Goal-Space Planning with Subgoal Models

by

Chunlok Lo

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

This thesis investigates a new approach to model-based reinforcement learning using background planning: mixing (approximate) dynamic programming updates and model-free updates, similar to the Dyna architecture. Background planning with learned models is often worse than model-free alternatives, such as Double DQN, even though the former uses significantly more memory and computation. The fundamental problem is that learned models can be inaccurate and often generate invalid states, especially when iterated many steps. In this work, we avoid this limitation by constraining background planning to a set of (abstract) subgoals and learning only local, subgoal-conditioned models. This goal-space planning (GSP) approach is more computationally efficient, naturally incorporates temporal abstraction for faster long-horizon planning and avoids learning the transition dynamics entirely. We show that our GSP algorithm can learn significantly faster than a Double DQN baseline in a variety of situations.

Preface

This thesis is based on work that is currently pending review at the time of writing, and was done in collaboration with Gábor Mihucz, Farzane Aminmasour, Adam White, and Martha White.

The root of all evil is in always relying on one of your other possibilities to get your wish. You must accept that you are the person here, now, and that you cannot become anyone else other than that person.

– Higuchi Seitarou, *Tatami Galaxy.*

Acknowledgements

I would like to thank my supervisor, Martha White, for her continued mentorship. She taught me the importance of communication and of understanding one another in research and in life, and pushed me to be a more empathetic and collaborative researcher. I am also the most grateful for the guidance she provided when the project direction seems unclear, pointing at questions to ask for the next steps, which I try to learn from to become a more independent researcher. I would also like to thank Adam White, who challenged me to be a clearer thinker and encouraged me to pursue research questions, and whose guidance has been invaluable in this project.

I would also like to thank Gábor Mihucz and Farzane Aminmansour, with whom I worked closely with and discussed with the ideas of this thesis. Their friendship and collaboration were indispensable during my time working on this project through both the highs and lows, and my time at the university overall.

Finally, I would extend my gratitude to the many individuals of the RLAI lab, and those who are part of Amii at large, who contributed to creating a welcoming and diverse community of researchers that continues to inspire me to push the boundaries of knowledge through research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The core idea behind reinforcement learning (RL) is learning from experience, the interaction between an agent and the environment. RL algorithms are often categorized into two groups: model-free and model-based RL algorithms. Model-free algorithms learn directly from experience, whereas model-based algorithms also uses experience to learn a model, an object that predicts the results of actions in the environment, and through a computational process called planning, uses knowledge from the model to improve the agent's actions.

While historically, model-free algorithms tend to work better than model-based alternatives, model-based RL is a key research area because learning a model provides the opportunity to better store, query, and generalize across experiences when compared to model-free algorithms. One basic form of a model, the *world model*, allows the agent to predict the results of its action at various states without actually performing the action in the environment. An agent can then use this model to simulate data in the background during the agent's interaction to improve the agent's policy, a process called *background planning*. Dyna [41] is a classic example of background planning. On each step, the agent simulates several transitions according to its model, and updates with those transitions as if they were real experience. Learning and using such a model is worthwhile in vast or ever-changing environments, where the agent learns over a long time period and can benefit from re-using knowledge about the environment.

The promise of Dyna is that we can exploit the Markov structure in the

RL formalism, to learn and adapt value estimates efficiently, but many open problems remain to make it more widely useful. These include that (1) one-step models learned in Dyna can be difficult to use for long-horizon planning, (2) learning probabilities over outcome states can be complex, especially for high-dimensional states and (3) planning itself can be computationally expensive for large state spaces.

A variety of strategies have been proposed to improve long-horizon planning. Incorporating options as additional (macro) actions in planning is one approach. An *option* is a policy coupled with a termination condition and initiation set [37]. They provide temporally-extended ways of behaving, allowing the agent to reason about outcomes further into the future. Incorporating options into planning is a central motivation of this work, particularly how to do so under function approximation. Options for planning has largely only been tested in tabular settings [37, 34, 49]. Recent work has considered mechanisms for identifying and learning option policies for planning under function approximation [40], but as yet did not consider issues with learning the models.

A variety of other 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 [17, 47, 22]. This problem has been tackled by using reverse models [28, 17, 47]; primarily using the model for decision-time planning [47, 32, 5]; and improving training strategies to account for accumulated errors in rollouts [43, 48, 44]. An emerging trend is to avoid approximating the true transition dynamics, and instead learn dynamics tailored to predicting values on the next step correctly [10, 9, 2]. 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 [45, 33, 27, 50, 11, 31]. We similarly attempt to avoid issues with iterating models, but do so by considering a different type of model.

Much less work has been done for the third problem in Dyna: the expense of planning. There is, however, a large literature on approximate dynamic programming—where the model is given—that is focused on efficient planning

(see [29]). Particularly relevant to this work is restricting value iteration to a small subset of landmark states [23].[1] The resulting policy is suboptimal, restricted to going between these landmark states, but planning is provably much more efficient.

Beyond this planning setting where the model is given, the use of landmark states has also been explored in *goal-conditioned RL*, where the agent is given a desired goal state or states. The first work to exploit this idea in reinforcement learning with function approximation, when learning online, was for learning universal value function approximators (UVFAs) [16]. The UVFA conditions action-values on both state-action pairs as well as landmark states. A search is done on a learned graph between landmark states, to identify which landmark to moves towards. A flurry of work followed, still in the goal-conditioned setting [26, 8, 54, 53, 1, 14, 12, 19, 7].

In this work, we exploit the idea behind landmark states for efficient background planning in general online reinforcement learning problems.

1. We introduce *subgoal-conditioned models*: temporally-extended models that condition on subgoals. Subgoal-conditioned models are designed to be simpler to learn when compared to traditional world models, as they are only learned for states local to subgoals and they avoid generating entire next state vectors. We show that these models can be formulated as general value functions and learned with standard off-policy TD algorithms.

2. We develop a simple value iteration algorithm to perform planning only over subgoals to quickly obtain subgoal values using subgoal-conditioned models.

3. We propose a novel bootstrapping update for the main policy, which we call subgoal-value bootstrapping, that leverages these quickly computed

[1] A similar idea to landmark states has been considered in more classical AI approaches, under the term bi-level planning [52, 15, 6]. These techniques are built on logical operators, and are quite different from the statistical foundations of Dyna-style planning—updating values with (stochastic) dynamic programming updates—and so we do not consider them further here.

subgoal values, but mitigates suboptimality by incorporating an update on real experience.

4. We introduce the Goal-Space Planning (GSP) algorithm that puts it all together: acting in the real world, learning models, background planning and updating the value function using subgoal-value bootstrapping.

5. We provide empirical insights into the characteristics and performance of GSP, the accuracy of models required to have good performance, and its performance in non-stationary environments when function approximation is needed. We find that GSP is able to speed up learning significantly with pre-learned models, is relatively robust to model inaccuracies, and can handle a changing environment more quickly when compared to a baseline model-free algorithm.

This thesis is divided into 5 chapters. Chapter 2 provides an overview of relevant background knowledge. Chapter 3 describes our Goal-Space Planning algorithm and its various components, and extensions to the deep RL setting. Chapter 4 details the experimental studies performed to investigate the behaviour and performance of GSP. Finally, Chapter 5 concludes the thesis and discusses possible future work.

Chapter 2

Background

In this chapter, we briefly cover concepts in reinforcement learning that are relevant to this thesis.

2.1 Markov Decision Process

Reinforcement learning is a field which studies how an *agent* should interact and learn to act through trial and error interaction with an *environment* to achieve an objective. Typically, this objective is to maximize the accumulation of a special signal called the *reward*. To maximize this cumulative reward, a quantity which is called the *value*, the agent might need to make decisions that might not maximize the immediate reward to receive more reward far into the future.

A common way to formalize this learning problem is through a Markov Decision Process (MDP). An MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P})$. $\mathcal{S}$ is the state space and $\mathcal{A}$ the action space. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ and the transition probability $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ describes the expected reward and probability of transitioning to a state, for a given state and action. On each discrete time step $t \in 0, 1, 2, ...$, the agent is at state S_t and selects an action A_t. Based on this action, the environment transitions to a new state $S_{t+1} \sim \mathcal{P}(S_t, A_t, \cdot)$ and emits a scalar reward $R_{t+1} = \mathcal{R}(S_t, A_t, S_{t+1})$. This process starts with the agent starting at some state S_0, and is then repeat forever, or until the agent reaches a terminal state, after which the episode ends and the agent begins again at some state S_0 based on a start state distribution.

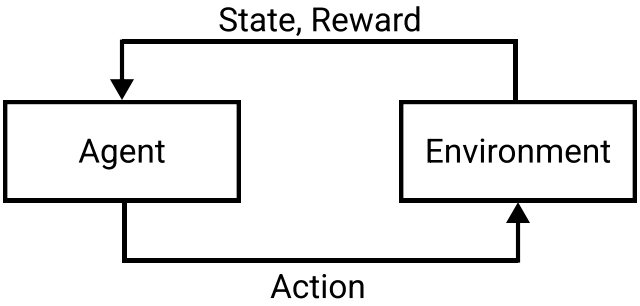

Figure 2.1: Diagram of the reinforcement learning interaction loop.

The agent's objective is to find a policy $\pi : S \times A \to [0, 1]$, where if the agent were to take action a at state s with probability $\pi(a|s)$ at all states, would maximize the expected *return*, the future discounted reward G_t, at each time step:

$$G_t \doteq R_{t+1} + \gamma_{t+1}R_{t+2} + \gamma_{t+1}\gamma_{t+2}R_{t+3} + \cdots = \sum_{k=0}^{\infty} R_{t+k+1} \prod_{j=1}^{k} \gamma^{t+k}$$

The state-based discount $\gamma_{t+1} \in [0, 1]$ depends on S_{t+1} [38], which allows us to specify termination. If S_{t+1} is a terminal state, then $\gamma_{t+1} = 0$; else, $\gamma_{t+1} = \gamma_c$ for some constant $\gamma_c \in [0, 1]$.

2.2 Value Function

An important step to finding the best policy is to figure out the value of the current policy, a task which is called the policy evaluation or *prediction* task. For a given policy π, the value $v_\pi : \mathcal{S} \to \mathbb{R}$ is the expected return at state s when following policy π:

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

Here $\mathbb{E}_\pi[\cdot]$ describes the expected value of the random variable given the agent follows π in the MDP. Similarly, the action-value $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the expected return at state s when taking action a and then following π.

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

The action-value function for a given policy can be learned using algorithms like Expected Sarsa [36], which at every time step t, updates the estimate Q

at state S_t with action A_t using the Bellman equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma_{t+1} \sum_{a \in \mathcal{A}} \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.3}$$

where α is the step size. Expected Sarsa is an off-policy algorithm, where the *behaviour policy* (often denoted b) that takes actions in the environment can be different from the *target policy* π whose action value function is being estimated.

