It should be noted that the cancellations of these intermediate terms mean that algorithms

It is important to note that if $v_{g^\star}$ can help improve learning, it can also make learning harder if its guidance makes it less likely for an agent to sample optimal actions. This increase in difficulty is likely if the models used to construct the abstract MDP and $v_{g^\star}$ have substantial errors. In this case, the agent has to learn to overcome the bad "advice" provided by $v_{g^\star}$. We investigate this further with inaccurate models and non-stationary environments in Sections 5.4 and 5.6 respectively.

4.3 Theoretical Results

4.3.1 Shaping and Q-initialization Equivalence

In the tabular setting, it is known that using PBRS is equivalent to initialising Q to Φ and then performing updates on the same set of experience (Wiewiora, 2003). While Wiewiora explicitly showed this for tabular Q-learning and stated it extends to all TD learners, in this thesis we explicitly show it for tabular Sarsa(λ), as we use it in our experiments.

Proposition 1. *Given the same sequence of experience, performing TD(λ) updates with potential-based reward shaping is equivalent to adding the potential to the learner's initial action values and updating using the unshaped rewards, in the tabular setting.*

Proof. We first explicitly show this result for Sarsa(λ), one of the algorithms we use to empirically analyze GSP with shaping, and then show how it extends to all TD learners.

like REINFORCE (Williams, 1992) or Proximal Policy Optimization (Schulman et al., 2017) will see little benefit when combined with PBRS (as they use the discount sum of all rewards to update the policy).

We start with two Sarsa(λ) learners L and L', with Q-tables Q_t and Q'_t. L will perform Sarsa(λ) updates with PBRS, whereas L' will have its Q-table initialized as $Q'_0(s,a) = Q_0(s,a) + \Phi(s)$ and it will use unshaped rewards. $\Phi : \mathcal{S} \mapsto \mathbb{R}$ is the potential function.

Our experiences are stored as a list of 5-tuples $\mathcal{D} = \{\langle S_i, A_i, R_{i+1}, S_{i+1}, \gamma_{i+1}\rangle\}_{i=0}^{n-1}$. Both learners will use this same list of experiences.

For tabular Sarsa(λ), the update rule for an experience $\langle s, a, r, s', \gamma\rangle$ is:

$$\begin{aligned}
\mathbf{z}_t(s,a) &= 1 \quad (\text{replacing trace}) \\
Q_{t+1}(s,a) &\leftarrow Q_t(s,a) + \alpha\delta_t \mathbf{z}_t \\
\mathbf{z}_{t+1} &\leftarrow \lambda\gamma\mathbf{z}_t,
\end{aligned}$$

and we use

$$\begin{aligned}
\delta_t &= r + \gamma\Phi(s') - \Phi(s) + \gamma Q_t(s',a') - Q_t(s,a), \\
\delta'_t &= r + \gamma Q'_t(s',a') - Q'_t(s,a)
\end{aligned}$$

as the TD errors for L, and L' respectively. $\mathbf{z}_t \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$ is the eligibility trace vector [3]. We denote the change in the Q-tables after k updates from initialization as $\Delta Q_k = \sum_{t=0}^{k-1} \alpha\delta_t\mathbf{z}_t$ and $\Delta Q'_k = \sum_{t=0}^{k-1} \alpha\delta'_t\mathbf{z}_t$. Since both learners use the same list of experience, γ and λ, they would have the same eligibility trace vector $\mathbf{z}_t \,\forall\, t$. We initialise $\mathbf{z}_{-1} = \mathbf{0}$.

For the theorem to be true, we require

$$\Delta Q_t = \Delta Q'_t \;\forall\; t.$$

[3] It is a memory mechanism that, on each learning update, gives a decaying amount of credit for state-action pairs that occurred previously. The λ in Sarsa(λ) determines the rate of this decay with time Sutton and Barto (2018).

We show this using a proof by induction.

Base Case: When $t = 1$,

$$\begin{aligned}\Delta Q_1 &= \alpha\delta_0\mathbf{z}_0 \\ &= \alpha \begin{bmatrix} R_1 + \gamma\Phi(S_{t+1}) - \Phi(S_t) \\ +\gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \end{bmatrix} \mathbf{z}_0,\end{aligned}$$

$$\begin{aligned}\Delta Q'_1 &= \alpha\delta'_0\mathbf{z}_0 \\ &= \alpha[R_1 + \gamma Q'(S_{t+1}, A_{t+1}) - Q'(S_t, A_t)]\mathbf{z}_0 \\ &= \alpha \begin{bmatrix} R_1 \\ +\gamma(Q(S_{t+1}, A_{t+1}) + \Phi(S_{t+1})) \\ -(Q(S_t, A_t) + \Phi(S_t)) \end{bmatrix} \mathbf{z}_0 \\ &= \alpha \begin{bmatrix} R_1 + \gamma\Phi(S_{t+1}) - \Phi(S_t) \\ +\gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \end{bmatrix} \mathbf{z}_0 \\ &= \Delta Q_1.\end{aligned}$$

The changes to the Q tables are equivalent after 1 update.

Assumption : $\exists\ k\ \in \mathbb{N}$ s.t. $\Delta Q_k = \Delta Q'_k$.

Inductive Step: When $t = k + 1$, the learner L updates with experience

$\langle s, a, r, s', \gamma \rangle$.

$$\begin{aligned}
\Delta Q_{k+1} &= \Delta Q_k + \alpha \delta_k \mathbf{z}_k \\
&= \Delta Q_k + \alpha [r + \gamma \Phi(s') - \Phi(s) + \gamma Q_k(s', a') - Q_k(s, a)] \mathbf{z}_k \\
&= \Delta Q_k + \alpha \begin{bmatrix} r + \gamma \Phi(s') - \Phi(s) \\ +\gamma(Q_0(s', a') + \Delta Q_k(s', a')) \\ -Q_0(s, a) - \Delta Q_k(s, a) \end{bmatrix} \mathbf{z}_k. \qquad (3.2)
\end{aligned}$$

The third line was possible because $Q_k = Q_0 + \sum_{t=0}^{k-1} \alpha \delta_t \mathbf{z}_t$ (i.e. Q_k is the initalization plus $k-1$ updates). Whereas learner L' updates as:

$$\begin{aligned}
\Delta Q'_{k+1} &= \Delta Q'_k + \alpha \delta'_k \mathbf{z}'_k \\
&= \Delta Q'_k + \alpha [r + \gamma Q'_k(s', a') - Q'_k(s, a)] \mathbf{z}'_k \\
&= \Delta Q'_k + \alpha \begin{bmatrix} r \\ +\gamma(Q_0(s', a') + \Phi(s') + \Delta Q'_k(s', a')) \\ -Q_0(s, a) - \Phi(s) - \Delta Q'_k(s, a) \end{bmatrix} \mathbf{z}'_k \\
&= \Delta Q'_k + \alpha \begin{bmatrix} r + \gamma \Phi(s') - \Phi(s) \\ +\gamma(Q_0(s', a') + \Delta Q'_k(s', a')) \\ -Q_0(s, a) - \Delta Q'_k(s, a) \end{bmatrix} \mathbf{z}'_k. \qquad (3.3)
\end{aligned}$$

By our assumption, $\Delta Q_k = \Delta Q'_k$. Furthermore as $\mathbf{z}_k = \mathbf{z}'_k$, we see that (3.2) = (3.3).

$$\implies \Delta Q_{k+1} = \Delta Q'_{k+1}.$$

So if $\Delta Q_k = \Delta Q'_k$, we have shown that $\Delta Q_{k+1} = \Delta Q'_{k+1}$. Since we have shown that $\Delta Q_0 = \Delta Q'_0$, by induction $\Delta Q_t = \Delta Q'_t \ \forall \ t \in \mathbb{N}$.

More generally, this holds for any TD-learner. This can be seen if we consider the TD errors,

$$\delta_t = r + \gamma\Phi(s') - \Phi(s) + \gamma\mathcal{C}Q_t(s', \cdot) - Q_t(s, a) \text{ and}$$
$$\delta'_t = r + \gamma\mathcal{C}Q_t(s', \cdot) - Q'_t(s, a)$$

Where $\mathcal{C}Q_t(s', \cdot) = \sum_{a' \in \mathcal{A}} \alpha_{a'} Q_t(s', a')$ denotes a convex combination over actions. This includes learners like Expected Sarsa, $\alpha_{a'} = \Pr(A_{t+1} = a')$, (John, 1994) and Q-learning, $\alpha_{a'} = \begin{cases} 1 & a' \in \operatorname{argmax}_{a \in \mathcal{A}} Q(s', a), \\ 0 & \text{otherwise.} \end{cases}$ □

4.3.2 Shaping with $v^\star$

Using $v_{g^\star}$ as a potential can help quickly identify $\pi^\star$. Specifically, when $v_{g^\star}$ is $v^\star$, and the value function is constant, e.g., initialized to $\mathbf{0}$, it only takes one application of the Bellman operator in each state to find the optimal policy. We formalize this in the proposition below.

Proposition 2. *For $v_{g^\star} = v^\star$ and $v_0 = c$, for $c \in \mathbb{R}$, then the policy, π_1 derived after a single Bellman update at all states will be optimal, i.e.,*

$$\pi_1(s) \in \operatorname*{argmax}_a q^\star(s, a) \;\forall\; s \in \mathcal{S}.$$

Proof. Let the q estimate for the k^{th} iteration be

$$q_k(s, a) = R(s, a) + \sum_{s'} P(s, a, s') \left(\gamma v_{g^\star}(s') - v_{g^\star}(s) + \gamma v_{k-1}(s')\right).$$

The value function for iteration k is $v_k = \max_a q_k(s, a)$ and the policy for the k^{th}

iteration is $\pi_k(s) \in \operatorname{argmax}_a q_k(s,a)$. The value of q_1 is

$$\begin{aligned}
q_1(s,a) &= R(s,a) + \sum_{s'} P(s,a,s') \left(\gamma_c v_{g^\star}(s') - v_{g^\star}(s) + \gamma v_0(s')\right) \\
&= R(s,a) + \sum_{s'} P(s,a,s') \left(\gamma v^\star(s') - v^\star(s) + \gamma v_0(s')\right) \\
&= \underbrace{R(s,a) + \sum_{s'} P(s,a,s') \gamma v^\star(s')}_{=q^\star(s,a)} + \underbrace{\sum_{s'} P(s,a,s') \gamma v_0(s')}_{=\gamma c} - v^\star(s) \\
&= q^\star(s,a) + \gamma c - v^\star(s)
\end{aligned}$$

where the last line follows because $v_0(s') = c$ for all s'. Then plugging this expression into π_1 yields

$$\pi_1(\cdot|s) \in \operatorname*{argmax}_a q^\star(s,a) + \gamma c - v^\star(s) = \operatorname*{argmax}_a q^\star(s,a).$$

□

While having $v_{g^\star} = v^\star$ is not realistic, Proposition 2 means that the policy will quickly align with what is preferable under $v_{g^\star}$ before finding what is optimal for the MDP without the shaping reward.

Chapter 5

Experiments

This chapter empirically studies our GSP with Reward Shaping algorithm in different settings. Since GSP plays a key role in helping the agent learn its value function, we first analyse the effects of GSP on a tabular value function before moving on to common linear and non-linear value function approximation techniques. We then look at how much the effect of GSP depends on its three core components: the models (in Section 5.4), the Goal-Space (in Section 5.5), and lastly the potential (in Section 5.7).

We do so using value-based learners in three domains: FourRooms, PinBall (Konidaris and Barto, 2009) and GridBall (a version of PinBall without velocities) [1]. Unless otherwise stated, all learning curves are averaged over 30 runs, with shaded regions representing one standard error.

[1]The PinBall configuration that we used is based on the easy configuration found at `https://github.com/DecisionMakingAI/BenchmarkEnvironments.jl`, which was released under the MIT license. We have modified the environment to support additional features such as GridBall, changing terminations, visualizing subgoals, and various bug fixes.

5.1 Tabular Results

A central hypothesis of this work is that GSP can accelerate value propagation. By using information from local models in our updates, our belief is that GSP will have a larger change in value to more states, leading to policy changes over larger regions of the state space.