2.3 Q-Learning

Returning to the topic of finding the optimal policy, for any given MDP, we know that there exists at least one optimal policy π^* that has the optimal action-value function, in that for given any policy π, $q_\pi(s, a) \leq q_{\pi^*}(s, a)$, $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}$. While there might be multiple π^* where the above inequality is satisfied, all optimal policies share the same unique optimal action-value function q^*. Therefore if we have q^*, we can simply take the greedy policy with respect to q^* (such that $\pi(a|s) = 1$ if $a = \max_{a \in \mathcal{A}} q^*(s, a)$, otherwise $\pi(a|s) = 0$) to recover an optimal policy.

Q-Learning is an off-policy algorithm which directly learns q^*. At each time step, Q-Learning updates the action value estimate $Q(s, a)$ as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma_{t+1} \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.4}$$

Notice how Q-Learning is exactly Expected Sarsa if the target policy is set to be greedy with respect to the current estimated action-value function. Commonly, the behaviour policy is set to be ϵ-greedy with respect to the current action value estimate, where the policy is to take the greedy action generally, but have ϵ probability to take a random action.

2.4 Value Function Approximation

The algorithms we've described above all have implicitly assumed the use of *tabular representation*, where the estimate for each state or state-action pair

is tracked individually. In cases where the number of states or state-action pairs are too big to represent individually, we need to turn to approximate solutions.

Many basic RL algorithms have natural *semi-gradient* extensions[1] when function approximation is involved. For example, given that $\hat{q}(s, a; \boldsymbol{\theta})$ is the estimated action-value function parameterized by $\boldsymbol{\theta}$, the semi-gradient Expected Sarsa update is

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha[R_{t+1} + \gamma_{t+1} \sum_{a \in \mathcal{A}} \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)] \nabla_{\boldsymbol{\theta}} \hat{q}(S_t, A_t; \boldsymbol{\theta}) \quad (2.5)$$

And similarly, the semi-gradient Q-Learning update is

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha[R_{t+1} + \gamma_{t+1} \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \nabla_{\boldsymbol{\theta}} \hat{q}(S_t, A_t; \boldsymbol{\theta}) \quad (2.6)$$

These algorithms may not converge when using neural networks or under off-policy sampling. However, they have been shown to often work in practice when using neural networks given modifications described in Section 3.7.

2.5 Model-Based RL

The methods we have described in the previous section are all what is called *model-free* RL algorithms. They are called model-free because they operate directly from gathered experience $(S_t, A_t, R_{t+1}, S_{t+1}, \gamma_{t+1})$ to update the value estimate. In contrast, model-based RL algorithms also uses experience to learn a model, a computational object that answers predictive questions about the environment, and uses a computational process called *planning* to transfer knowledge from the model to improve the policy and value estimate.

The most common form of model is called a world model, which learns to simulate environment transitions. When this model is perfectly accurate, the agent can effectively simulate its own experience that mirrors real transitions without interacting with the environment.

There are several ways an agent can use such a model. One way is to use it to directly estimate the utility of different action choices at the current

[1] See [42] for more details about semi-gradient methods, and why they are often preferred over true gradient alternatives.

state by "rolling out" the model and predicting the agent's trajectory far into the future. These *decision-time planning* algorithms improves the quality of the action taken at the agent's current state by predicting the result of future trajectories using the model. Examples of such algorithms are model predictive control algorithms and MuZero [31], which uses Monte Carlo tree search online to perform these rollouts.

Another way to use such a model is to do *background planning*, which uses the simulated data to update the estimates of different state-action pairs in the background. The Dyna [41] architecture is a classic example of using both model-free learning and background planning to update its policy and value estimate. On each step, the agent simulates several transitions according to its model, and updates with those transitions as if they were real experience. Learning and using such a model is worthwhile in vast or ever-changing environments, where the agent learns over a long time period and can benefit from re-using knowledge about the environment. A simple illustrative example is Dyna-Q [41] which uses the standard Q-Learning update for both its model-free update and planning:

Algorithm 1 Dyna-Q

Initialize $Q(s, a)$ and model $\mathcal{M}(s, a)$
$s_0 \leftarrow$ current state
for $t \in 0, 1, 2, ...$ **do**
    Take action a_t using Q (e.g., ϵ-greedy), observe $s_{t+1}, r_{t+1}, \gamma_{t+1}$
    $\mathcal{M}(s_t, a_t) \leftarrow s_{t+1}, \gamma_{t+1}, r_{t+1}$
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma_{t+1} \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t)]$
    for n times **do**
        Sample s, a according to some algorithm or distribution
        $s', r, \gamma \leftarrow M(s, a)$
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s, a)]$

Many open problems with Dyna remain to make it more widely useful. These include (1) one-step models learned in Dyna can be difficult to use for long-horizon planning, (2) learning probabilities over outcome states can be complex, especially for high-dimensional states and (3) planning itself can be computationally expensive for large state spaces. The Goal-Space Planning

algorithm proposed in this thesis attempts to alleviate many of these issues.

2.6 Options and Temporal Abstraction

Options [37] are temporally extended courses of action that help the agent reason at a higher level of temporal abstraction. They often represent skills such as navigating to a specific room or opening the door, describing how to accomplish subtasks an agent might want to perform to achieve its overall goal. More formally, an option o is defined by the tuple (d, γ, π). $d : \mathcal{S} \to \{0, 1\}$ is the initiation function that describes whether the option can be initiated from that state ($d(s) = 1$) or not ($d(s) = 0$). $\gamma : \mathcal{S} \to [0, 1]$ describes the option termination probability at state s, and $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the policy that describes how the agent should behave when performing the option.

A simple way to use options is to expand the agent's action space with options and learn an action-value function $Q : \mathcal{S} \times (\mathcal{A} \cup \mathcal{O}) \to \mathbb{R}$. The agent decides if it could perform option o according to $d_o(s)$. If the agent can and decides to perform option o based on $Q(s, o)$, it would follow π_o until the option terminates as determined by γ_o. When options are executed in this manner, standard Q-Learning transfer seamlessly into this new action space, and these *temporally abstract* actions allow the agent to "jump" multiple steps when performing credit assignment, speeding up learning by reducing the horizon of the problem.

2.7 Goal-Conditioned RL

Goals often serve as a foundation for higher-level abstraction when planning for humans. We think of plans in terms of transitions between goals, and when new information becomes available, we often form new goals that we include when planning. These goals allow us to reason efficiently over long horizons and allow us to adapt our behaviour quickly to account for goals we might have in the far future.

There is a large and growing literature on goal-conditioned RL (GCRL), which takes the concept of goals and applies it to RL. This is a problem setting

where the aim is to learn a policy $\pi(a|s, g)$, or action-value function $Q(s, a|g)$ that can be (zero-shot) conditioned to quickly reach different possible goals. The agent learns for a given set of goals, with the assumption that at the start of each episode the goal state is explicitly given to the agent and the agent's task is to reach that specific goal. After this training phase, the policy should generalize to previously unseen goals. Naturally, this idea has particularly been applied to navigation, having the agent learn to navigate to different states (goals) in the environment.

Subgoals, also called landmark states, have also been used to improve planning for GCRL [16]. Planning is done between landmarks, using graph-based search, to find the shortest path to the goal that paths through landmark states. The policy is set to reach the nearest goal (using action-values with cost-to-goal rewards of -1 per step) and learned distance functions between states and goals and between goals.

The idea of learning models that immediately apply to new subtasks, using successor features, is like GCRL but goes beyond navigation. The option keyboard involves encoding options (or policies) as vectors that describe the corresponding (pseudo) reward [3]. This work has been expanded more recently, using successor features [4]. New policies can then be easily obtained for new reward functions, by linearly combining the (basis) vectors for the already learned options. No planning is involved in this work, beyond a one-step decision-time choice amongst options.

This setting bears a strong resemblance to what we do in this work, but is notably different. Our models can be seen as goal-conditioned models—part of the solution—for planning in the general RL setting. GCRL, on the other hand, is a problem setting. Many approaches do not consider planning, but instead focus on effectively learning the goal-conditioned value functions or policies.

Chapter 3

Goal-Space Planning

Goal-Space Planning is an algorithm that incorporates the intuitive idea of planning with subgoals to tackle the computational challenges of Dyna, to (1) reduce accumulation of errors in long-horizon planning through using temporally abstract models, (2) avoid learning probabilities over outcome states, and (3) make planning more efficient for large state spaces by planning at a more abstract level.

The idea is that the agent has knowledge of a set of subgoals, and learns a corresponding set of *subgoal-conditioned models*, a minimal model focused around planning utility that answers how good the path to each subgoal is. These models allow the agent to quickly find the value of reaching different subgoals by planning in a temporally abstract MDP formed with subgoals as states, and options to reach each subgoal as actions. Finally, we can update the behaviour policy based on these subgoal values to speed up learning. Figure 3.1 provides a visual overview of this process.

These subgoal-conditioned models have several important properties that make them easier to learn than one-step world models. The first is that these models do not need to predict the next state feature, thus bypassing the possible issue of predicting irrelevant features. The second is that these models only need to be learned within a local region where that subgoal is relevant and reachable, further focusing function approximation resources.

First, we will provide an illustrative example of GSP for policy evaluation to highlight the computational benefits of GSP in Section 3.1. We formalize

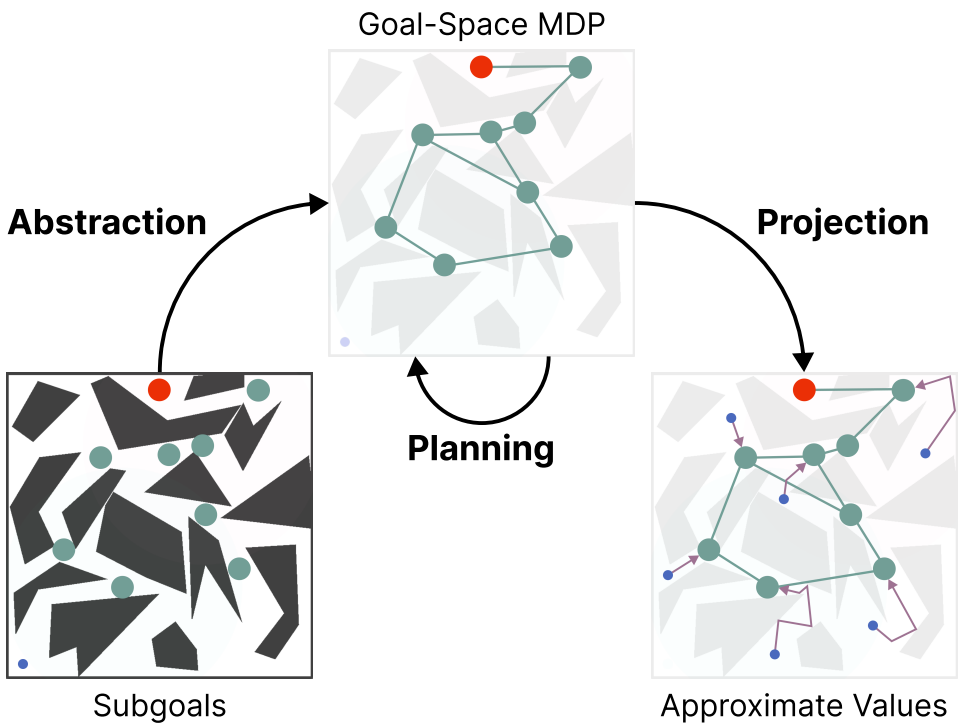

Figure 3.1: Visual overview of Goal-Space Planning in an example 2D environment (refer to Section 4.1 for details about the shown environment). The agent begins with a set of subgoals (denoted in teal) and learns a set of subgoal-conditioned models. **(Abstraction)** Using these models, the agent forms an abstract goal-space MDP where the states are subgoals with options to reach each subgoal as actions. **(Planning)** The agent then 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.

the idea of subgoals, subgoal-conditioned models, and describe the basic algorithms to learn these models in 3.2, 3.3, and 3.4. In Section 3.5, we describe how to use these models to plan quickly and generate values of subgoals, and how these values are used to improve the base policy. Then, we summarize the full Goal-Space Planning algorithm in Section 3.6. Finally, we extend GSP to the deep RL setting in Section 3.7.

3.1 GSP for Policy Evaluation

To further illustrate the ideas of GSP and its computational benefits, let us start with a simple version of GSP for a policy evaluation problem where the goal is to learn v^π for a fixed deterministic policy π in a deterministic environment, assuming access to perfect models.

The key idea is to propagate values quickly across the space by updating between a subset of states that we call *subgoals*, $g \in \mathcal{G} \subset \mathcal{S}$. (Later we extend $\mathcal{G} \not\subset \mathcal{S}$ to abstract subgoal vectors that need not correspond to any state.) To do so, we need temporally extended models between pairs g, g' that may be further than one-transition apart. For policy evaluation, these models are the accumulated rewards $r_{\pi,\gamma} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ and discounted probabilities $P_{\pi,\gamma} : \mathcal{S} \times \mathcal{S} \to [0, 1]$ of transitioning between goals under π:

$$r_{\pi,\gamma}(g, g') \stackrel{\text{def}}{=} \mathbb{E}_\pi[R_{t+1} + \gamma_{g',t+1} r_{\pi,\gamma}(S_{t+1}, g') | S_t = g]$$
$$P_{\pi,\gamma}(g, g') \stackrel{\text{def}}{=} \mathbb{E}_\pi[1(S_{t+1} = g')\gamma_{t+1} + \gamma_{g',t+1} P_{\pi,\gamma}(S_{t+1}, g') | S_t = g]$$

where $\gamma_{g',t+1} = 0$ if $S_{t+1} = g'$ and otherwise equals γ_{t+1}, the environment discount. In an undiscounted environment, because both the environment and the policy are deterministic, $P_{\pi,\gamma}(g, g')$ would either be 1 or 0, indicating whether the agent would or would not reach g' from g under π, respectively. With discounting, if the agent would reach g' from g under π, then $P_{\pi,\gamma}(g, g')$ would be the multiplicative accumulation of discounts between g and g' (i.e. if the discount equals a constant γ and it takes k steps under π to reach g' from g, then $P_{\pi,\gamma}(g, g') = \gamma^k$). If the agent would not reach g' from g under π, then $P_{\pi,\gamma}(g, g')$ will simply be zero. We can treat $\mathcal{G}$ as our new state space and plan in this space, to get value estimates v for all $g \in \mathcal{G}$

$$v(g) = r_{\pi,\gamma}(g, g') + P_{\pi,\gamma}(g, g')v(g') \quad \text{where } g' = \text{argmax}_{g' \in \bar{\mathcal{G}}} P_{\pi,\gamma}(g, g')$$

where $\bar{\mathcal{G}} = \mathcal{G} \cup \{s_{\text{terminal}}\}$ if there is a terminal state (episodic problems) and otherwise $\bar{\mathcal{G}} = \mathcal{G}$.[1] It is straightforward to show this converges, because $P_{\pi,\gamma}$ is a substochastic matrix (see Section 3.1.1).

[1] Remember that because the environment is deterministic, $\text{argmax}_{g' \in \bar{\mathcal{G}}} P_{\pi,\gamma}(g, g')$ gives us the closest g' to g.

Once we have these values, we can propagate these to other states, locally, again using the closest g to s. We can do so by noticing that the above definitions can be easily extended to $r_{\pi,\gamma}(s, g')$ and $P_{\pi,\gamma}(s, g')$, since for a pair (s, g) they are about starting in the state s and reaching g under π.

$$v(s) = r_{\pi,\gamma}(s, g) + P_{\pi,\gamma}(s, g)v(g) \quad \text{where } g = \text{argmax}_{g \in \tilde{\mathcal{G}}} P_{\pi,\gamma}(s, g). \tag{3.1}$$

Because the rhs of this equation is fixed, we only cycle through these states once to get their values.

All of this might seem like a lot of work for policy evaluation; indeed, it will be more useful to have this formalism for control. But, even here Goal-Space Planning can be beneficial. Let us assume the environment is a chain $s_1, s_2, \ldots, s_n$, where $n = 1000$ and $\mathcal{G} = \{s_{100}, s_{200}, \ldots, s_{1000}\}$. GSP tackles this problem by first planning over a much smaller number of subgoals $g \in \mathcal{G}$, then learning the values of other states based on these subgoals. Finding the value of subgoals is much easier as it converts a 1000 step horizon problem into a 10 step one. Once the agent have the values of subgoals, we can update all the states in one sweep, using Equation (3.1), to set the value of s to be the expected return to reach g plus the value of g. As a result, changes in the environment also propagate faster. If the reward at s' changes, locally the reward model around s' can be updated quickly, to change $r_{\pi,\gamma}(g, g')$ for pairs g, g' where s' is along the way from g to g'. This local change quickly updates the values back to earlier $\tilde{g} \in \mathcal{G}$.

We can also plan efficiently by updating the value at the end in s_n, and then updating states backwards from the end. But, without knowing this structure, it is not a general purpose strategy. For general MDPs, we would need smart ways to do search control: the approach to pick states for one-step updates. GSP effectively reduces the number of states we are required to sweep over by constraining them to subgoals, then afterwards, backing up values from these subgoals to individual states. In fact, we can leverage search control strategies to improve the Goal-Space Planning step. Then we get the benefit of these approaches, as well as the benefit of planning over a much smaller state space.

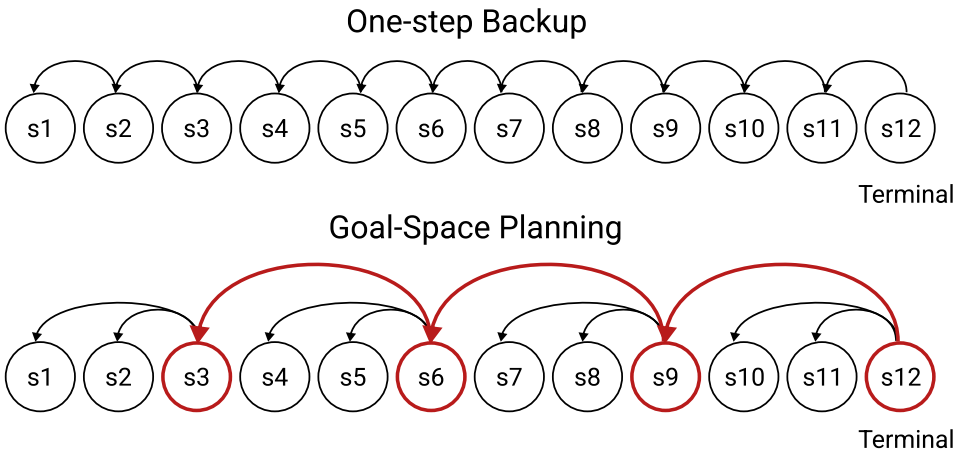

Figure 3.2: Comparing one-step backup with Goal-Space Planning. GSP first focuses planning over a smaller set of subgoals (in red), then updates the values of individual states.

3.1.1 Proofs for the Deterministic Policy Evaluation

We provide the proofs for deterministic policy evaluation here. We assume throughout that the environment discount γ_{t+1} is a constant $\gamma_c \in [0, 1)$ for every step in an episode, until termination when it is zero. The below results can be extended to the case where $\gamma_c = 1$, using the standard strategy for the stochastic shortest path problem setting.

First, we want to show that given $r_{\pi,\gamma}$ and $P_{\pi,\gamma}$, we can guarantee that the update for the values for $\mathcal{G}$ will converge. Recall that $\bar{\mathcal{G}} = \mathcal{G} \cup \{s_{\text{terminal}}\}$ is the augmented goal space that includes the terminal state. This terminal state is not a subgoal—since it is not a real state—but is key for appropriate planning.

Lemma 1. *Assume that we have a deterministic MDP, deterministic policy π, $\gamma_c < 1$, a discrete set of subgoals $\mathcal{G} \subset \mathcal{S}$, and that we iteratively update $v_t \in \mathbb{R}^{|\bar{\mathcal{G}}|}$ with the dynamic programming update*

$$v_t(g) = r_{\pi,\gamma}(g, g') + P_{\pi,\gamma}(g, g')v_{t-1}(g') \quad \text{where } g' = \operatorname*{argmax}_{g' \in \bar{\mathcal{G}}} P_{\pi,\gamma}(g, g') \quad (3.2)$$

for all $g \in \mathcal{G}$, starting from an arbitrary (finite) initialization $v_0 \in \mathbb{R}^{|\bar{\mathcal{G}}|}$, with $v_t(s_{terminal})$ fixed at zero. Then then v_t converges to a fixed point.

Proof. To analyze this as a matrix update, we need to extend $P_{\pi,\gamma}(g, g')$ to include an additional row transitioning from s_{terminal}. This row is all zeros,

because the value in the terminal state is always fixed at zero. Note that there are ways to avoid introducing terminal states, using transition-based discounting [51], but for this work it is actually simpler to explicitly reason about them and reaching them from subgoals.

To show this we simply need to ensure that $P_{\pi,\gamma}$ is a substochastic matrix. Recall that

$$P_{\pi,\gamma}(g, g') \stackrel{\text{def}}{=} \mathbb{E}_\pi[1(S_{t+1} = g')\gamma_{t+1} + \gamma_{g',t+1} P_{\pi,\gamma}(S_{t+1}, g') | S_t = g]$$

where $\gamma_{g',t+1} = 0$ if $S_{t+1} = g'$ and otherwise equals γ_{t+1}, the environment discount. If it is substochastic, then $\|P_{\pi,\gamma}\|_2 < 1$. Consequently, the Bellman operator

$$(Bv)(g) = r_{\pi,\gamma}(g, g') + P_{\pi,\gamma}(g, g')v(g') \quad \text{where } g' = \operatorname*{argmax}_{g' \in \bar{\mathcal{G}}} P_{\pi,\gamma}(g, g')$$

is a contraction, because $\|Bv_1 - Bv_2\|_2 = \|P_{\pi,\gamma}v_1 - P_{\pi,\gamma}v_2\|_2 \leq \|P_{\pi,\gamma}\|_2 \|v_1 - v_2\|_2 < \|v_1 - v_2\|_2$.