Hypothesis 1. *GSP changes the value for more states with the same set of experience, compared to a model-free learner.*

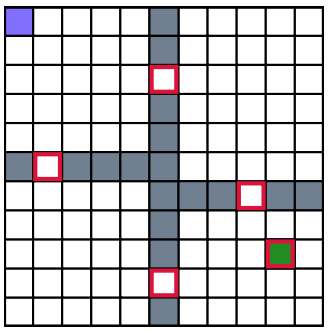


Figure 5.1: The FourRooms domain. The blue square is the initial state, green square the goal state, and red boxes the subgoals. A subgoal's initiation set contains the states in any room connected to that subgoal.

In order to verify whether GSP helps to quickly propagate value, we first test this hypothesis in a simple grid world environment: the FourRooms domain. The agent can choose from one of 4 actions in a discrete action space $\mathcal{A} = \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$. All state transitions are deterministic. The grey squares in Figure 5.1 indicate walls, and the state remains unchanged if the agent takes an action that leads into a wall. This is an episodic task, where the base learner has a fixed start state and must navigate to a fixed goal state where the episode terminates. Episodes can also terminate by timeout after 1000 timesteps.

In this domain, we test the effect of using GSP with pre-trained models on a Sarsa(λ) base learner in the tabular setting (i.e. no function approximation for the value function). Full details on using GSP with this temporal difference (TD) learner can be found in Algorithm 4. We set the four hallway states plus the goal state as subgoals, with their initiation sets being the two rooms they connect. Full details of option policy learning can be found

in the appendix E.

Figure 5.2 shows the base learner's action-value function after a single episode using four different algorithms: Sarsa(0), Sarsa(λ), Sarsa(0) + GSP, and Sarsa(λ) + GSP. In Figure 5.2, the Sarsa(0) learner updates the value of the state-action pair that immediately preceded the +1 reward at the goal state. The plot for Sarsa(λ) shows a decaying trail of updates made at the end of the episode, to assign credit to the state-action pairs that led to the +1 reward. The plots for the GSP variants show that all state-action pairs sampled receive instant feedback on the quality of their actions. The updates with GSP can be both positive or negative based on if the agent makes progress towards the goal state or not. This direction of update comes from the potential-based reward shaping rewards/penalizes transitions based on whether $\gamma_{t+1} v_{g^\star}(S_{t+1}) > v_{g^\star}(S_t)$. It is clear that projecting subgoal values from the abstract MDP leads to action-value updates over more of the visited states, even without credit assignment mechanisms such as eligibility traces.

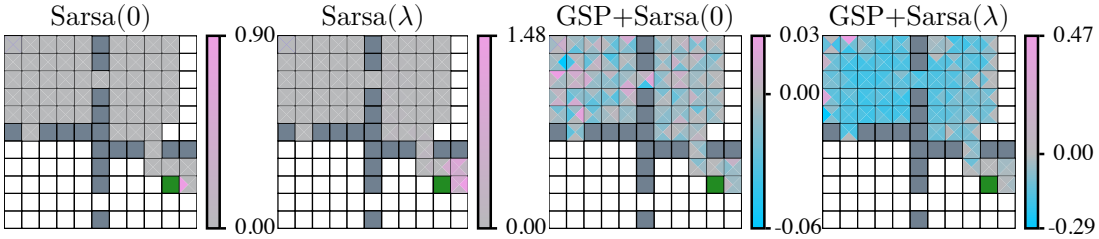


Figure 5.2: These four plots show the action values after a single episode of updates for Sarsa with and without GSP and eligibility traces, i.e., $\lambda = 0.9$. Each algorithm's update is simulated from the same data collected from a uniform random policy. Each state (square) is made up of four triangles representing each of the four available actions. White squares represent states not visited in the episode.

It is evident from these updates over a single episode that the resulting policy from GSP updates should be more likely to go to the goal. We would like to

quantify how much faster this propagated value can help our base learner over multiple episodes of experience. More specifically, we want to test the following hypothesis.

Hypothesis 2. *GSP enables a TD base-learner to learn faster.*

We expect GSP to improve a base learner's performance on a task within fewer environment interactions. We shall test whether the value propagation over the state-action space as seen in Figure 5.2 makes this the case over the course of several episodes (i.e. we are now testing the effect of value propagation over time). Figure 5.3 shows the performance of a Sarsa(λ) base learner with and without GSP in the FourRooms domain with a reward of -1 per step.

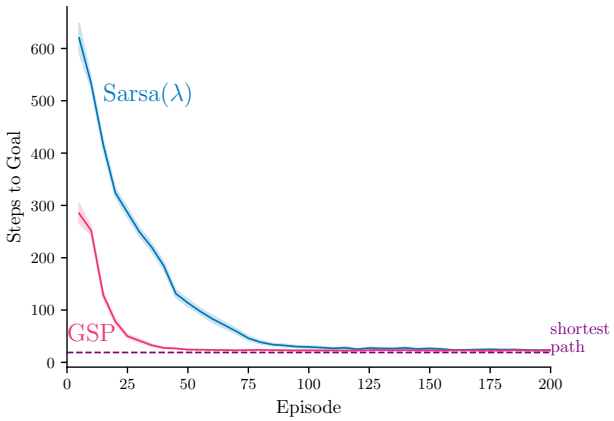


Figure 5.3: This plot shows the average number of steps to goal smoothed over five episodes in the FourRooms domain. Shaded region represents 1 standard error across 100 runs.

5.2 Linear Function Approximation Results

Many real-world applications of RL involve large and/or continuous state spaces. Besides making it unfeasible to maintain a look-up table of the value of each

state, current planning techniques struggle with such state spaces. This motivates an investigation into how well Hypotheses 1 and 2 hold when GSP is used in such environments (e.g. the PinBall domain). To better analyze GSP and its value propagation across state space, we also created an intermediate environment between FourRooms and PinBall called GridBall.

PinBall is a continuous state domain where the agent navigates a ball through a set of obstacles to reach the main goal. This domain uses a four-dimensional state representation of positions and velocities, $(x, y, \dot{x}, \dot{y}) \in [0, 1] \times [0, 1] \times [-2, 2] \times [-2, 2]$. The agent chooses from one of five actions at each timestep. $\mathcal{A} = \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}, \texttt{no\_op}\}$, where the `no_op` action adds no change to the ball's velocity, and the other actions each add an impulse force in one of the four cardinal directions. In all our experiments, the agent is initialized with zero velocity at a fixed start position at the beginning of every episode.

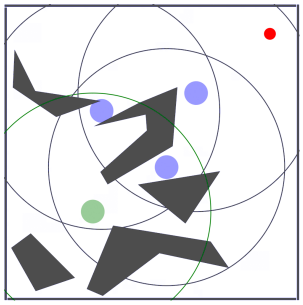


Figure 5.4: Obstacles and subgoals for GridBall and PinBall. The larger circles show the initiation set boundaries. Subgoals are defined in position space.

All collisions are elastic and we use a drag coefficient of 0.995. This is an episodic task with a fixed starting state and main goal. An episode ends when the agent reaches the main goal or after 1,000 time steps. It should be noted that, unlike in the FourRooms environment, there exist states which are not in the initiation set of any subgoal - a common occurrence when deploying GSP in the state spaces of real-world applications.

GridBall is like PinBall, but changed to be more like a grid world to facilitate visualization. The velocity components of the state are removed, meaning the state only consists of (x, y) locations, and the action space is changed to displace the ball by a fixed amount in each cardinal dimension. We keep the same obstacle

collision mechanics and calculations from PinBall. Since GridBall does not have any velocity components, we can plot heatmaps of value propagation without having to consider the velocity at which the agent arrived at a given position.

For Hypothesis 1, we repeat the experiments on GridBall with base learners that use tile-coded features (Sutton and Barto, 2018), and linear value function approximation. We use the same subgoal configuration as Figure 5.4. Full details on the option policies and subgoal models used for this are outlined in shown in Appendices A and B. Like in the FourRooms experiment, we set the reward to be 0 at all states and +1 once the agent reaches any state in the main goal, in order to show value propagation. We collect a single episode of experience from the Sarsa(0)+GSP learner and use its trajectory to perform a batch update on all learners. This controls for any variability in trajectories between learners, so we can isolate and study the change in value propagation.

Figure 5.5 compares the state value function (averaged over the action value estimates) of Sarsa(0), Sarsa(λ), Sarsa(0)+GSP, and Sarsa(λ)+GSP learners after a single episode of interaction with the environment. The results are similar to those on FourRooms. The Sarsa(0) algorithm only updates the value of the tiles activated by the state preceding the goal. Sarsa(λ) has a decaying trail of updates to the tiles activated preceding the goal, and the GSP learners update values at all states in the initiation set of a subgoal.

To examine how GSP translates to faster learning (Hypothesis 2), we measure the performance (steps to goal) over time for each algorithm in both GridBall and PinBall domains. Figure 5.6 shows that GSP significantly improves the rate of learning in these larger domains too, with the base learner able to reach near its top performance within 75 and 100 episodes in GridBall and PinBall respectively. All runs can find a similar length path to the goal. As the size of the state space

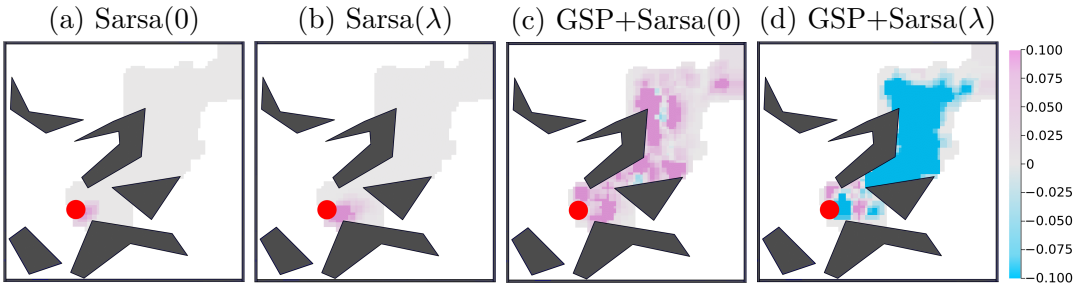


Figure 5.5: The tile-coded value function after one episode in GridBall. Like Figure 5.2, the gray regions show the visited states that were not updated. The red circle is the main goal.

increases, the benefit of using local models in the GSP updates still holds.

Similar to the previous domains, the Sarsa(λ) learner using GSP is able to reach a good policy much faster than the base learner without GSP. In both domains, the GSP and non-GSP Sarsa(λ) learners plateau at the same average steps to the goal. Even though the obstacles remain unchanged from GridBall, it takes roughly 50 episodes longer for even the GSP variant to reach a good policy in PinBall. This is likely due to the continuous 4-dimensional state space making the task harder.

5.3 Deep Reinforcement Learning Results

The previous results shed light on the dynamics of value propagation with GSP when a learner is given a representation of its environment (a look-up table or a tile coding). A natural next step is to look at whether the reward and transition dynamics learned with GSP can still propagate value (Hypothesis 2) in the deep RL setting, where the learner must also learn a representation of its environment.

We test this by running a DDQN base learner (van Hasselt et al., 2016) in the PinBall domain, with GSP layered on DDQN as in Algorithm 3. The base

(a) GridBall (b) PinBall

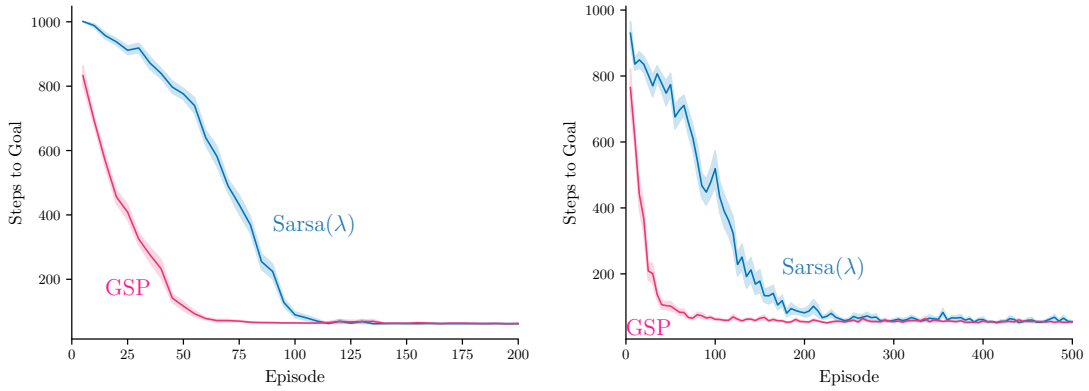


Figure 5.6: Five episode moving average of return in the GridBall over 200 episodes (left) and PinBall over 500 episodes (right). We performed 30 runs, and showed 1 standard error in the shaded region. All learners used linear value function approximation on their tile-coded features.

learner's complete hyper-parameter specifications are in Appendix F.