Because $\gamma_c < 1$, then either g immediately terminates in g', giving $1(S_{t+1} = g')\gamma_{t+1} + \gamma_{g',t+1} P_{\pi,\gamma}(S_{t+1}, g') = \gamma_{t+1} + 0 \leq \gamma_c$. Or, it does not immediately terminate, and $1(S_{t+1} = g')\gamma_{t+1} + \gamma_{g',t+1} P_{\pi,\gamma}(S_{t+1}, g') = 0 + \gamma_c P_{\pi,\gamma}(S_{t+1}, g') \leq \gamma_c$ because $P_{\pi,\gamma}(S_{t+1}, g') \leq 1$. Therefore, if $\gamma_c < 1$, then $\|P_{\pi,\gamma}\|_2 \leq \gamma_c$.

□

Proposition 1. *For a deterministic MDP, deterministic policy π, and a discrete set of subgoals $\mathcal{G} \subset \mathcal{S}$ that are all reached by π in the MDP, given the $\tilde{v}(g)$ obtained from Equation 3.2, if we set*

$$v(s) = r_\gamma(s, g) + P_{\pi,\gamma}(s, g)\tilde{v}(g) \quad \textit{where } g = \operatorname*{argmax}_{g \in \bar{\mathcal{G}}} P_{\pi,\gamma}(s, g) \tag{3.3}$$

for all states $s \in \mathcal{S}$ then we get that $v = v_\pi$.

Proof. For a deterministic environment and deterministic policy this result is straightforward. The term $P_{\pi,\gamma}(s, g) > 0$ only if g is on the trajectory from s when the policy π is executed. The term $r_\gamma(s, g)$ consists of deterministic (discounted) rewards and $\tilde{v}(g)$ is the true value from g, as shown in Lemma 3.2 (namely $\tilde{v}(g) = v_\pi(g)$). The subgoal g is the closest subgoal on the trajectory from s, and $P_{\pi,\gamma}(s, g)$ is γ_c^t where t is the number of steps from s to g. □

3.2 Defining Subgoals

Intuitively, subgoals represents abstract intermediate objectives that the agent can achieve. These can be at different levels of abstractions and cover many different modes. It can be anything from the amount of sleep one needs, getting to a pizza place, or the temperature outside.

We formalize each subgoal as a subgoal description vector that is within the space of possible subgoal descriptions. Each subgoal vector could be a one-hot vector encoding indicating the desired subgoal, and more generally can be a vector of features describing the subgoal. This vector space need not correspond directly to the (possibly continuous) state space, as subgoals can be a further abstraction on states.

In this work, we assume that we have a finite set of subgoals, in the form of a finite set of subgoal description vectors $\mathcal{G}$. For example, one subgoal for a robot might be 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.

To fully specify a subgoal, we need a *membership function* m that indicates if a state s is a member of subgoal g: $m(s, g) = 1$, and zero otherwise. Many states can be mapped to the same subgoal g. For the above example, if the first two elements of the state vector s consist of the front and side distance sensor, $m(s, g) = 1$ for any states where s_1, s_2 are less than some threshold ϵ.

Finally, we only reason about reaching subgoals from a subset of states, called *initiation sets* for options [37]. This constraint is key for locality, to learn and reason about a subset of states for a subgoal. We assume the existence of a *initiation function* $d(s, g)$ that is 1 if s is in the initiation set for g (e.g., sufficiently close in terms of reachability) and zero otherwise. From this initiation set, the agent needs an option policy $\pi_g : \mathcal{S} \times \mathcal{A} \to [0, 1]$ for subgoal g that starts from any s in the initiation set, and terminates in g—in $\tilde{s}$ where $m(\tilde{s}, g) = 1$.

Here, we assume that subgoals and their membership function are the product of a subgoal discovery algorithm. We discuss some approaches to learning the initiation function and option policy in Section 3.4

3.3 Defining Subgoal-Conditioned Models

Goal-Space Planning involves planning and acting at two different levels. One involves the goal-space MDP, formed by subgoals and options to reach each subgoal. Another involves the state-space MDP, the one that the behaviour policy operates in. This requires two different "levels" of models, state-to-subgoal models that answers questions about reaching a subgoal g from state s, and subgoal-to-subgoal models that answers questions about reaching subgoal g' from some subgoal g.

The state-to-subgoal models are $r_\gamma : \mathcal{S} \times \bar{\mathcal{G}} \to \mathbb{R}$ and $\Gamma : \mathcal{S} \times \bar{\mathcal{G}} \to [0, 1]$. The reward-model $r_\gamma(s, g)$ is the discounted rewards under option policy π_g:

$$r_\gamma(s, g) = \mathbb{E}_{\pi_g}[R_{t+1} + \gamma_g(S_{t+1}) r_\gamma(S_{t+1}, g) | S_t = s]$$

where the discount is zero upon reaching subgoal g

$$\gamma_g(S_{t+1}) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } m(S_{t+1}, g) = 1, \text{ if subgoal } g \text{ is achieved by being in } S_{t+1} \\ \gamma_{t+1} & \text{else} \end{cases}$$

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

$$\Gamma(s, g) = \mathbb{E}_{\pi_g}[m(S_{t+1}, g)\gamma_{t+1} + \gamma_g(S_{t+1})\Gamma(S_{t+1}, g) | S_t = s].$$

These state-to-subgoal models will only be queried for (s, g) where $d(s, g) > 0$: they are local models.

To define subgoal-to-subgoal models,[2] $\tilde{r}_\gamma : \mathcal{G} \times \bar{\mathcal{G}} \to \mathbb{R}$ and $\tilde{\Gamma} : \mathcal{G} \times \bar{\mathcal{G}} \to [0, 1]$, we use the state-to-subgoal models. For each subgoal $g \in \mathcal{G}$, we aggregate $r_\gamma(s, g')$ and $\Gamma(s, g')$ for all s where $m(s, g) = 1$.

$$\tilde{r}_\gamma(g, g') \stackrel{\text{def}}{=} \frac{1}{z(g)} \sum_{s:m(s,g)=1} r_\gamma(s, g') \quad \text{and} \quad \tilde{\Gamma}(g, g') \stackrel{\text{def}}{=} \frac{1}{z(g)} \sum_{s:m(s,g)=1} \Gamma(s, g') \tag{3.4}$$

for normalizer $z(g) \stackrel{\text{def}}{=} \sum_{s:m(s,g)=1} m(s, g)$. This definition assumes a uniform weighting over the states s where $m(s, g) = 1$. We could allow a non-uniform

[2]The first input is any $g \in \mathcal{G}$, the second is $g' \in \bar{\mathcal{G}}$, which includes s_{terminal}. We need to reason about reaching any subgoal or s_{terminal}. But s_{terminal} is not a real state: we do not reason about starting from it to reach subgoals.

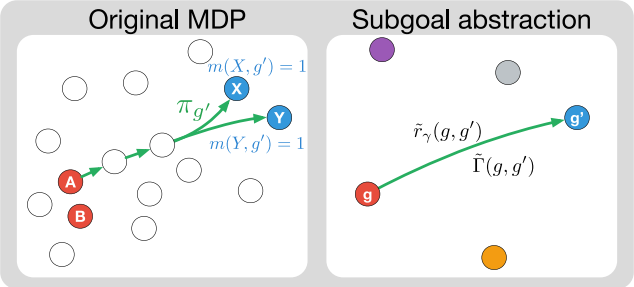

Figure 3.3: Illustration of the original and abstract goal space. The colored circles on the right represents different subgoals.

weighting, potentially based on visitation frequency in the environment. For this work, however, we assume the subgoals are defined such that all states s where $m(s, g) = 1$ have similar $r_\gamma(s, g')$ and $\Gamma(s, g')$, making a uniform weighting reasonable.

These models are also local models, as we can similarly extract $\tilde{d}(g, g')$ from $d(s, g')$ and only reason about g' nearby or relevant to g. We set $\tilde{d}(g, g') = \max_{s \in \mathcal{S}: m(s,g)>0} d(s, g')$, indicating that if there is a state s that is in the initiation set for g' and has membership in g, then g' is also relevant to g.

Let us consider an example, in Figure 3.3. The red states are members of g ($m(A, g) = 1$, $m(B, g) = 1$) and the blue members of g' ($m(X, g') = 1$, $m(Y, g') = 1$). For all s in the diagram, $d(s, g') > 0$ (all are in the initiation set): the policy $\pi_{g'}$ can be queried from any s to get to g'. The green path in the left indicates the trajectory under $\pi_{g'}$ from A, stochastically reaching either X or Y, with accumulated reward $r_\gamma(A, g')$ and discount $\Gamma(A, g')$ (averaged over reaching X and Y). The subgoal-to-subgoal models, on the right, indicate g' can be reached from g, with $\tilde{r}_\gamma(g, g')$ averaged over both $r_\gamma(A, g')$ and $r_\gamma(B, g')$ and $\tilde{\Gamma}(g, g')$ over $\Gamma(A, g')$ and $\Gamma(B, g')$, as described in Equation (3.4).

3.4 Learning Subgoal-Conditioned Models

We have defined subgoal-conditioned models in the previous section, now we need methods to learn them. We first introduce general value functions [38], as we leverage this idea to apply standard off-policy RL algorithms to learn state-to-subgoal models and option policies. Then, we discuss how to learn

state-to-subgoal models $r_\gamma(s, g)$ and $\Gamma(s, g)$ in Section 3.4.2 and how to learn subgoal-to-subgoal models $\tilde{r}_\gamma(s, g)$ and $\tilde{\Gamma}(s, g)$ in Section 3.4.3. Finally, we describe a general algorithm for learning option policies in Section 3.4.4.

3.4.1 General Value Functions

Just as value functions capture the expected return under a policy, it is possible to capture the expected accumulation of any other quantity. General value functions (GVFs) [38] formalize this concept and measure the expected accumulation of a signal (the cumulant) under some policy. Each GVF is defined by a tuple of 3 functions (C, γ, π). $C : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the cumulant function, the signal that is accumulated and which generalizes the reward function. $\gamma : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a transition-based discount function that is similar to the environment's discount function. π is the policy under which the cumulant is accumulated.

Because of the similarity between value functions and general value functions, standard RL algorithms like Expected Sarsa can be used to learn GVFs by replacing the reward, discount, and policy with the GVF's C, γ, and π.