Unlike the previous experiments, using GSP out of the box resulted in the base learner converging to a sub-optimal policy. This is despite the fact that we used the same $v_{g^\star}$ as the previous PinBall experiments. We investigated the distribution of shaping terms added to the environment reward and observed that they were occasionally an order of magnitude greater than the environment reward. Though the linear and tabular methods handled these spikes in potential difference gracefully, these large displacements seemed to cause issues when using neural networks and a DDQN base learner.

We tested two variants of GSP that better control the magnitudes of the raw potential differences $(\gamma\Phi(S_{t+1}) - \Phi(S_t))$. We adjusted for this by either clipping or down-scaling the potential difference added to the reward. The scaled reward multiplies the potential difference by 0.1. Clipped GSP clips the potential difference into the $[-1, 1]$ interval. It should be noted that clipping the poten-

(a) GSP modified for Deep RL (b) Robustness to model accuracy

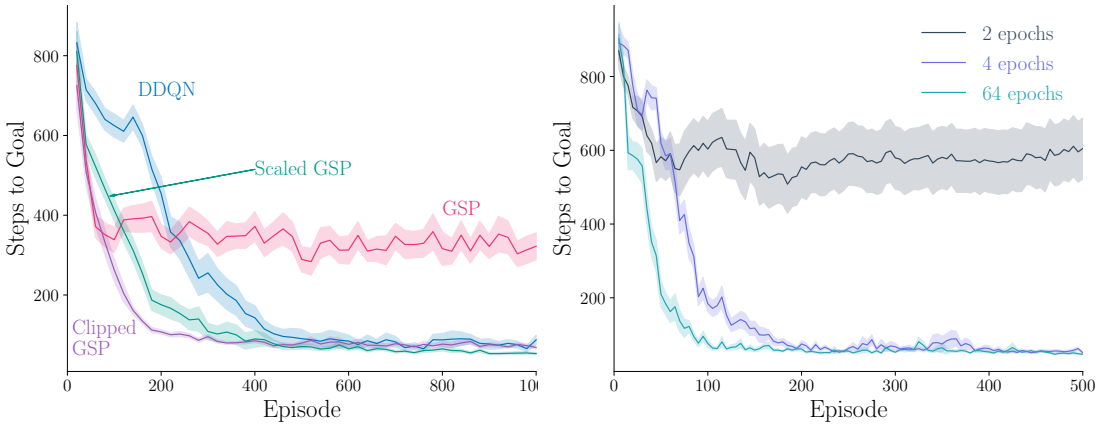


Figure 5.7: Investigating the behavior of GSP in the deep reinforcement learning setting in PinBall. (a) Following the format of Figure 5.6, we show the 20 episode moving average of steps to the main goal in PinBall. (b) Five episode moving average of steps to goal in PinBall for GSP with models trained with differing numbers of epochs.

tial difference no longer guarantees the optimal policy will be preserved. With these basic magnitude controls, GSP again learns significantly faster than its base learner, as shown in Figure 5.7a.

5.4 Robustness to Accuracy of the Learned Models

In this section, we investigate how robust GSP is to inaccuracy of its models. When examining the accuracy of the learned models, we found the errors in r_γ and Γ could be as high as 20% in some parts of the state space (see Appendix E for more information). Despite this level of inaccuracy in some states, GSP still learned effectively, as seen in Sections 5.1, 5.2 and 5.3.

We conducted a targeted experiment controlling the level of accuracy to better understand this robustness and test the following hypothesis.

Hypothesis 3. *GSP can learn faster with more accurate models, but can still improve on the base learner even with partially learned models.*

We varied the number of epochs to obtain models of varying accuracy. Our models were fully connected artificial neural networks, and we learn the models for each subgoal by performing mini-batch stochastic gradient descent on a dataset of trajectories that end in a member state of that subgoal g. Full implementation details for this mini-batch stochastic gradient descent can be found in Appendix B.

As expected, Figure 5.7b shows that more epochs over the same dataset of transitions improves how quickly the base learner reaches the main goal. Within 4 epochs of model training, the learner is able to reach a good policy to the main goal. However, if the model is very inaccurate (2 epochs), the GSP update will bias the base learner to a sub-optimal policy. There is a trend of diminishing improvement when iterating over the same dataset of experience: doubling the number of epochs from two to four results in a policy that reaches the main goal 10× quicker, but a learner which used a further 16× the number of epochs attains a statistically identical episode length by episode 500. While more accurate models lead to faster learning, relatively few epochs are required to propagate enough value to help the learner reach a good policy.

5.5 The Role of Subgoal Selection

While the above experiments show that goal-space planning can speed up learning and propagate value faster, it is crucial to understand how value propagation

depends on the selection of subgoals. Specifically, we want to identify 1) how the graphical structure of the subgoals impacts value propagation in $v_{g^\star}$ and 2) how quickly the base learner can change their policy.

To answer these questions we consider a setting where the agent is presented with new information that indicates it should change its behavior. We will then update the state-to-subgoal and subgoal-to-subgoal models online and measure how much $v_{g^\star}$ changes, along with how quickly the base learner can change its policy on different subgoal configurations.

For this task, the agent has to decide between taking one of two paths to a goal state. We initialize the agent to use an optimal policy so it takes the shorter of the two paths. Then we introduce a lava pool along the optimal path that gives the agent a large negative reward for entering it. This negative reward means the initial path is no longer optimal and that the agent needs to switch to the alternate, reward-respecting path.

The FourRooms environment uses -1 reward per step, and each state in the lava pool has a reward of -20. The agent, initialized with $q^\star$ for the original FourRooms environment, is run for 100 episode in the new FourRooms environment with the lava pool. We run GSP with Sarsa in this tabular setting for all subgoal configurations for 200 runs each.

We test the following hypothesis.

Hypothesis 4. *The placement of subgoals along the initial and alternate optimal paths are essential for fast adaptation.*

To test this hypothesis, we will evaluate the following four subgoal configurations. The first subgoal arrangement contains no subgoals near the goal state and the goal state is not connected the other subgoals. The second contains a

subgoal on the initial optimal path, but no subgoal on the alternate path. The third is where there is subgoal on the alternate path but no subgoal on the initial optimal path. The last is where there are subgoals on both paths. We illustrate these subgoal configurations in Figure 5.8. Recall that a subgoal's initiation set is the states in the two adjacent rooms.

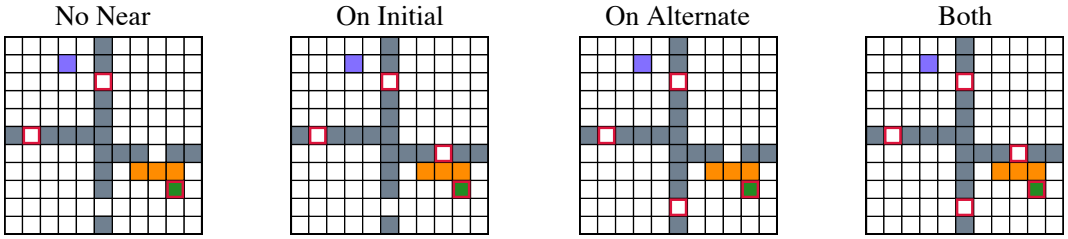


Figure 5.8: Different subgoal configurations in the FourRooms environment with a lava pool. The purple square is the learner's starting location, the gray squares the walls, the orange squares the location of the lava pool, and the green square the goal location. The only difference between these figures are the red boxes, which indicate the states that are subgoals for that configuration.

For this experiment, the state-to-subgoal models and abstract MDP need to be updated online. However, since only the reward function is changing, we only need to update the reward models r_γ and $\tilde{r}_\gamma$. Furthermore, we can represent r_γ using successor features so that the agent only needs to estimate the reward function (Barreto et al., 2017). Let $\boldsymbol{\psi}^{\pi_g}(s) \approx \mathbb{E}_{\pi_g}\left[\sum_{k=0}^{\infty}\prod_{k'=0}^{k}\gamma_{t+k'}\boldsymbol{\phi}(S_{t+k})|S_t = s\right]$, where $\boldsymbol{\phi}(S_t) \in \mathbb{R}^n$ is the feature vector at state S_t and actions are selected according to option policy π_g. Then $r_\gamma(s,g) = \mathbf{w}^\top\boldsymbol{\psi}^{\pi_g}(s)$, where $\mathbf{w} \in \mathbb{R}^n$. The learner can then update r_γ by estimating the reward function with stochastic gradient descent, i.e., $\mathbf{w} \leftarrow \mathbf{w} + \eta[R_t - \mathbf{w}^\top\boldsymbol{\phi}(S_t)]\boldsymbol{\phi}(S_t)$ for some scalar step size η.

To understand how learning is impacted by the subgoal configuration we show the return and probability the agent takes the alternative path in Figure 5.9. The first thing that is apparent is that all configurations are able to change the policy so that the probability of taking the alternative path increases. The main

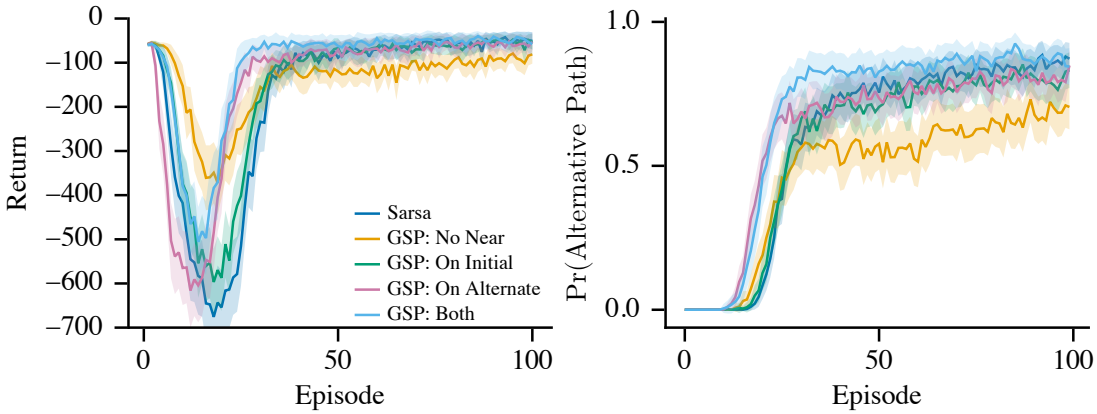


Figure 5.9: This figure shows the average return (left) and average probability the agent will take the alternative path (right) from each episode. Shaded regions represent (0.05,0.9)-tolerance intervals (Patterson et al., 2020) over 200 trials.

differences come from how quickly, in expectation, each configuration is able to change the policy to have a high probability of taking the alternate path. The Both and On Alternate subgoal configurations have the quickest change in the policy on average, while the other methods are slower. The No Near configuration also seems to, on average, have the smallest increase in probability of taking the alternate path. These results suggest that for GSP to be most impactful, there needs to be a path through the subgoals that represents the desirable path.

To better understand these results, we look more closely at $v_{g^\star}$ for each configuration. We measure how $v_{g^\star}$ changes over learning, i.e. $v_{g^\star,t} - v_{g^\star,0}$, where $v_{g^\star,i}$ is the value of $v_{g^\star}$ after episode i. We first examine the values of $v_{g^\star}$ for each subgoal configuration before the introduction of the lava pool (top row in Figure 5.10). For the No Near subgoals configuration, $v_{g^\star}$ has a disconnected graph, so all but the room with the goal state has a large negative value.

For both On Initial and On Alternate configurations, $v_{g^\star}$ is the smallest in the room that is furthest from the goal state according to the abstract MDP. This is due to the structure of the abstract MDP only knowing about a single path to

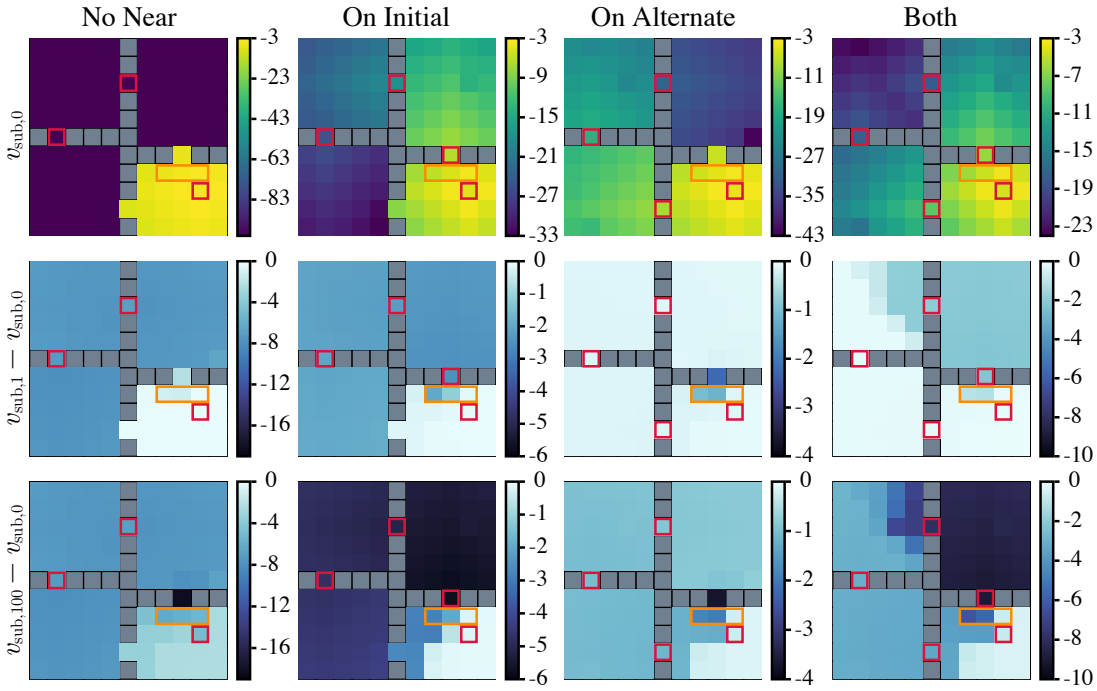


Figure 5.10: The top row of this figure shows the value of $v_{g^\star}$ for each state before the lava pool, for each subgoal configuration. The second and third rows show the change in $v_{g^\star}$ after the first and 100^{th} episode, after the lava pool is introduced.

the goal state.

In the Both subgoal configuration $v_{g^\star}$ closely represents the optimal value function in each state.

We then look at the change in $v_{g^\star}$ after the lava pool is introduced, after one episode (middle row in Figure 5.10) and 100 episodes (bottom row in Figure 5.10). We notice that the change in $v_{g^\star}$ follows the same patterns as the value representation. The value in the No Near subgoal configuration does not propagate information from the lava pool to rooms outside the bottom left room. For the On Initial configuration, the value decreases quickly in the top right room, but also the other two rooms as well. After 100 episodes the value is decreased in most states but the top right room sees the largest decrease. For the On Al-

ternate configuration value is not quickly propagated after discovering the lava pool because there is no connected region from the path the agent took to the lava pool. However, small changes are propagated over time due to the small probability of hitting the lava pool on the alternate path. With the Both subgoal configuration, value is quickly decreased in the states that would take the initial path, but not the alternate path. This indicates the desirable path through subgoals changes in the abstract MDP. Over time the decrease in value is largely isolated to the top right room with the decreases in the other rooms coming from small chances of hitting the lava pool on the alternate path.

Remark: We also examined the utility of these subgoals for learning before the lava pool was introduced. Here we found that the On Alternate subgoal placement actually caused the agent to learn a suboptimal policy, because it biased it towards the alternate path initially. You can see a visualization of this $v_{g^\star}$ in Figure 5.10 (top row, third column). The base learner does not use a smart exploration strategy to overcome this initial bias, and so settles on a suboptimal solution—namely, to take the slightly longer alternate path. See Appendix 5.6 for the full details and results for this experiment. Note that this suboptimality did not arise in the above experiment, because the lava pool made one path significantly worse than the other, pushing the agent.

5.6 Subgoal Placement and the Region of Attraction

A counter intuitive observation from the experiments in Section 5.5 was that the On Alternate path helped the agent quickly change its policy but $v_{g^\star}$ did not quickly change. In this section, we investigate this reason and put forth the

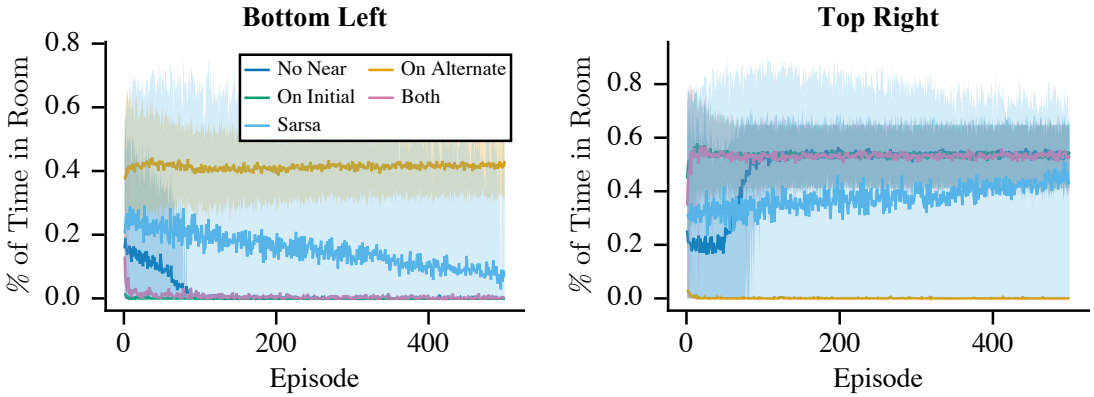


Figure 5.11: This figure shows the time the agent spends per episode in the bottom left and top right rooms. The lines convey the average % of time the agent spend and the shaded lines represent $(0.05, 0.9)$ tolerance intervals computed from 100 trials.

following hypothesis:

Hypothesis 5. *GSP creates a region of attraction so that the agent follows the optimal path as determined by the abstract MDP.*

That is to say, if a single chain of subgoals is represented in the abstract MDP, then the learner will initially try and closely follow this path even if it is not the optimal path. To test this hypothesis, we want to see that the agent will occupy states similar to what is specified by the optimal path in the abstract MDP. For this experiment, we ran GSP on FourRooms (without the lava pools) with each subgoal configuration defined in the previous section. We measured how much time the agent spends in the bottom left room and the top right room. The agent should, as it learns about the environment, spend more time in the top right room and less time in the bottom left room. We would expect all agents to follow this trend, except for the one that is missing a subgoal to go through the top right room. We show the results for each configuration and Sarsa(0) with no GSP in Figure 5.11. The results in Figure 5.11 are clear. All methods learn to go through

the top right room except for the subgoal configuration missing a subgoal on that path to the goal state. This supports our hypothesis that the agent will learn to follow the optimal path as specified by the abstract MDP. This also means that while potential-based shaping (used to propagate value information from the abstract MDP to the base learner) does not change the optimal policy, it can make it harder for the learner to find the optimal policy.

Based on the experiments in Section 5.5 and this one, we can conclude a few key points about GSP. The first is that $v_{g^\star}$ can only provide value information as determined by the optimal value function through the abstract MDP, which may not reflect the connectivity of the original MDP. Second, the learner's exploration through the state space will be highly impacted by the known subgoals. With the basic ϵ-greedy exploration policy that GSP currently uses, GSP will quickly follow and refine the best policy found within the abstract MDP. If the optimal policy is near to the policy found by the abstract MDP, then GSP will be able to quickly discover it. However, if the optimal policy is very different than the one found by the abstract MDP (for example, if the best abstract MDP policy follows an alternate sub-optimal path), this will make the agent explore around its sub-optimal policy, and thus possibly slowing down the discovery of the optimal policy, because the basic ϵ-greedy exploration policy centralizes exploration around the current best policy known by the agent.

This is all to say that there is work to be done to improve GSP's exploration by incorporating more sophisticated exploration strategies. There are also opportunities to develop new exploration strategies that takes advantage of how GSP learns with the knowledge of subgoals within the environment. For example, one may consider leveraging an existing subgoal formulation for more directed exploration by introducing reward bonuses at other subgoals, once we know the environment has changed. Additional work to find new subgoals or refine the

current subgoal configurations can also have a high impact in how well GSP can explore and adapt to changes in the environment.

5.7 Comparison with other Potentials

Having shown several instances of $v_{g^\star}$ being used as a potential for reward shaping, we shall now investigate how much of the GSP performance improvements are due to $v_{g^\star}$ capturing useful information about the MDP, rather than just being a general consequence of using a good heuristic with potential-based reward shaping.

Hypothesis 6. *Using any potential function that captures the relative importance of a transition will increase the learning efficiency of the base learner, but $v_{g^\star}$ that is tailored to the MDP will allow for faster learning.*

We test this by comparing $v_{g^\star}$ with two other potentials - an informative and an uninformative one - in the PinBall domain. The first potential function is the negative L_2 distance in position space (scaled) to the main goal, $(x_g, y_g)^\top$,

$$\Phi(S_t) = -100 \left\| \begin{pmatrix} x_g \\ y_g \end{pmatrix} - \begin{pmatrix} x(S_t) \\ y(S_t) \end{pmatrix} \right\|_2, \tag{5.1}$$

where $x(S_t)$ and $y(S_t)$ are functions that return the x and y coordinates of the agent's state respectively. This potential function captures a measure of closeness to the goal state, but does not consider obstacles or the velocity component. So it should provide some learning benefit but should not be as helpful as $v_{g^\star}$. We scale this potential by a factor of 100 to make it comparable in magnitude to $v_{g^\star}$. Reward shaping with the unscaled L_2 distance did not have any significant effect on the base learner.

The second potential is created by randomly assigning a value for each state,

$$\forall s \in \mathcal{S},\ \Phi(s) \leftarrow \mathcal{U}[-100, 0]. \tag{5.2}$$

This potential does not encode any useful information about the environment on average. It would make learning harder as it encourages the agent to take sub-optimal actions.

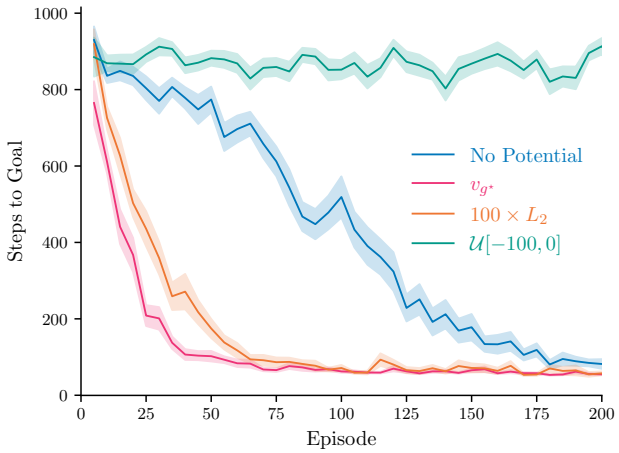


Figure 5.12: Five episode moving average of steps to goal in PinBall with different potential functions for $\Phi(s)$. We follow the format of Figure 5.7a.

We compare the performance of a Sarsa(λ) base learner using each of the three potentials. We use the PinBall domain with the same subgoal configuration and settings as in Section 5.2 and display the results in Figure 5.12. Using $v_{g^\star}$ for the potential reaches the main goal fastest, though using L_2 also resulted in significant speed-ups over the base learner (No Potential). The L_2 heuristic, however, is specific to navigation environments, and finding general purpose heuristics is difficult. Using a subgoal formulation for the potential is more easily extensible to other environments. The random potential harms performance, likely because it skews the reward and impacts exploration negatively.

Chapter 6

Conclusions and Future Works

Goal-Space Planning provides a new approach for using background planning to improve value propagation with local models and computationally efficient planning. In this thesis, we showed and studied a new way to use the GSP framework. We showed that with potential-based reward shaping, the information from subgoal-conditioned models can be used to quickly propagate value through state spaces of varying sizes. We find a consequent learning speed-up on base learners with different types of value function approximation. Subgoal selection was found to play a big role on the value function and policy the base learner reaches. In particular, we see that GSP with Reward Shaping helps the base learner find a path through the state space, based on the high-level path found in the abstract MDP. We also verify that the performance improvement observed in GSP is the result of $v_{g^\star}$ capturing the MDP dynamics, and not a general consequence of potential-based reward shaping.