Estimating the general value function when given a specific policy can be seen as equivalent to the prediction problem in standard RL. There is also the idea of a control GVF, where the goal is to learn a policy that maximizes the expected cumulative cumulant given C and γ, just as the goal of a control problem in standard RL is to learn a policy that maximizes the expected return. Similarly, we can use Q-learning to learn this policy.

The generalization from value functions to GVFs extends the utility of standard off-policy RL algorithms to learn any prediction and control question that can be formulated as a prediction or maximization of the future accumulation of a Markov quantity with discount, like our formulation of r_γ and Γ. The exact details about how we perform off-policy learning to learn these models and option policies are described in the sections below.

3.4.2 Learning State-to-Subgoal Models

We start by assuming that we have π_g and discuss learning it after understanding learning these models. Note that the algorithms discussed here are basic off-policy algorithms that might diverge in the function approximation setting, but there exists practical techniques to stabilize learning in deep RL that are discussed in Section 3.7. To improve model learning, we can leverage the large literature on GVFs [38] and UVFAs [30]. There are also nuances involved in (1) restricting updating to relevant states according to $d(s, g)$ and (2) considering ways to jointly learn d and Γ that we discuss below.

The data is 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, but in our experiments we use the action-value variant since it avoids importance sampling and the need to have the distribution over actions under behavior b.

Model Update using Importance Sampling We can update $r_\gamma(\cdot, g)$ with an importance-sampled temporal difference (TD) learning update $\rho_t \delta_t^r \nabla r_\gamma(S_t, g)$ 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 $\rho_t \delta_t^\Gamma \nabla \Gamma(S_t, g)$ 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.

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 $\delta_t^r \nabla r_\gamma(S_t, A_t, g)$. For Γ we have

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

and using update $\delta_t^\Gamma \nabla \Gamma(S_t, A_t, g)$. We then define $r_\gamma(s, g) \stackrel{\text{def}}{=} r_\gamma(s, \pi_g(s), g)$ and $\Gamma(s, g) \stackrel{\text{def}}{=} \Gamma(s, \pi_g(s), g)$ 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 [39], with connections made between interest and initiation sets [51]. 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 $r_\gamma(s, g)$ and $\Gamma(s, g)$ in states s where $d(s, g) = 0$. 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 $\pi_g(s) = a$.

One issue with limiting model updates only to relevant states is if the state distribution for following the option policy, starting from some state s where $d(s, g) = 1$, is non-zero at some other state s' where $d(s', g) = 0$. If that is the case, then the model will not converge to the correct values as the update equation assumes that the next state s' will also be updated. This can be resolved with *emphatic weightings* [39] that allows us to use 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. In our experiments, we found it sufficient to alleviate this issue by encouraging the learned option policy to stay within the relevant states by setting the bootstrap value of states where $d(s, g) = 0$ to the minimum possible value for the reward function used to learn the option policy, such that a well-learned option policy should always stay within relevant states where the bootstrap target will be updated.

Learning the relevance model d We assume in this work that we simply have $d(s, g)$, but we can at least consider ways to learn it. One approach is to learn and use Γ to determine which states 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 [18], where both options and such a relevance function are learned jointly. For us, they could also be learned jointly, where a larger set of states start with $d(s, g) = 1$, then if $\Gamma(s, g)$ remains small, these may be switched to $d(s, g) = 0$ and they will stop being learned in the model updates.

3.4.3 Learning Subgoal-to-Subgoal Models

Finally, we need to extract the subgoal-to-subgoal models $\tilde{r}_\gamma, \tilde{\Gamma}$ from r_γ, Γ. 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 and 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.

3.4.4 A General Algorithm for Learning Option Policies

Finally, we need to learn the option policies π_g. In the simplest case, it is enough to learn π_g that makes $r_\gamma(s, g)$ maximal for every relevant s (i.e., $d(s, g) > 0$). We can learn the action-value variant $r_\gamma(s, a, g)$ using a Q-learning 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.[3] In our experiments, this strategy is sufficient for learning π_g and $r_\gamma(s, g)$.

More generally, however, this approach may be ineffective because maximizing environment reward may be at odds with reaching the subgoal in a reasonable number of steps (or at all). For example, in environments where the reward is always positive, maximizing environment reward might encourage the option policy not to terminate.[4] However, we do want π_g to reach g, while also obtaining the best return along the way to g. For example, if there is a lava pit along the way to a goal, even if going through the lava pit is the shortest path, we want the learned option to get to the goal by going around the lava pit. We therefore want to be reward-respecting, as introduced for reward-respecting subtasks [40], but also ensure termination.

We can consider a spectrum of option policies that range from the policy that reaches the goal as fast as possible to one that focuses on environment reward. We can specify a new reward for learning the option: $\tilde{R}_{t+1} = cR_{t+1} +$

[3]The additional maximization over $r_\gamma(s, a, g)$ to obtain the option policy can reduce the quality of the estimate, as r_γ is estimating the value for the policy induced by the previous estimate instead of the current r_γ. While this will not be different when the induced option policy has converged, we cannot guarantee this generally. However, we find that this way of finding the option policy was sufficient for our experiments.

[4]It is not always the case that positive rewards result in option policies that do not terminate. If there is a large, positive reward at the subgoal in the environment, Even if all rewards are positive, if $\gamma_c < 1$ and there is larger positive reward at the subgoal than in other nearby states, then the return is higher when reaching this subgoal sooner, since that reward is not discounted as many steps. This outcome is less nuanced for negative reward. If the rewards are always negative, on the other hand, then the option policy will terminate, trying to find the path with the best (but still negative) return.

$(1-c)(-1)$. When $c=0$, we have a cost-to-goal problem, where the learned option policy should find the shortest path to the goal, regardless of reward along the way. When $c=1$, the option policy focuses on environment reward, but may not terminate in g. We can start by learning the option policy that takes the shortest path with $c=0$, and the corresponding $r_\gamma(s,g), \Gamma(s,g)$. The constant c can be increased until π_g stops going to the goal, or until the discounted probability $\Gamma(s,g)$ drops below a specified threshold.

Even without a well-specified c, the values under the option policy can still be informative. For example, it might indicate that it is difficult or dangerous to attempt to reach a goal. For this work, we propose a simple default, where we fix $c=0.5$. Adaptive approaches, such as the idea described above, are left to future work.

The resulting algorithm to learn π_g involves learning a separate value function for these rewards. We can learn action-values (or a parameterized policy) using the above reward. For example, we can learn a policy with the Q-learning update to action-values $\tilde{q}$

$$\Big(cR_{t+1} + c - 1 + \gamma_{g,t+1} \max_{a'} \tilde{q}(S_{t+1}, a', g) - \tilde{q}(S_t, A_t, g)\Big) \nabla \tilde{q}(S_t, A_t, g)$$

Then we can set π_g to be the greedy policy, $\pi_g(s) = \text{argmax}_{a \in \mathcal{A}} \tilde{q}(s,a,g)$.

3.5 GSP with Subgoal-Conditioned Models

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

$$\tilde{v}(g) = \max_{g' \in \bar{\mathcal{G}}: \tilde{d}(g,g') > 0} \tilde{r}_\gamma(g,g') + \tilde{\Gamma}(g,g')\tilde{v}(g') \quad \text{(Background Planning)} \tag{3.5}$$

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

We can interpret this update as a standard value iteration update in a new MDP, where (1) the set of states is $\mathcal{G}$, (2) 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 $\tilde{d}(g, g') > 0$ and (3) the rewards are $\tilde{r}_\gamma$ and the discounted transition probabilities are $\tilde{\Gamma}$. Under this correspondence, it is straightforward to show that the above converges to the optimal values in this new goal-space MDP, shown in Proposition 2 in Section 3.5.1.

This goal-space planning approach does not suffer from typical issues with model-based RL. First, the model is not iterated, but we still obtain temporal abstraction because the model itself incorporates it. Second, we do not need to predict entire state vectors—or distributions over them—because we instead input the outcome g' into the function approximator. This may feel like a false success as it potentially requires restricting ourselves to a smaller number of subgoals. If we want to use a larger number of subgoals, then we may need a function to generate these subgoal vectors anyway—bringing us back to the problem of generating vectors. However, this is likely easier as (1) the subgoals themselves can be much smaller and more abstract, making it more feasibly to procedurally generate them and (2) it may be more feasible to maintain a large set of subgoal vectors, or generate individual subgoal vectors, than producing relevant subgoal vectors from a given subgoal.

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.

$$v_{\text{sub}}(s) \stackrel{\text{def}}{=} \max_{g \in \bar{\mathcal{G}}: d(s,g) > 0} r_\gamma(s, g) + \Gamma(s, g)\tilde{v}(g) \quad \text{(Projection Step)} \tag{3.6}$$

and take action a that corresponds to the action given by π_g for this maximizing g. However, this approach has two issues. First, restricting to going through subgoals might result in suboptimal policies. From a given state s, the set of relevant subgoals g may not be on the optimal path. Second, the learned models themselves may have inaccuracies, or planning may not have been

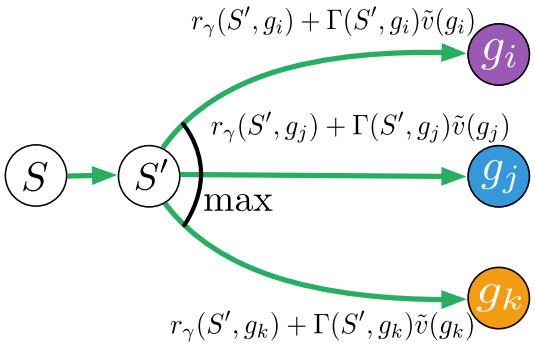

Figure 3.4: Computing $v_{\text{sub}}(S')$ to update the policy at S.

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

We instead propose to use $v_{\text{sub}}(s)$ within the bootstrap target for the action-values for the main policy. For a given transition $(S_t, A_t, R_{t+1}, S_{t+1})$, either as the most recent experience or from a replay buffer, the proposed *subgoal-value bootstrapping* update to parameterized $q(S_t, A_t; \mathbf{w})$ uses TD error

$$\delta \stackrel{\text{def}}{=} R_{t+1} + \gamma_{t+1}\Big((1-\beta)\underbrace{\max_{a'} q(S_{t+1}, a'; \mathbf{w})}_{\text{Standard bootstrap target}} + \beta \underbrace{v_{\text{sub}}(S_{t+1})}_{\text{Subgoal value}}\Big) - q(S_t, A_t; \mathbf{w}) \tag{3.7}$$

for some $\beta \in [0, 1]$. For $\beta = 0$, we get the standard Q-learning update. For $\beta = 1$, we fully bootstrap off the value provided by $v_{\text{sub}}(S_{t+1})$. This may result in suboptimal values $q(S_t, A_t; \mathbf{w})$, but should learn faster because a reasonable estimate of value has been propagated back quickly using goal-space planning. On the other hand, $\beta = 0$ is not biased by a potentially suboptimal $\tilde{v}(g)$, but does not take advantage of this fast propagation. An interim β can allow for fast propagation, but also help overcome suboptimality in the values.