This work introduces a new formalism and many new technical questions along with it. Our experiments learning the models online using successor representa-

tions indicate that GSP can get similar learning speed boosts. Using a recency buffer, however, accumulates transitions only along the optimal trajectory, sometimes causing the models to become inaccurate part-way through learning. An important next step is to incorporate smarter model learning strategies. The other critical open question is subgoal discovery. For this work, we relied on hand-picked subgoals, but an intelligent agent should discover its subgoals. One utility of this work is that it could help narrow the scope of the discovery question, to that of finding abstract subgoals that help a learner plan more efficiently. Additionally, algorithms that use discounted sums of rewards from a trajectory to update the policy (like REINFORCE (Williams, 1992) or the widely used Proximal Policy Optimization (Schulman et al., 2017)) will see little benefit when combined with our reward shaping techniques. In this setting we would need to estimate a $q_{g^\star}$ to leverage trajectory-wise control variates. Lastly, it would be interesting if the agent can control the level of abstraction it plans at. Namely learning a value function in multiple abstract MDPs, and giving the agent the option to choose what level it plans at before making decisions.

References

D. Abel. *A Theory of Abstraction in Reinforcement Learning*. PhD thesis, Brown University, 2022.

D. Arumugam and B. Van Roy. Deciding what to model: Value-equivalent sampling for reinforcement learning. *Advances in Neural Information Processing Systems*, 35:9024–9044, 2022.

A. Ayoub, Z. Jia, C. Szepesvári, M. Wang, and L. Yang. Model-Based Reinforcement Learning with Value-Targeted Regression. In *International Conference on Machine Learning*, 2020.

A. Bagaria, J. K. Senthil, and G. Konidaris. Skill discovery for exploration and planning using deep skill graphs. In *International Conference on Machine Learning*, 2021.

B. Bakker, Z. Zivkovic, and B. Krose. Hierarchical dynamic programming for robot path planning. In *International Conference on Intelligent Robots and Systems*, 2005.

E. Barnard. Temporal-difference methods and markov models. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(2):357–365, 1993.

A. Barreto, W. Dabney, R. Munos, J. J. Hunt, T. Schaul, D. Silver, and H. van

Hasselt. Successor features for transfer in reinforcement learning. In *Advances in Neural Information Processing Systems*, 2017.

P. Behboudian, Y. Satsangi, M. E. Taylor, A. Harutyunyan, and M. Bowling. Policy invariant explicit shaping: an efficient alternative to reward shaping. *Neural Computing and Applications*, pages 1–14, 2022.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

D. P. Bertsekas. *Dynamic programming: deterministic and stochastic models*. Prentice-Hall, Inc., 1987.

R. Chitnis, T. Silver, J. B. Tenenbaum, T. Lozano-Pérez, and L. P. Kaelbling. Learning Neuro-Symbolic Relational Transition Models for Bilevel Planning. In *International Conference on Intelligent Robots and Systems*. IEEE, 2022.

T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

C. Diuk, A. L. Strehl, and M. L. Littman. A hierarchical approach to efficient reinforcement learning in deterministic domains. In *International Joint Conference on Autonomous Agents and Multiagent Systems*, 2006.

A.-m. Farahmand. Iterative Value-Aware Model Learning. In *Advances in Neural Information Processing Systems*, 2018.

A.-m. Farahmand, A. M. S. Barreto, and D. N. Nikovski. Value-Aware Loss Function for Model-based Reinforcement Learning. In *International Conference on Artificial Intelligence and Statistics*, 2017.

G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson. TreeQN and ATreeC: Differentiable Tree-Structured Models for Deep Reinforcement Learning. In *International Conference on Learning Representations*, 2018.

N. Gopalan, M. Littman, J. MacGlashan, S. Squire, S. Tellex, J. Winder, L. Wong, et al. Planning with abstract markov decision processes. In *International Conference on Automated Planning and Scheduling*, 2017.

D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. Mastering Diverse Domains through World Models. *arXiv:2301.04104*, 2023.

M. Hauskrecht, N. Meuleau, L. P. Kaelbling, T. L. Dean, and C. Boutilier. Hierarchical solution of markov decision processes using macro-actions. In *Uncertainty in Artificial Intelligence*, 2013.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *IEEE International Conference on Computer Vision*, 2015.

C. Hogg, U. Kuter, and H. Muñoz-Avila. Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning. In *AAAI Conference on Artificial Intelligence*, 2010.

T. Jafferjee, E. Imani, E. Talvitie, M. White, and M. Bowling. Hallucinating Value: A Pitfall of Dyna-style Planning with Imperfect Environment Models. *arXiv:2006.04363*, 2020.

K. Javed and R. S. Sutton. The big world hypothesis and its ramifications for artificial intelligence. In *Finding the Frame Workshop at The Reinforcement Learning Conference*, 2024.

G. H. John. When the best move isn't optimal: Q-learning with exploration. In *AAAI*, volume 1464, 1994.

K. Khetarpal, Z. Ahmed, G. Comanici, D. Abel, and D. Precup. What can I do here? A Theory of Affordances in Reinforcement Learning. In *International Conference on Machine Learning*, 2020.

G. Konidaris. Constructing abstraction hierarchies using a skill-symbol loop. In *International Joint Conference on Artificial Intelligence*, 2016.

G. Konidaris and A. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. *Advances in neural information processing systems*, 22, 2009.

G. Konidaris, L. Kaelbling, and T. Lozano-Perez. Constructing symbolic representations for high-level planning. In *AAAI Conference on Artificial Intelligence*, 2014.

R. K. Kumaraswamy. *Sample-Efficient Control with Directed Exploration in Discounted MDPs Under Linear Function Approximation.* PhD thesis, University of Alberta, 2022.

N. Lambert, K. Pister, and R. Calandra. Investigating Compounding Prediction Errors in Learned Dynamics Models. *arXiv:2203.09637*, 2022.

C. Lo, G. Mihucz, A. White, F. Aminmansour, and M. White. Goal-space planning with subgoal models, 2022.

C. Lo, K. Roice, P. M. Panahi, S. Jordan, A. White, G. Mihucz, F. Aminmansour, and M. White. Goal-space planning with subgoal models, 2024.

T. A. Mann, S. Mannor, and D. Precup. Approximate Value Iteration with Temporally Extended Actions. *Journal of Artificial Intelligence Research*, 53, 2015.

A. McGovern and A. G. Barto. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. In *International Conference on Machine Learning*, 2001.

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

J. Oh, S. Singh, and H. Lee. Value Prediction Network. In *Advances in Neural Information Processing Systems*, 2017.

G. Ostrovski, M. G. Bellemare, A. Oord, and R. Munos. Count-based exploration with neural density models. In *International Conference on Machine Learning*, 2017.

A. Patterson, S. Neumann, M. White, and A. White. Empirical Design in Reinforcement Learning. *arXiv:2304.01315*, 2020.

R. Penrose. A Generalized Inverse for Matrices. *Mathematical Proceedings of the Cambridge Philosophical Society*, 51(3), 1955.

G. B. Peterson. A day of great illumination: Bf skinner's discovery of shaping. *Journal of the Experimental Analysis of Behavior*, 82(3):317–328, 2004.

M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

J. Randløv and P. Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *ICML*, volume 98, pages 463–471, 1998.

R. Rodriguez-Sanchez and G. Konidaris. Learning abstract world model for value-preserving planning with options. In *The Reinforcement Learning Conference*, 2024.

K. Roice, P. M. Panahi, S. M. Jordan, A. White, and M. White. A new view on planning in online reinforcement learning. In *Planning and Reinforcement Learning Workshop at International Conference for Automated Planning and Scheduling*, 2024.

G. A. Rummery. *Problem Solving with Reinforcement Learning*. PhD thesis, University of Cambridge, 1995.

G. A. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *Nature*, 588(7839), 2020.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347*, 2017.

D. Silver, H. Hasselt, M. Hessel, T. Schaul, A. Guez, T. Harley, G. Dulac-Arnold, D. Reichert, N. Rabinowitz, A. Barreto, and T. Degris. The Predictron: End-To-End Learning and Planning. In *International Conference on Machine Learning*, 2017.

B. F. Skinner. Reinforcement today. *American Psychologist*, 13(3):94, 1958.

M. Stolle and D. Precup. Learning Options in Reinforcement Learning. In *Abstraction, Reformulation, and Approximation*, 2002.

R. S. Sutton. Integrated modeling and control based on reinforcement learning and dynamic programming. In *Advances in Neural Information Processing Systems*, 1991.

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

R. S. Sutton, D. Precup, and S. P. Singh. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, 112(1-2), 1999.

R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. M. Pilarski, A. White, and D. Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768, 2011.

R. S. Sutton, R. A. Mahmood, and M. White. An Emphatic Approach to the Problem of Off-Policy Temporal-Difference Learning. *The Journal of Machine Learning Research*, 2016.

R. S. Sutton, M. C. Machado, G. Z. Holland, D. Szepesvari, F. Timbers, B. Tanner, and A. White. Reward-respecting subtasks for model-based reinforcement learning. *Artificial Intelligence*, 324:104001, 2023.

A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel. Value Iteration Networks. In *Advances in Neural Information Processing Systems*, 2016.

H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. In *AAAI Conference on Artificial Intelligence*, 2016.

H. van Hasselt, M. Hessel, and J. Aslanides. When to use Parametric Models in Reinforcement Learning? In *Advances in Neural Information Processing Systems*, 2019.

V. Veeriah. *Discovery in Reinforcement Learning*. PhD thesis, 2022.

Y. Wan and R. S. Sutton. Toward discovering options that achieve faster planning. *arXiv preprint arXiv:2205.12515*, 2022.

C. J. C. H. Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, 1989.

T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Silver, and D. Wierstra. Imagination-Augmented Agents for Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2017.

M. White. Unifying Task Specification in Reinforcement Learning. In *International Conference on Machine Learning*, 2017.

E. Wiewiora. Potential-based shaping and q-value initialization are equivalent. *Journal of Artificial Intelligence Research*, 19:205–208, 2003.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8, 1992.

J. Wolfe, B. Marthi, and S. Russell. Combined Task and Motion Planning for Mobile Manipulation. *International Conference on Automated Planning and Scheduling*, 2010.

Appendix

A Learning the Option Policies

In this section we detail the implementation of option learning which was used in all the experiments presented in this thesis. This is followed by a brief description of how these option policies could be learnt more generally across domains. Our full procedure is summarised in Figure B.2.

In the simplest case, it is enough to learn π_g that makes $r_\gamma(s, g)$ maximal for every relevant s (i.e., $\forall\, s \in \mathcal{S}$ s.t. $d(s, g) > 0$). For each subgoal g, we learn its corresponding option model π_g by initialising the base learner in the initiation set of g, and terminating the episode once the learner is in a state that is a member of g. We used a reward of -1 per step and save the option policy once we reach a 90% success rate, and the last 100 episodes are within some domain-dependent cut off. This cut off was 10 steps for FourRooms, and 50 steps for GridBall and PinBall.

We could have also learned the action-value variant $r_\gamma(s, a, g)$ using a Sarsa update, and set $\pi_g(s) = \text{argmax}_{a \in \mathcal{A}}\, r_\gamma(s, a, g)$, where we overloaded the definition of r_γ. We can then extract $r_\gamma(s, g) = \max_{a \in \mathcal{A}}\, r_\gamma(s, a, g)$, to use in all the above updates and in planning. In our experiments, this strategy is sufficient for learning π_g.

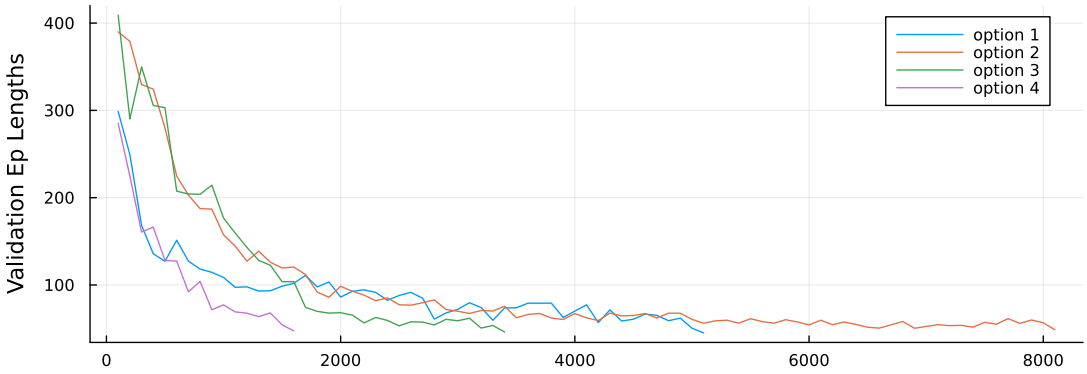


Figure A.1: Evaluation of PinBall option policies by average trajectory length. Policies were saved once they were able to reach their respective subgoal in undeer 50 steps, averaged across 100 trajectories. Subgoal 2 was the hardest to learn an option policy for, due to its proximity to obstacles.