We can show that the above update improves the convergence rate. This result is intuitive: subgoal-value bootstrapping changes the discount rate to $\gamma_{t+1}(1-\beta)$. In the extreme case of $\beta = 1$, we are moving our estimate towards $R_{t+1} + \gamma_{t+1} v_{\text{sub}}(S_{t+1})$ for v_{sub} not based on q without any bootstrapping: it is effectively a regression problem. We prove this intuitive result in Section 3.5.1. One other benefit of this approach is that the initiation sets need not cover the whole space: we can have a state $d(s, g) = 0$ for all g. If this occurs, we simply do not use v_{sub} and bootstrap as usual.

3.5.1 Proofs for the General Control Setting

In this section we assume that $\gamma_c < 1$, to avoid some of the additional issues for handling proper policies. The same strategies apply to the stochastic shortest path setting with $\gamma_c = 1$, with additional assumptions.

Proposition 2 (Convergence of Value Iteration in Goal-Space)**.** *Assuming that $\tilde{\Gamma}$ is a substochastic matrix, with $v_0 \in \mathbb{R}^{|\bar{\mathcal{G}}|}$ initialized to an arbitrary value and fixing $v_t(s_{terminal}) = 0$ for all t, then iteratively sweeping through all $g \in \mathcal{G}$ with update*

$$v_t(g) = \max_{g' \in \bar{\mathcal{G}}: \tilde{d}(g,g') > 0} \tilde{r}_\gamma(g, g') + \tilde{\Gamma}(g, g') v_{t-1}(g') \tag{3.8}$$

convergences to a fixed-point.

Proof. We can use the same approach typically used for value iteration. For any $v_0 \in \mathbb{R}^{|\bar{\mathcal{G}}|}$, we can define the operator

$$(B^g v)(g) \stackrel{\text{def}}{=} \max_{g' \in \bar{\mathcal{G}}: \tilde{d}(g,g') > 0} \tilde{r}_\gamma(g, g') + \tilde{\Gamma}(g, g') v(g')$$

First we can show that B^g is a γ_c-contraction. Assume we are given any two vectors v_1, v_2. Notice that $\tilde{\Gamma}(g, g') \leq \gamma_c$, because for our problem setting the discount is either equal to γ_c or equal to zero at termination. Then we have that for any $g \in \bar{\mathcal{G}}$

$$\begin{aligned}
&|(B^g v_1)(g) - (B^g v_2)(g)| \\
&= \left| \max_{g' \in \bar{\mathcal{G}}: \tilde{d}(g,g') > 0} \tilde{r}_\gamma(g, g') + \tilde{\Gamma}(g, g') v_1(g') - \max_{g' \in \bar{\mathcal{G}}: \tilde{d}(g,g') > 0} \tilde{r}_\gamma(g, g') + \tilde{\Gamma}(g, g') v_2(g') \right| \\
&\leq \max_{g' \in \bar{\mathcal{G}}: \tilde{d}(g,g') > 0} |\tilde{r}_\gamma(g, g') + \tilde{\Gamma}(g, g') v_1(g') - (\tilde{r}_\gamma(g, g') + \tilde{\Gamma}(g, g') v_2(g'))| \\
&= \max_{g' \in \bar{\mathcal{G}}: \tilde{d}(g,g') > 0} |\tilde{\Gamma}(g, g')(v_1(g') - v_2(g'))| \\
&\leq \max_{g' \in \bar{\mathcal{G}}: \tilde{d}(g,g') > 0} \gamma_c |v_1(g') - v_2(g')| \\
&\leq \gamma_c \|v_1 - v_2\|_\infty
\end{aligned}$$

Since this is true for any g, it is true for the max over g, giving

$$\|B^g v_1 - B^g v_2\|_\infty \leq \gamma_c \|v_1 - v_2\|_\infty.$$

Because the operator B^g is a contraction, since $\gamma_c < 1$, we know by the Banach Fixed-Point Theorem that the fixed-point exists and is unique. □

Now we analyze the update to the main policy, that incorporates the subgoal value estimates into the bootstrap target. We assume we have a finite number of state-action pairs n, with parameterized action-values $q(\cdot; \mathbf{w}) \in \mathbb{R}^n$ represented as a vector with one entry per state-action pair. Value iteration to find q^* corresponds to updating with the Bellman optimality operator

$$(Bq)(s,a) \stackrel{\text{def}}{=} r(s,a) + \sum_{s'} P(s'|s,a)\gamma(s') \max_{a' \in \mathcal{A}} q(s',a') \tag{3.9}$$

On each step, for the current $q_t \stackrel{\text{def}}{=} q(\cdot; \mathbf{w}_t)$, if we assume the parameterized function class can represent Bq_t, then we can reason about the iterations of $\mathbf{w}_1, \mathbf{w}_2, \ldots$ obtain when minimizing distance between $q(\cdot; \mathbf{w}_{t+1})$ and Bq_t, with

$$q(s,a;\mathbf{w}_{t+1}) = (Bq(\cdot;\mathbf{w}_t))(s,a)$$

Under function approximation, we do not simply update a table of values, but we can get this equality by minimizing until we have zero Bellman error. Note that $q^* = Bq^*$, by definition.

In this *realizability* regime, we can reason about the iterates produced by value iteration. The convergence rate is dictated by γ_c, as is well known, because

$$\|Bq_1 - Bq_2\|_\infty \leq \gamma_c \|q_1 - q_2\|_\infty$$

Specifically, if we assume $|r(s,a)| \leq r_{\max}$, then we can use the fact that (1) the maximal return is no greater than $G_{\max} \stackrel{\text{def}}{=} \frac{r_{\max}}{1-\gamma_c}$, and (2) for any initialization q_0 no larger in magnitude than this maximal return we have that $\|q_0 - q^*\|_\infty \leq 2G_{\max}$. Therefore, we get that

$$\|Bq_0 - q^*\|_\infty = \|Bq_0 - Bq^*\|_\infty \leq \gamma_c \|q_0 - q^*\|_\infty$$

and so after t iterations we have

$$\begin{aligned}\|q_t - q^*\|_\infty = \|Bq_{t-1} - Bq^*\|_\infty \leq \gamma_c \|q_{t-1} - q^*\|_\infty \leq \gamma_c^2 \|q_{t-2} - q^*\|_\infty \ldots \\ \leq \gamma_c^t \|q_0 - q^*\|_\infty = \gamma_c^t G_{\max}\end{aligned}$$

We can use the exact same strategy to show convergence of value iteration, under our subgoal-value bootstrapping update. Let $r_{\text{sub}}(s,a) \stackrel{\text{def}}{=} \sum_{s'} P(s'|s,a) v_{\text{sub}}(s')$,

assuming $v_{\text{sub}} : \mathcal{S} \to [-G_{\max}, G_{\max}]$ is a given, fixed function. Then the modified Bellman optimality operator is

$$(B^\beta q)(s,a) \stackrel{\text{def}}{=} r(s,a) + \beta r_{\text{sub}}(s,a) + (1-\beta)\sum_{s'} P(s'|s,a)\gamma(s')\max_{a' \in \mathcal{A}} q(s',a') \tag{3.10}$$

Proposition 3 (Convergence rate of tabular value iteration under subgoal bootstrapping)**.** *The fixed point $q^*_\beta = B^\beta q^*_\beta$ exists and is unique. Further, for q_0, and the corresponding $\mathbf{w}_0$, initialized such that $|q_0(s,a;\mathbf{w}_0)| \leq G_{max}$, the value iteration update with subgoal bootstrapping $q_t = B^\beta q_{t-1}$ for $t = 1, 2, \ldots$ satisfies*

$$\|q_t - q^*_\beta\|_\infty \leq (1-\beta)^t \gamma_c^t \frac{r_{max} + \beta G_{max}}{1 - (1-\beta)\gamma_c}$$

Proof. First we can show that B^β is a $\gamma_c(1-\beta)$-contraction. Assume we are given any two vectors q_1, q_2. Notice that $\gamma(s) \leq \gamma_c$, because for our problem setting it is either equal to γ_c or equal to zero at termination. Then we have that for any (s,a)

$$\begin{aligned}
&|(B^\beta q_1(s,a) - (B^\beta q_2)(s,a)| \\
&\qquad = |(1-\beta)\sum_{s'} P(s'|s,a)\gamma(s')[\max_{a' \in \mathcal{A}} q_1(s',a') - \max_{a' \in \mathcal{A}} q_2(s',a')]| \\
&\qquad \leq (1-\beta)\gamma_c \sum_{s'} P(s'|s,a)|[\max_{a' \in \mathcal{A}} q_1(s',a') - \max_{a' \in \mathcal{A}} q_2(s',a')]| \\
&\qquad \leq (1-\beta)\gamma_c \sum_{s'} P(s'|s,a) \max_{a' \in \mathcal{A}} |q_1(s',a') - q_2(s',a')| \\
&\qquad \leq (1-\beta)\gamma_c \sum_{s'} P(s'|s,a) \max_{s' \in \mathcal{S}, a' \in \mathcal{A}} |q_1(s',a') - q_2(s',a')| \\
&\qquad \leq (1-\beta)\gamma_c \sum_{s'} P(s'|s,a) \|q_1 - q_2\|_\infty \\
&\qquad = (1-\beta)\gamma_c \|q_1 - q_2\|_\infty
\end{aligned}$$

Since this is true for any (s,a), it is true for the max, giving

$$\|B^\beta q_1 - B^\beta q_2\|_\infty \leq (1-\beta)\gamma_c \|q_1 - q_2\|_\infty.$$

Because the operator is a contraction, since $(1-\beta)\gamma_c < 1$, we know by the Banach Fixed-Point Theorem that the fixed-point exists and is unique.