B Learning the Subgoal Models

Now we need a way to learn the state-to-subgoal models, $r_\gamma(s,g)$ and $\Gamma(s,g)$, still following the progression in Figure B.2. These can both be expressed as General Value Functions (GVFs) (Sutton et al., 2011),

$$\Gamma(s,g) = \mathbb{E}_{\pi_g}\left[\sum_{k=0}^{\infty}\left(\prod_{k'=0}^{k}\gamma_{t+k'+1}\right) m(S_{t+1},g)\Big|S_t = s\right], \tag{1}$$

$$r_\gamma(s,g) = \mathbb{E}_{\pi_g}\left[\sum_{k=0}^{\infty}\left(\prod_{k'=0}^{k}\gamma_{t+k'+1}\right) R_{t+k+1}\Big|S_t = s\right], \tag{2}$$

and we leverage this form to use standard algorithms in RL to learn them.

In our experiments, the data is generated offline according to each π_g. We then use this episode dataset from each π_g to learn the subgoal models for that subgoal g. This is done by ordinary least squares regression to fit a linear model in four-room, and by stochastic gradient descent with neural network models in GridBall and PinBall. Full experimental details for these methods are described

in Appendix E.

Offline Model Update We first collect a dataset of n episodes leading to a subgoal g, $\mathcal{D}_g = \{\langle S_{i,1}, A_{i,1}, R_{i,1}, S_{i,1}, \ldots, S_{i,T_i}\rangle\}_{i=1}^{n}$. $S_{i,t}, A_{i,t}, R_{i,t}$ represent the state, action and reward at timestep t of episode i. T_i is the length of episode i. $S_{i,0}$ is a randomised starting state within the initiation set of g, and S_{i,T_i} is a state that is a member of subgoal g. For each g, we use $\mathcal{D}_g$ to generate a matrix of all visited states, $\mathbf{X} \in \mathbb{R}^{l \times |\mathcal{S}|}$, and a vector of all reward model returns, $\mathbf{g}_r \in \mathbb{R}^l$, and transition model returns $\mathbf{g}_\gamma \in \mathbb{R}^l$,

$$\mathbf{X} = \begin{pmatrix} S_{i,1} \\ S_{i,2} \\ \vdots \\ S_{n,T_n} \end{pmatrix}, \mathbf{g}_r = \begin{pmatrix} R_{i,2} + \gamma r_\gamma(S_{i,3}, g) \\ R_{i,3} + \gamma r_\gamma(S_{i,4}, g) \\ \vdots \\ R_{n,T_n} \end{pmatrix}, \mathbf{g}_\gamma = \begin{pmatrix} \gamma^{T_1 - 0} \\ \gamma^{T_1 - 1} \\ \vdots \\ \gamma^{T_n - T_n} \end{pmatrix},$$

where $l = \sum_{i=1}^{n} T_i$ is the total number of visited states in $\mathcal{D}_g$.

This creates a system of linear equations, whose weights we can solve for numerically in the four-room domain,

$$\mathbf{X}\boldsymbol{\theta}^r = \mathbf{g}_r \implies \boldsymbol{\theta}^r = \mathbf{X}^+ \mathbf{g}_r,$$
$$\mathbf{X}\boldsymbol{\theta}^\Gamma = \mathbf{g}_\gamma \implies \boldsymbol{\theta}^\Gamma = \mathbf{X}^+ \mathbf{g}_\gamma,$$

where $\boldsymbol{\theta}^r, \boldsymbol{\theta}^\Gamma \in \mathbb{R}^{|\mathcal{S}|}$ and $\mathbf{X}^+$ is the Moore-Penrose pseudoinverse of $\mathbf{X}$ (Penrose, 1955).

For GridBall and PinBall, we used fully connected artificial neural networks for r_γ and Γ, and performed mini-batch stochastic gradient descent to solve $\boldsymbol{\theta}^r$ and $\boldsymbol{\theta}^\Gamma$ for that subgoal g. We use each mini-batch of m states, reward model

returns and transition model returns to perform the update:

$$\boldsymbol{\theta}^r \leftarrow \boldsymbol{\theta}^r - \eta_r \sum_{j=1}^{m} \nabla_{\boldsymbol{\theta}^r} (\boldsymbol{\theta}^{r\top} \mathbf{X}_{j,:} - \mathbf{g}_{r,j})^2,$$

$$\boldsymbol{\theta}^\Gamma \leftarrow \boldsymbol{\theta}^\Gamma - \eta_\Gamma \sum_{j=1}^{m} \nabla_{\boldsymbol{\theta}^\Gamma} (\boldsymbol{\theta}^{\Gamma\top} \mathbf{X}_{j,:} - \mathbf{g}_{\gamma,j})^2,$$

where η_r and η_Γ are the learning rates for the reward and discount models respectively. $\mathbf{X}_{j,:}$ is the j^{th} row of $\mathbf{X}$. $\mathbf{g}_{r,j}$ and $\mathbf{g}_{\gamma,j}$ are the j^{th} entry of $\mathbf{g}_r$ and $\mathbf{g}_\gamma$ respectively. In our experiments, we had a fully connected artificial neural network with two hidden layers of 128 units and ReLU activation for each subgoal. The network took a state $s = (x, y, \dot{x}, \dot{y})$ as input and outputted both $r_\gamma(s, g)$ and $\Gamma(s, g)$. All weights were initialised using Kaiming initialisation (He et al., 2015). We use the Adam optimizer with $\eta = 0.001$ and the other parameters set to the default ($b_1 = 0.9, b_2 = 0.999, \epsilon = 10^{-8}$), mini-batches of 1024 transitions and 100 epochs.

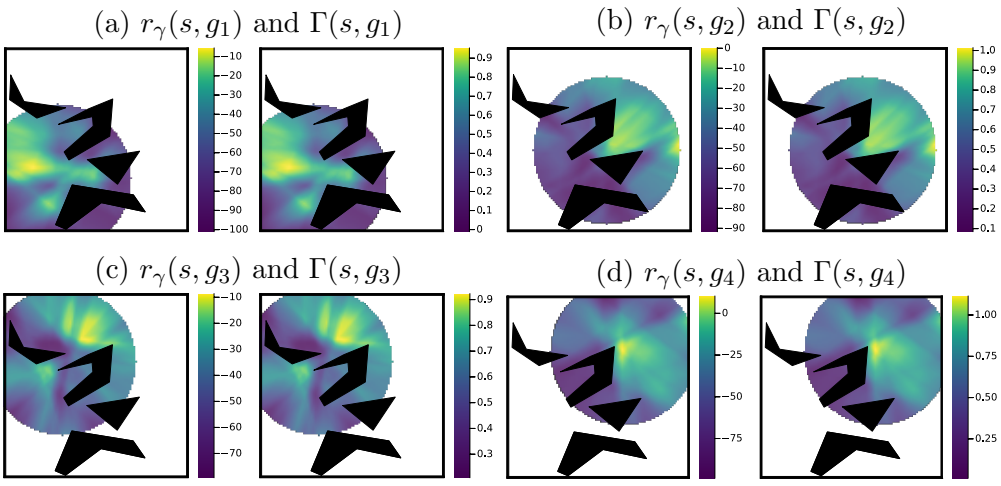


Figure B.1: State-to-Subgoal models learnt by neural models after 100 epochs.

The data could also be generated off-policy—according to some behavior b rather than from π_g. We can either use importance sampling or we can learn the

action-value variants of these models to avoid importance sampling. We describe both options here.

Off-policy Model Update using Importance Sampling We can update $r_\gamma(\cdot, g)$ with an importance-sampled temporal difference (TD) learning update $\rho_t \delta_t \nabla r_\gamma(S_t, g)$ where $\rho_t = \frac{\pi_g(a|S_t)}{b(a|S_t)}$ and

$$\delta_t^r = R_{t+1} + \gamma_{g,t+1} r_\gamma(S_{t+1}, g) - r_\gamma(S_t, g)$$

The discount model $\Gamma(s, g)$ can be learned similarly, because it is also a GVF with cumulant $m(S_{t+1}, g)\gamma_{t+1}$ and discount $\gamma_{g,t+1}$. The TD update is $\rho_t \delta_t^\Gamma$ where

$$\delta_t^\Gamma = m(S_{t+1}, g)\gamma_{t+1} + \gamma_{g,t+1}\Gamma(S_{t+1}, g) - \Gamma(S_t, g)$$

All of the above updates can be done using any off-policy GVF algorithm, including those using clipping of IS ratios and gradient-based methods, and can include replay.

Off-policy Model Update without Importance Sampling Overloading notation, let us define the action-value variants $r_\gamma(s, a, g)$ and $\Gamma(s, a, g)$. We get similar updates to above, now redefining

$$\delta_t^r = R_{t+1} + \gamma_{g,t+1} r_\gamma(S_{t+1}, \pi_g(S_{t+1}), g) - r_\gamma(S_t, A_t, g)$$

and using update $\delta_t \nabla r_\gamma(S_t, A_t, g)$. For Γ we have

$$\delta_t^\Gamma = m(S_{t+1}, g)\gamma_{t+1} + \gamma_{g,t+1}\Gamma(S_{t+1}, \pi_g(S_{t+1}), g) - \Gamma(S_t, A_t, g)$$

We then define $r_\gamma(s,g) \doteq r_\gamma(s,\pi_g(s),g)$ and $\Gamma(s,g) \doteq \Gamma(s,\pi_g(s),g)$ as deterministic functions of these learned functions.

Restricting the Model Update to Relevant States Recall, however, that we need only query these models where $d(s,g) > 0$. We can focus our function approximation resources on those states. This idea has previously been introduced with an interest weighting for GVFs (Sutton et al., 2016), with connections made between interest and initiation sets (White, 2017). For a large state space with many subgoals, using goal-space planning significantly expands the models that need to be learned, especially if we learn one model per subgoal. Even if we learn a model that generalizes across subgoal vectors, we are requiring that model to know a lot: values from all states to all subgoals. It is likely such a models would be hard to learn, and constraining what we learn about with $d(s,g)$ is likely key for practical performance.

The modification to the update is simple: we simply do not update $r_\gamma(s,g), \Gamma(s,g)$ in states s where $d(s,g) = 0$.[1]For the action-value variant, we do not update for state-action pairs (s,a) where $d(s,g) = 0$ and $\pi_g(s) \neq a$. The model will only ever be queried in (s,a) where $d(s,g) = 1$ and $\pi_g(s) = a$.

Learning the relevance model d We assume in this work that we simply have $d(s,g)$, but we can at least consider ways that we could learn it. One approach is to attempt to learn Γ for each g, to determine which are pertinent. Those with $\Gamma(s,g)$ closer to zero can have $d(s,g) = 0$. In fact, such an approach was taken for discovering options (Khetarpal et al., 2020), where both options and such a

[1]More generally, we could use *emphatic weightings* (Sutton et al., 2016) that allow us to incorporate such interest weightings $d(s,g)$, without suffering from bootstrapping off of inaccurate values in states where $d(s,g) = 0$. Incorporating this algorithm would likely benefit the whole system, but we keep things simpler for now and stick with a typical TD update.

relevance function are learned jointly. For us, they could also be learned jointly, where a larger set of goals start with $d(s, g) = 1$, then if $\Gamma(s, g)$ remains small, then these may be switched to $d(s, g) = 0$ and they will stop being learned in the model updates.

Learning the Subgoal-to-Subgoal Models Finally, we need to extract the subgoal-to-subgoal models $\tilde{r}_\gamma, \tilde{\Gamma}$ from r_γ, Γ. These models were defined as means of the GVFs taken over member states of each subgoal, as specified in Equation 3.3. The strategy involves updating towards the state-to-subgoal models, whenever a state corresponds to a subgoal. In other words, for a given s, if $m(s, g) = 1$, then for a given g' (or iterating through all of them), we can update $\tilde{r}_\gamma$ using

$$(r_\gamma(s, g') - \tilde{r}_\gamma(g, g'))\nabla\tilde{r}_\gamma(g, g'),$$

and update $\tilde{\Gamma}$ using

$$(\Gamma(s, g') - \tilde{\Gamma}(g, g'))\nabla\tilde{\Gamma}(g, g').$$

Note that these updates are not guaranteed to uniformly weight the states where $m(s, g) = 1$. Instead, the implicit weighting is based on sampling s, such as through which states are visited and in the replay buffer. We do not attempt to correct this skew, as mentioned in the main body, we presume that this bias is minimal. An important next step is to better understand if this lack of reweighting causes convergence issues, and how to modify the algorithm to account for a potentially changing state visitation.

Computing $v_{g^\star}$ In order to compute $v_{g^\star}$, we first need a $\tilde{v}$ from our abstract MDP to look up the subgoal values. We compute $\tilde{v}$ by value iteration in the

abstract MDP with a tolerance of $\epsilon = 10^{-8}$ and maximum of 10,000 iterations. The resulting $\tilde{v}$ from these subgoal models was used in the projection step to obtain $v_{g^\star}$, by iterating over relevant subgoals as described in Equation (3.5).

C Putting it all together

We summarize the above updates in pseudocode, specifying explicit parameters and how they are updated. The algorithm is summarized in Algorithm 1. An online update is used for the action-values for the main policy, without replay. All background computation is used for model learning using a replay buffer and for planning with those models. The pseudocode assumes a small set of subgoals, and is for episodic problems. We provide extensions to the DDQN setting in Section 4.2, including using a Double DQN update for the policy update. We also discuss in-depth differences to existing related ideas, including landmark states and UVFAs.

Note that we overload the definitions of the subgoal models. We learn action-value variants $r_\gamma(s, a, g; \boldsymbol{\theta}^r)$, with parameters $\boldsymbol{\theta}^r$, to avoid importance sampling corrections. We learn the option-policy using action-values $\tilde{q}(s, a; \boldsymbol{\theta}^\pi)$ with parameters $\boldsymbol{\theta}^\Gamma$, and so query the policy using $\pi_g(s; \boldsymbol{\theta}^\pi) \doteq \text{argmax}_{a \in \mathcal{A}} \tilde{q}(s, a, g; \boldsymbol{\theta}^\pi)$. The policy π_g is not directly learned, but rather defined by $\tilde{q}$. Similarly, we do not directly learn $r_\gamma(s, g)$; instead, it is defined by $r_\gamma(s, a, g; \boldsymbol{\theta}^r)$. Specifically, for model parameters $\boldsymbol{\theta} = (\boldsymbol{\theta}^r, \boldsymbol{\theta}^\Gamma, \boldsymbol{\theta}^\pi)$, we set $r_\gamma(s, g; \boldsymbol{\theta}) \doteq r_\gamma(s, \pi_g(s; \boldsymbol{\theta}^\pi), g; \boldsymbol{\theta}^r)$ and $\Gamma(s, g; \boldsymbol{\theta}) \doteq \Gamma(s, \pi_g(s; \boldsymbol{\theta}^\pi), g; \boldsymbol{\theta}^\Gamma)$. We query these derived functions in the pseudocode.

Finally, we assume access to a given set of subgoals. But there have been several natural ideas already proposed for option discovery, that nicely apply in

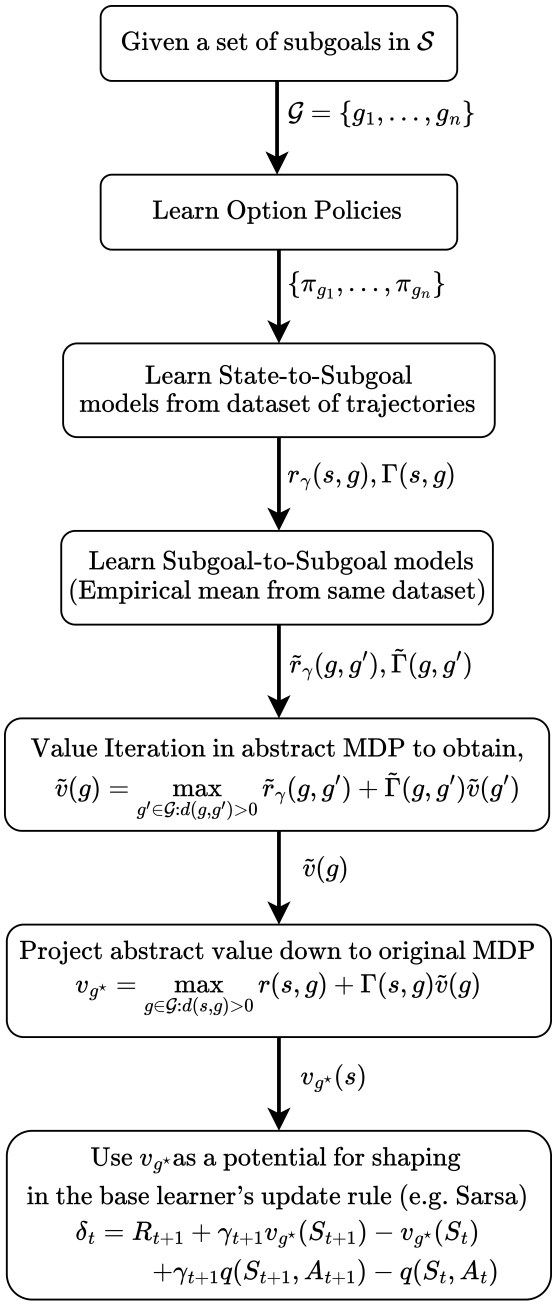


Figure B.2: Learning and using pre-trained models for GSP.

our more constrained setting. One idea was to use subgoals that are often visited by the agent (Stolle and Precup, 2002). Such a simple idea is likely a reasonable starting point to make a GSP algorithm that learns everything from scratch, including subgoals. Other approaches have used bottleneck states (McGovern and Barto, 2001).

C.1 GSP Pseudocode

This subsection shows the full pseudocode for vanilla GSP from Lo et al. (2022).

Algorithm 4 `MainPolicyUpdate`$(s, a, s', r, \gamma, a')$

// For a Sarsa(λ) base learner
$v_{g^\star} \leftarrow \max_{g \in \bar{\mathcal{G}}: d(s,g)>0} r_\gamma(s, g; \boldsymbol{\theta}) + \Gamma(s, g; \boldsymbol{\theta}) \tilde{v}(g)$
$\delta \leftarrow r + \gamma v_{g^\star}(s') - v_{g^\star}(s) + \gamma q(s', a'; \mathbf{w}) - q(s, a; \mathbf{w})$
$\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z} \nabla_{\mathbf{w}} q(s, a; \mathbf{w})$
$\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + \nabla_{\mathbf{w}} q(s, a; \mathbf{w})$

Algorithm 5 `Update_Models`(s, a, s', r, γ)

Add new transition (s, a, s', r, γ) to buffer B
for $g' \in \bar{\mathcal{G}}$, for multiple transitions (s, a, r, s', γ) sampled from B **do**
  $\gamma_{g'} \leftarrow \gamma(1 - m(s', g'))$
  // Update option policy - e.g. by Sarsa
  $a' \leftarrow \pi_{g'}(s'; \boldsymbol{\theta}^\pi)$
  $\delta^\pi \leftarrow \frac{1}{2}(r - 1) + \gamma_{g'} \tilde{q}(s', a', g'; \boldsymbol{\theta}^\pi) - \tilde{q}(s, a, g'; \boldsymbol{\theta}^\pi)$
  $\boldsymbol{\theta}^\pi \leftarrow \boldsymbol{\theta}^\pi + \alpha^\pi \delta^\pi \nabla_{\boldsymbol{\theta}^\pi} q(s, a, g'; \boldsymbol{\theta}^\pi)$
  // Update reward model and discount model
  $\delta^r \leftarrow r + \gamma_{g'} r_\gamma(s', a', g'; \boldsymbol{\theta}^r) - r_\gamma(s, a, g'; \boldsymbol{\theta}^r)$
  $\delta^\Gamma \leftarrow m(s', g)\gamma + \gamma_{g'} \Gamma(s', a', g'; \boldsymbol{\theta}^\Gamma) - \Gamma(s, a, g'; \boldsymbol{\theta}^\Gamma)$
  $\boldsymbol{\theta}^r \leftarrow \boldsymbol{\theta}^r + \alpha^r \delta^r \nabla_{\boldsymbol{\theta}^r} r_\gamma(s, a, g'; \boldsymbol{\theta}^r)$
  $\boldsymbol{\theta}^\Gamma \leftarrow \boldsymbol{\theta}^\Gamma + \alpha^\Gamma \delta^\Gamma \nabla_{\boldsymbol{\theta}^\Gamma} \Gamma(s, a, g'; \boldsymbol{\theta}^\Gamma)$
  // Update goal-to-goal models using state-to-goal models
  for each g such that $m(s, g) > 0$ **do**
    $\tilde{\boldsymbol{\theta}}^r \leftarrow \tilde{\boldsymbol{\theta}}^r + \tilde{\alpha}^r (r_\gamma(s, g'; \boldsymbol{\theta}) - \tilde{r}_\gamma(g, g'; \tilde{\boldsymbol{\theta}}^r)) \nabla_{\boldsymbol{\theta}^r} \tilde{r}_\gamma(g, g'; \tilde{\boldsymbol{\theta}}^r)$
    $\tilde{\boldsymbol{\theta}}^\Gamma \leftarrow \tilde{\boldsymbol{\theta}}^\Gamma + \tilde{\alpha}^\Gamma (\Gamma(s, g'; \boldsymbol{\theta}) - \tilde{\Gamma}(g, g'; \tilde{\boldsymbol{\theta}}^r)) \nabla_{\boldsymbol{\theta}^\Gamma} \tilde{\Gamma}(g, g'; \tilde{\boldsymbol{\theta}}^\Gamma)$

It is simple to extend the above pseudocode for the main policy update and the option policy update to use Double DQN (van Hasselt et al., 2016) updates with neural networks. The changes from the above pseudocode are 1) the use of a target network to stabilize learning with neural networks, 2) changing the one-step bootstrap target to the DDQN equivalent, 3) adding a replay buffer for learning the main policy, and 4) changing the update from using a single sample to using a batch update. Because the number of subgoals is discrete, the equations for learning $\tilde{\boldsymbol{\theta}}^r$ and $\tilde{\boldsymbol{\theta}}^\Gamma$ does not change. We previously summarized these changes for learning the main policy in Algorithm 3 and now detail the subgoal model learning in Algorithm 6.

Algorithm 6 `Update_GSP_Models`(s, a, s', r, γ)

Add new transition (s, a, s', r, γ) to buffer D_{model}
for $g' \in \bar{\mathcal{G}}$ **do**
  for n_{model} mini-batches **do**
    Sample batch $B_{\text{model}} = \{(s, a, r, s', \gamma)\}$ from D_{model}
    $\gamma_{g'} \leftarrow \gamma(1 - m(s', g'))$
    // Update option policy
    $a' \leftarrow \text{argmax}_{a' \in \mathcal{A}} \tilde{q}(s', a', g'; \boldsymbol{\theta}^\pi)$
    $\delta^\pi(s, a, s', r, \gamma) \leftarrow \frac{1}{2}(r - 1) + \gamma_{g'} \tilde{q}(s', a', g'; \boldsymbol{\theta}^\pi_{\text{targ}}) - q(s, a, g'; \boldsymbol{\theta}^\pi)$
    $\boldsymbol{\theta}^\pi \leftarrow \boldsymbol{\theta}^\pi - \alpha^\pi \nabla_{\boldsymbol{\theta}^\pi} \frac{1}{|B_{\text{model}}|} \sum_{(s,a,r,s',\gamma) \in B_{\text{model}}} (\delta^\pi)^2$
    ${\boldsymbol{\theta}^\pi}_{\text{targ}} \leftarrow \rho_{\text{model}} \boldsymbol{\theta}^\pi + (1 - \rho_{\text{model}}) {\boldsymbol{\theta}^\pi}_{\text{targ}}$
    // Update reward model and discount model
    $\delta^r(s, a, r, s', \gamma) \leftarrow r + \gamma_{g'}(\gamma, s') r_\gamma(s', a', g'; {\boldsymbol{\theta}^r}_{\text{targ}}) - r_\gamma(s, a, g'; \boldsymbol{\theta}^r)$
    $\delta^\Gamma(s, a, r, s', \gamma) \leftarrow m(s', g)\gamma + \gamma_{g'}(\gamma, s') \Gamma(s', a', g'; {\boldsymbol{\theta}^\Gamma}_{\text{targ}}) - \Gamma(s, a, g'; \boldsymbol{\theta}^\Gamma)$
    $\boldsymbol{\theta}^r \leftarrow \boldsymbol{\theta}^r - \alpha^r \nabla_{\boldsymbol{\theta}^r} \frac{1}{|B_{\text{model}}|} \sum_{(s,a,r,s',\gamma) \in B_{\text{model}}} (\delta^r)^2$
    $\boldsymbol{\theta}^\Gamma \leftarrow \boldsymbol{\theta}^\Gamma - \alpha^\Gamma \nabla_{\boldsymbol{\theta}^\Gamma} \frac{1}{|B_{\text{model}}|} \sum_{(s,a,r,s',\gamma) \in B_{\text{model}}} (\delta^\Gamma)^2$
    if $n_{\text{updates}} \% \tau == 0$ **then**
      ${\boldsymbol{\theta}^r}_{\text{targ}} \leftarrow \boldsymbol{\theta}^r$
      ${\boldsymbol{\theta}^\Gamma}_{\text{targ}} \leftarrow \boldsymbol{\theta}^\Gamma$
    $n_{\text{updates}} = n_{\text{updates}} + 1$
  // Update goal-to-goal models using state-to-goal models
  ... same as in prior pseudocode.