Now we can also use contraction property for the convergence rate. Notice first that we can consider $\tilde{r}(s,a) \stackrel{\text{def}}{=} r(s,a) + \beta r_{\text{sub}}(s,a)$ as the new reward, with maximum value $r_{\max} + \beta G_{\max}$. Further, the new discount is $(1-\beta)\gamma_c$. Consequently, the maximal return is $\frac{r_{\max}+\beta G_{\max}}{1-(1-\beta)\gamma_c}$.

$$\begin{aligned}\|q_t - q^*_\beta\|_\infty &= \|B^\beta q_{t-1} - B^\beta q^*_\beta\|_\infty \leq (1-\beta)\gamma_c \|q_{t-1} - q^*\|_\infty \dots \\ &\leq (1-\beta)^t \gamma_c^t \|q_0 - q^*\|_\infty \\ &\leq (1-\beta)^t \gamma_c^t \frac{r_{\max} + \beta G_{\max}}{1-(1-\beta)\gamma_c}\end{aligned}$$

□

This rate is dominated by $((1-\beta)\gamma_c)^t$, and for β near 1 gives a much faster convergence rate than $\beta = 0$. We can determine after how many iteration this term overcomes the increase in the upper bound on the return. In other words, we want to know how big t needs to be to get

$$(1-\beta)^t \gamma_c^t \frac{r_{\max} + \beta G_{\max}}{1-(1-\beta)\gamma_c} \leq \gamma_c^t G_{\max}.$$

Rearranging terms, we get that this is true for

$$t > \log\left(\frac{r_{\max} + \beta G_{\max}}{G_{\max}(1-(1-\beta)\gamma_c)}\right) / \log\left(\frac{1}{1-\beta}\right).$$

For example if $r_{\max} = 1$, $\gamma_c = 0.99$ and $\beta = 0.5$, then we have that $t > 1.56$. If we have that $r_{\max} = 10$, $\gamma_c = 0.99$ and $\beta = 0.5$, then we get that $t \geq 5$. If we have that $r_{\max} = 1$, $\gamma_c = 0.99$ and $\beta = 0.1$, then we get that $t \geq 22$.

3.6 Putting it All Together: The Full Goal-Space Planning Algorithm

We summarize the algorithm and provide the pseudocode for the full GSP algorithm along with model learning. Visualized in Figure 3.5, the steps of agent-environment interaction include:

1) take action A_t in state S_t, to get S_{t+1}, R_{t+1} and γ_{t+1};

2) query the model for $r_\gamma(S_{t+1}, g), \Gamma(S_{t+1}, g), \tilde{v}(g)$ for all g where $d(S_{t+1}, g) > 0$;

3) compute projection $v_{\text{sub}}(S_{t+1})$ using Eq. (3.6) and step 2;

4) update the main policy with the transition and $v_{\text{sub}}(S_{t+1})$, using Eq. (3.7).

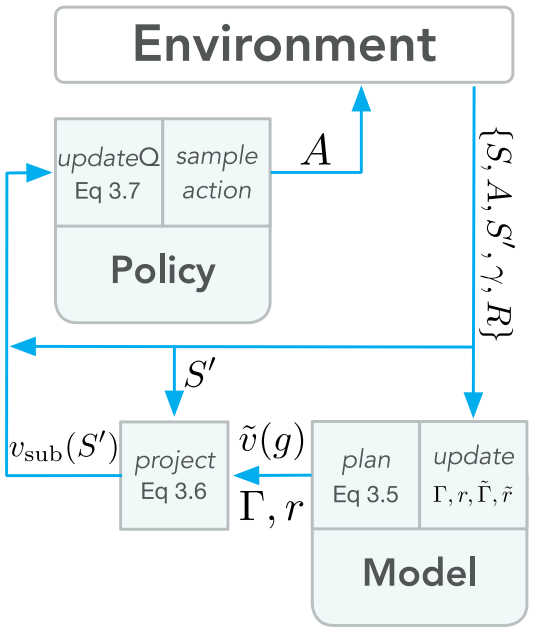

Figure 3.5: Overview of Goal-Space Planning.

All background computation is used for model learning using a replay buffer and for planning to obtain $\tilde{v}$, so that it can be queried at any time on step 2.

We summarize the above updates in pseudocode in Algorithm 2, specifying explicit parameters and how they are updated, and with a diagram in Figure 3.5. 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 learn action-value variants $r_\gamma(s, a, g; \theta^r)$, with parameters θ^r, to avoid importance sampling corrections. We learn the option-policy using action-values $\tilde{q}(s, a; \theta^\pi)$ with parameters θ^π. From this, we query the option-policy using $\pi_g(s; \theta^\pi) \stackrel{\text{def}}{=} \text{argmax}_{a \in \mathcal{A}} \tilde{q}(s, a, g; \theta^\pi)$. The policy π_g is not directly learned, but rather defined by $\tilde{q}$. Similarly, we do not directly learn $r_\gamma(s, g)$ and $\Gamma(s, g)$; instead, it is defined by $r_\gamma(s, a, g; \theta^r)$ and $\Gamma(s, a, g; \theta^r)$. Specifically, for model parameters $\theta = (\theta^r, \theta^\Gamma, \theta^\pi)$, we set $r_\gamma(s, g; \theta) \stackrel{\text{def}}{=} r_\gamma(s, \pi_g(s; \theta^\pi), g; \theta^r)$ and $\Gamma(s, g; \theta) \stackrel{\text{def}}{=} \Gamma(s, \pi_g(s; \theta^\pi), g; \theta^\Gamma)$. 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 our more constrained setting. One idea was to use subgoals that are often visited by the agent [35]. 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 [24].

Algorithm 2 Goal-Space Planning for Episodic Problems

Assume given subgoals $\mathcal{G}$ and relevance function d
Initialize table $v \in \mathbb{R}^{|\mathcal{G}|}$, main policy $\mathbf{w}$, model parameters $\theta = (\theta^r, \theta^\Gamma, \theta^\pi), \tilde{\theta} = (\tilde{\theta}^r, \tilde{\theta}^\Gamma)$
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 $s_{t+1}, r_{t+1}, \gamma_{t+1}$
    `ModelUpdate`$(s_t, a_t, s_{t+1}, r_{t+1}, \gamma_{t+1})$
    `Planning`()
    `MainPolicyUpdate`$(s_t, a_t, s_{t+1}, r_{t+1}, \gamma_{t+1})$

Algorithm 3 `MainPolicyUpdate`(s, a, s', r, γ)

$v_{\text{sub}} \leftarrow \max_{g \in \bar{\mathcal{G}}: d(s,g)>0} r_\gamma(s, g; \theta) + \Gamma(s, g; \theta)\tilde{v}(g)$
$\delta \leftarrow r + \gamma\beta v_{\text{sub}} + \gamma(1-\beta)\max_{a'} q(s', a'; \mathbf{w}) - q(s, a; \mathbf{w})$
$\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla_{\mathbf{w}} q(s, a; \mathbf{w})$

Algorithm 4 `Planning`()

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

Algorithm 5 `ModelUpdate`(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
$\delta^\pi \leftarrow \frac{1}{2}(r-1) + \gamma_{g'} \max_{a' \in \mathcal{A}} \tilde{q}(s', a', g'; \theta^\pi) - q(s, a, g'; \theta^\pi)$
$\theta^\pi \leftarrow \theta^\pi + \alpha^\pi \delta^\pi \nabla q(s, a, g'; \theta^\pi)$
// Update reward model and discount model
$a' \leftarrow \pi_{g'}(s'; \theta^\pi)$
$\delta^r \leftarrow r + \gamma_{g'} r_\gamma(s', a', g'; \theta^r) - r_\gamma(s, a, g'; \theta^r)$
$\delta^\Gamma \leftarrow m(s', g)\gamma + \gamma_{g'} \Gamma(s', a', g'; \theta^\Gamma) - \Gamma(s, a, g'; \theta^\Gamma)$
$\theta^r \leftarrow \theta^r + \alpha^r \delta^r \nabla r_\gamma(s, a, g'; \theta^r)$
$\theta^\Gamma \leftarrow \theta^\Gamma + \alpha^\Gamma \delta^\Gamma \nabla \Gamma(s, a, g'; \theta^\Gamma)$
// Update goal-to-goal models using state-to-goal models
for each g such that $m(s, g) > 0$ **do**
$\tilde{\theta}^r \leftarrow \tilde{\theta}^r + \tilde{\alpha}^r (r_\gamma(s, g'; \theta) - \tilde{r}_\gamma(g, g'; \tilde{\theta}^r)) \nabla \tilde{r}_\gamma(g, g'; \tilde{\theta}^r)$
$\tilde{\theta}^\Gamma \leftarrow \tilde{\theta}^\Gamma + \tilde{\alpha}^\Gamma (\Gamma(s, g'; \theta) - \tilde{\Gamma}(g, g'; \tilde{\theta}^r)) \nabla \tilde{\Gamma}(g, g'; \tilde{\theta}^\Gamma)$

3.7 Extending GSP to Deep RL

While GSP as described in the prior sections are for the general function approximation setting, there are several deep RL specific techniques that are used to stabilize learning with neural networks, which we use on our experiments. Many examples of these techniques can be seen in the classic Deep Q-Networks (DQN) [25]. The first involves the use of a separate *target network* to stabilize learning by slowing the changes to the bootstrap target as the agent's value estimate updates. Another change involves the use of *experience replay*, where prior experiences are stored in a buffer, which is then sampled from to perform batch updates using prior experience.

Here, we describe Double DQN (DDQN) [46], a small modification to DQN to reduce overestimation. In our implementation, rather than the standard DQN implementation of the target network which is updated to match the online network's parameters every some number of steps, we opt to use Polyak averaging, slowly updating the target network's parameters based on an exponential average of the prior online network's parameters every step. Assuming the standard environment interaction loop, we show the pseudocode for our

DDQN implementation with subgoal-value bootstrapping in Algorithm 6.

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

Add experience (s, a, s', r, γ) to replay buffer D_{main}
for n_{main} mini-batches **do**
  Sample batch $B_{main} = \{(s, a, r, s', \gamma)\}$ from D_{main}
  $v_{\text{sub}}(s) = \max_{g \in \bar{\mathcal{G}}: d(s,g) > 0} r_\gamma(s, g; \theta) + \Gamma(s, g; \theta)\tilde{v}(g)$
  $Y(r, s', \gamma) = r + \gamma\beta v_{\text{sub}} + \gamma(1 - \beta)q(s', \max_{a'} q(s', a'; \mathbf{w}), \mathbf{w}_{targ})$
  $L = \frac{1}{|B_{main}|} \sum_{(s,a,r,s',\gamma) \in B_{main}} (Y(r, s', \gamma) - q(s, a; \mathbf{w}))^2$
  $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} L$
  $\mathbf{w}_{targ} \leftarrow \rho\mathbf{w} + (1 - \rho)\mathbf{w}_{targ}$

We also use neural networks for GSP's option policies and state-to-subgoal models and apply similar modifications.