C.2 Optimizations for GSP using Fixed Models

It is possible to reduce computation cost of GSP when learning with a fixed model. When the subgoal models are fixed, $v_{g^\star}$ for an experience sample does not change over time as all components that are used to calculate $v_{g^\star}$ are fixed. This means that the agent can calculate $v_{g^\star}$ when it first receives the experience sample and save it in the buffer, and use the same calculated $v_{g^\star}$ whenever this sample is used for updating the main policy. When doing so, $v_{g^\star}$ only needs to be calculated once per sample experienced, instead of with every update. This is beneficial when training neural networks, where each sample is often used multiple times to update network weights.

An additional optimization possible on top of caching of $v_{g^\star}$ in the replay buffer is that we can batch the calculation of $v_{g^\star}$ for multiple samples together, which can be more efficient than calculating $v_{g^\star}$ for a single sample every step. To do this, we create an intermediate buffer that stores up to some number of samples. When the agent experiences a transition, it adds the sample to this intermediate buffer rather than the main buffer. When this buffer is full, the agent calculates $v_{g^\star}$ for all samples in this buffer at once and adds the samples alongside $v_{g^\star}$ to the main buffer. This intermediate buffer is then emptied and added to again every step. We set the maximum size for the intermediate buffer to 1024 in our experiments.

D An Alternative way of using $v_{g^\star}$

We used $v_{g^\star}$ through potential-based reward shaping, but other approaches are possible. For example, another approach is to solely bootstrap off of the predic-

tion from $v_{g^\star}$, instead of the base learner's q estimate,

$$R_{t+1} + \gamma_{t+1} v_{g^\star}(S_{t+1}) - q(S_t, A_t; \mathbf{w}). \tag{3}$$

The update with this TD error is reminiscent of an algorithm called Landmark Approximate Value Iteration (LAVI) (Mann et al., 2015). LAVI is designed for the setting where a model, or simulator, is given. Similar to GSP, the algorithm plans only over a set of landmarks (subgoals). They assume that they have options that terminate near the landmarks, and do value iteration with the simulator by executing options from only the landmarks. The greedy policy for a state uses the computed values for landmark states by selecting the option that takes the agent to the best landmark state, and using options to move only between landmark states from there. The planning is much more efficent, because the number of landmark states is relatively small, but the policies are suboptimal.

We could similarly use $v_{g^\star}$, by running the option to bring the agent to the best nearby subgoal. However, a more direct comparison in our setting is to use the modified TD error update above. We call this update Approximate LAVI, to recognize the similarity to this elegant algorithm. In all environments, the approximate LAVI learner either learns much slower or converges to a sub-optimal policy instead.

In our preliminary experiments, we had investigated an update rule that partially bootstraps off $v_{g^\star}$. Namely, we used a TD error of $R_{t+1} + \gamma_{t+1}(\beta v_{g^\star}(S_{t+1}) + (1-\beta) q(S_{t+1}, A_{t+1})) - q(S_t, A_t)$, where $\beta \in [0, 1]$. Potential based reward shaping with $v_{g^\star}$ was found to outperform this technique. We discuss this more in Appendix D.

(a) FourRooms (b) GridBall (c) PinBall

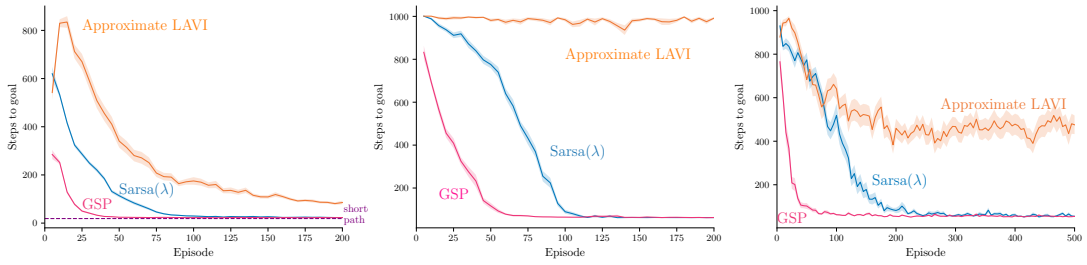
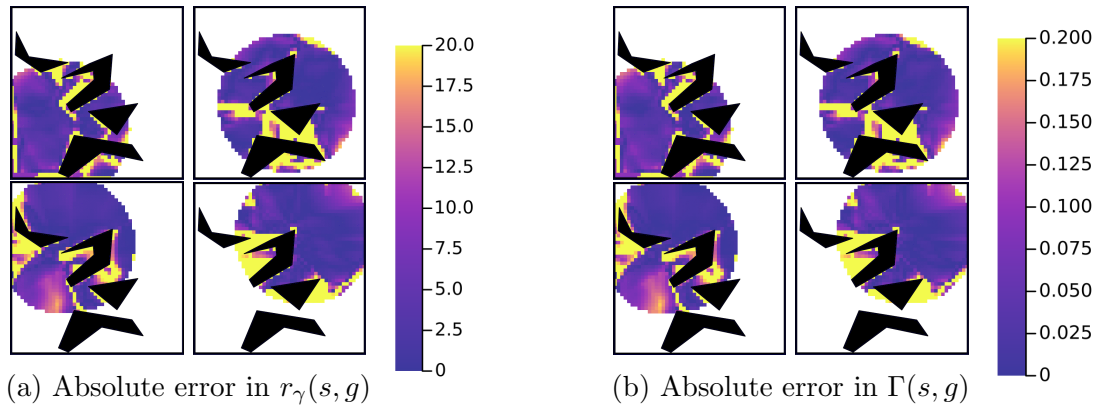


Figure D.1: Five episode moving average of return in FourRooms, GridBall and PinBall. Curves are averaged over 30 runs where the shaded region is one standard error.

E Errors in Learned Subgoal Models



(a) Absolute error in $r_\gamma(s, g)$ (b) Absolute error in $\Gamma(s, g)$

Figure E.1: Model errors in State-to-Subgoal models used in GridBall.

To better understand the accuracy of our learned subgoal models, we performed roll-outs of the learned option policy at different (x, y) locations on GridBall and compared the true r_γ and Γ with the estimated values. Figure E.1 shows a heatmap of the absolute error of the model compared to the ground truth, with the mapping of colors on the right. The error in each pixel was computed by rolling out episodes from that state and logging the actual reward and discounted probability of reaching the subgoal. The models tend to be more ac-

curate in regions that are clear of obstacles, and less near these obstacles or near the boundary of the initiation set. The distribution of error over the initiation set is very similar for both r and Γ models. While the magnitudes of errors are not unreasonable, they are also not very near zero. This results is encouraging in that inaccuracies in the model do not prevent useful planning.

Epochs	**Mean Squared Error across models**
2	0.608
4	0.464
10	0.334

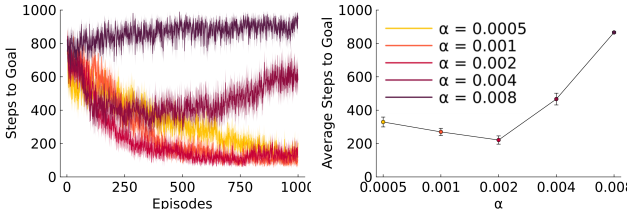
Table 1: Mean squared error across state-to-subgoal models used in PinBall.

F Hyperparameter Sweeps

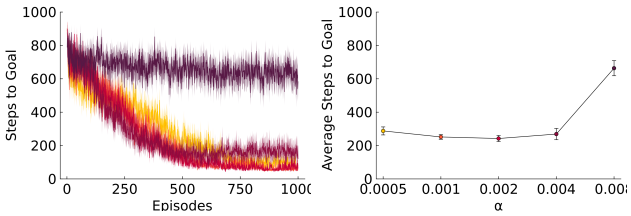
In FourRooms, we use Sarsa(0) and Sarsa(0.9) base learners with learning rate $\alpha = 0.01$, discount factor $\gamma_c = 0.99$ and an $\epsilon = 0.02$ for its ϵ-greedy policy. In GridBall, we used Sarsa(0) and Sarsa(0.9) base learners with $\alpha = 0.05$, $\gamma_c = 0.99$ and $\epsilon = 0.1$. ϵ is decayed by 0.5% each timestep. In the linear function approximation setting, these learners use a tilecoder with 16 tiles and 4 tilings across each of the both the GridBall dimensions. In PinBall, the Sarsa(0.9) learner was tuned to $\alpha = 0.1$, $\gamma_c = 0.99$, $\epsilon = 0.1$, decayed in the same manner as in GridBall. The same tile coder was used on on the 4-dimensional state space of PinBall. For the DDQN base learner, we use $\alpha = 0.004$, $\gamma_c = 0.99$, $\epsilon = 0.1$, a buffer that holds up to $10,000$ transitions a batch size of 32, and a target refresh rate of every 100 steps. The Q-Network weights used Kaiming initialisation (He et al., 2015).

For Sarsa(λ), we swept it's learning rate over $[0.001, 0.01, 0.05\ 0.01, 0.5, 0.9]$. 0.01, 0.05 and 0.1 were found to be the best for FourRooms, GridBall and PinBall

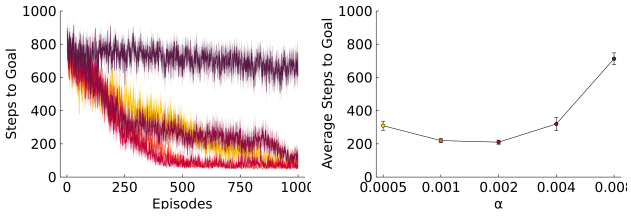
respectively. For DDQN, we swept its learning rate α over $[5 \times 10^{-4}, 1 \times 10^{-3}, 2 \times 10^{-3}, 4 \times 10^{-3}, 5 \times 10^{-3}]$ and target refresh rate τ over $[1, 50, 100, 200, 1000]$ as shown in Figure F.1.



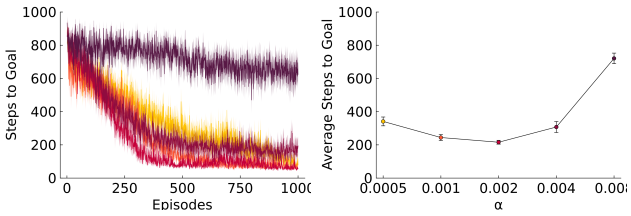
(a) $\tau = 1$



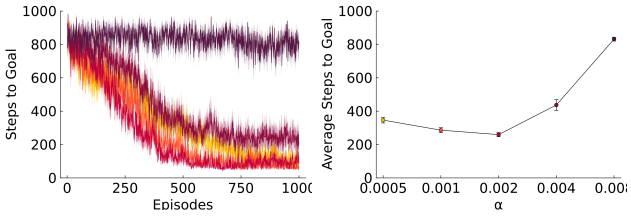
(b) $\tau = 50$



(c) $\tau = 100$



(d) $\tau = 200$



(e) $\tau = 1000$

Figure F.1: Left Column: learning curves for five different step sizes, α, averaged over 30 runs. Right Column: sensitivity to different step sizes. Each dot represents the steps to goal averaged over 30 runs and 1000 episodes. The error bars show one standard error. The refresh rate τ increases with each row.