Algorithm 7 `ModelDDQNUpdate`(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 $B_{model} = \{(s, a, r, s', \gamma)\}$ from D_{model}
    $\gamma_{g'} \leftarrow \gamma(1 - m(s', g'))$
    // Update option policy
    $a' \leftarrow \text{argmax}_{a' \in \mathcal{A}} \tilde{q}(s', a', g'; \theta^\pi)$
    $\delta^\pi(s, a, s', r, \gamma) \leftarrow \frac{1}{2}(r - 1) + \gamma_{g'}\tilde{q}(s', a', g'; \theta^\pi_{targ}) - q(s, a, g'; \theta^\pi)$
    $\theta^\pi \leftarrow \theta^\pi + \alpha^\pi \nabla_{\theta^\pi} \frac{1}{|B_{model}|} \sum_{(s,a,r,s',\gamma) \in B_{model}} (\delta^\pi(s, a, s', r, \gamma))^2$
    $\theta^\pi_{targ} \leftarrow \rho_{model}\theta^\pi + (1 - \rho_{model})\theta^\pi_{targ}$
    // Update reward model and discount model
    $\delta^r(s, a, r, s', \gamma) \leftarrow r + \gamma_{g'}(\gamma, s')r_\gamma(s', a', g'; {\theta^r}_{targ}) - r_\gamma(s, a, g'; \theta^r)$
    $\delta^\Gamma(s, a, r, s', \gamma) \leftarrow m(s', g)\gamma + \gamma_{g'}(\gamma, s')\Gamma(s', a', g'; {\theta^\Gamma}_{targ}) - \Gamma(s, a, g'; \theta^\Gamma)$
    $\theta^r \leftarrow \theta^r - \alpha^r \nabla_{\theta^r} \frac{1}{|B_{model}|} \sum_{(s,a,r,s',\gamma) \in B_{model}} (\delta^r)^2$
    $\theta^\Gamma \leftarrow \theta^\Gamma - \alpha^\Gamma \nabla_{\theta^\Gamma} \frac{1}{|B_{model}|} \sum_{(s,a,r,s',\gamma) \in B_{model}} (\delta^\Gamma)^2$
    ${\theta^r}_{targ} \leftarrow \rho_{model}\theta^r + (1 - \rho_{model}){\theta^r}_{targ}$
    ${\theta^\Gamma}_{targ} \leftarrow \rho_{model}\theta^\Gamma + (1 - \rho_{model}){\theta^\Gamma}_{targ}$
  // Update goal-to-goal models using state-to-goal models
  ... same as in prior pseudocode.

We would obtain a version of GSP with DDQN for its main policy and state-to-subgoal model update if, in Algorithm 2, we replace `ModelUpdate` with `ModelDDQNUpdate` and `MainPolicyUpdate` with `MainPolicyDDQNUpdate`.

Chapter 4

Experiments with Goal-Space Planning

We investigate the utility of GSP, for (1) improving sample efficiency and (2) re-learning under non-stationarity. We also investigate how well GSP performs under different levels of model inaccuracy. We compare to Double DQN (DDQN) [46], which uses replay and target networks.[1] We layer GSP on top of this agent: the action-value update is modified to incorporate subgoal-value bootstrapping. By selecting $\beta = 0$, we perfectly recover DDQN, allowing us to test different β values to investigate the impact of incorporating subgoal values computed using background planning.

4.1 Experiment Specification

We perform experiments in the PinBall environment [21].[2] In this environment, the agent has to navigate a small ball to a destination in a maze-like environment with fully elastic and irregularly shaped obstacles. The state is described by 4 features: $x \in [0, 1], y \in [0, 1], \dot{x} \in [-1, 1], \dot{y} \in [-1, 1]$. The agent has 5 discrete actions: increase/decrease $\dot{x}$, increase/decrease $\dot{y}$, and nothing. The agent receives a reward of -5 per step and a reward of 10,000 upon termi-

[1] Aside from being a model-free baseline, DDQN can also be viewed as a simple form of Dyna, where the replay buffer is a non-parametric model that is used to update the main policy in the background.

[2] Our implementation is based on code at `https://github.com/amarack/python-rl`, which was released under the GPL-3.0 license. We have modified the environment to support additional features such as changing terminations, visualizing subgoals, and various bug fixes. Our code is released at `https://github.com/chunloklo/goal-space-planning`

nation at the goal location. PinBall has a continuous state space with complex and sharp dynamics that make learning and control difficult. We use a harder version of PinBall in our first experiment, shown in Figure 4.1, and a simpler one for the non-stationary experiment, shown in Figure 4.6, to allow DDQN a better chance to adapt under non-stationarity.[3]

Subgoal Selection and Definition The set of subgoals are chosen such that (1) they roughly cover the environment in terms of (x, y) locations and (2) it should be possible to access at least one subgoal from each subgoal, such that the higher level goal-space MDP is connected. Note that this network of subgoals does not necessarily contain the optimal path, and in our experiments, they do not. We leave combining GSP with subgoal discovery algorithms for future work.

For each subgoal g with location (x_g, y_g), we set $m(s, g) = 1$ for $s = (x, y, \dot{x}, \dot{y})$ if the Euclidean distance between (x, y) and (x_g, y_g) is below 0.035. Using a region, rather than requiring $(x, y) = (x_g, y_g)$, is necessary for a continuous state space. The agent's velocity is not taken into account for subgoal termination. The width of the region for the initiation function is 0.4. The exact layout of the environment, positions of these subgoals and initiation functions are shown in Figure 4.1 and 4.6

4.1.1 Algorithm Hyperparameters

The different hyperparameters we use in experiment 1 and 2 are summarized in Table 4.1.

For both experiments, we use the Adam optimizer [20] for training both the main policy and the subgoal models. We use the default hyperparameters for Adam except the step size ($b_1 = 0.9, b_2 = 0.999, \epsilon = 1e^{-8}$). The main policy was trained with 4 mini-batches per step with batch size of 16, while the subgoal models were trained with 1 mini-batch per step with the same batch size. We use the ϵ-greedy exploration strategy, with ϵ fixed to $\epsilon = 0.1$ in our

[3]The pinball configurations used are based on the "simple" and "slightly harder configuration" found at `http://irl.cs.brown.edu/pinball/`.

Common Hyperparameters	
b_1, b_2, ϵ (Adam)	$0.9, 0.999, 1e^{-8}$
mini-batches (policy) n_{main}	4
mini-batches (model) n_{model}	1
batch size $\lvert B_{main}\rvert, \lvert B_{model}\rvert$	16
ϵ (ϵ-greedy)	0.1
Experiment 1 Hyperparameters	
γ	0.99
α^r, α^Γ	$5e^{-4}$
Model Network Hidden Layers (Γ, r_γ)	$[256, 256, 128, 128, 64, 64, 32, 32]$
Policy Network Hidden Layers	$[256, 256, 128, 128, 64, 64]$
ρ_{model}	0.4
Experiment 2 Hyperparameters	
γ	0.95
α^r, α^Γ	$1e^{-3}$
Model Network Hidden Layers (Γ, r_γ)	$[128, 128, 128, 128, 64, 64]$
Policy Network Hidden Layers	$[256, 256, 128, 128, 64, 64]$
ρ_{model}	0.1

Table 4.1: Summary of hyperparameters used in experiment 1 and 2 for DDQN and GSP (where relevant). Network architecture lists the sizes of hidden layers from closest to the input layer (left) to closest to the output layer (right). Each layer except the last uses a ReLU activation function.

experiments. We use DDQN to learn r_γ and set $\pi_g(s) = \text{argmax}_{a \in A}\, r_\gamma(s, a, g)$.

For experiment 1, $\gamma = 0.99$, the model step sizes $\alpha^r = \alpha^\Gamma = 5e^{-4}$, and $\rho_{model} = 0.4$. For experiment 2, $\gamma = 0.95$, $\alpha^r = \alpha^\Gamma = 1e^{-3}$, and $\rho_{model} = 0.1$. We selected the learning rate for Adam and the Polyak averaging rate ρ for updating the main policy in each experiment using the methodology described below.

Network Architecture We use separate feedforward neural networks for learning the main policy, Γ, and r_γ, with ReLU activation function for each layer aside from the output layer. Each layer's weights are initialized using He uniform initialization [13], with the bias weights being initialized to 0.001. Each network output a vector of length 5, one for each action. The number of layers and size of each layer varied between experiment 1 and 2, with details listed in Table 4.1.

Hyperparameter	Experiment 1	Experiment 2
ρ	$[0.0125, \mathbf{0.025}, 0.05, 0.1]$	$[0.8, 0.4, 0.2, 0.1, \mathbf{0.05}, 0.025]$
α	$[1e^{-3}, \mathbf{5e^{-4}}, 3e^{-4}, 1e^{-4}]$	$[1e^{-2}, 5e^{-3}, \mathbf{1e^{-3}}, 5e^{-4}]$

Table 4.2: Ranges of hyperparameters swept for DDQN in experiment 1 across 4 seeds and experiment 2 across 8 seeds. The combination with the best performance in each sweep is **bolded**. These combinations were used for DDQN and GSP in experiment 1 and 2.

Hyperparameter Sweep Methodology We swept over the Polyak averaging rate ρ and step size hyperparameter α for Adam for both experiments for DDQN. Table 4.2 describes the ranges of ρ and α swept, and the best found combinations based on the average reward rate over 4 and 8 independent random seed for experiment 1 and 2, respectively. We then used these combinations of hyperparameters for both DDQN and GSP in experiment 1 and 2.

4.1.2 Learning Subgoal Models in PinBall

To ensure that we provide sufficient variety of data to learn the models accurately when pre-training subgoal models, the agent is randomly initialized in the environment at a valid state, ran in the environment for 20 steps with a random policy, then randomly reset again. To ensure that the agent gets sufficient experience near goal states, we initialize the agent, with a 0.01 probability, at states where $m(s, g) = 1$ for any g with added jitter sampled from $U(-0.01, 0.01)$ for each feature. The model is trained for 300k steps in this data gathering regime.

We restrict model update to relevant states in our experiments. Because the only relevant experience for learning r_γ and Γ are samples where $d(s, g) > 0$, we maintain a separate buffer for each subgoal g for learning $r_\gamma(s, g)$ and $\Gamma(s, g)$ such that all experience within that buffer are relevant. We require 10k samples in the buffer of each subgoal before learning for the corresponding r_γ and Γ begins, so that the mini-batches drawn contain a sufficiently diverse set of samples.

Similarly, a sample is only relevant for updating $\tilde{\Gamma}$ and $\tilde{r}_\gamma$ if $m(s, g) > 0$ for some g, but this might not be true for samples stored in the buffers for learning Γ and r_γ. To be able to obtain batches where all samples are relevant for learning $\tilde{\Gamma}$ and $\tilde{r}_\gamma$, the agent uses another buffer that exclusively stores samples where $m(s, g) > 0$ to learn $\tilde{\Gamma}$ and $\tilde{r}_\gamma$.

Experiments 1 and 2 described in this chapter all follow this procedure for pre-training the model. What these learned models are like, their empirical accuracy, and the effect of varying the levels of training for our model are further discussed in Section 4.3.

4.1.3 Optimizations for GSP using Fixed Models

It is possible to reduce the computation cost of GSP when learning with a pre-trained fixed model, as is the case in experiment 1. When the subgoal models are fixed, v_{sub} for an experience sample does not change over time as all components that are used to calculate v_{sub} are fixed. This means that the agent can calculate v_{sub} when it first receives the experience sample, save it in the buffer, and use the same calculated v_{sub} whenever this sample is drawn from the buffer to update the main policy. By doing so, v_{sub} only needs to be calculated once per sample, instead of with every update. This is beneficial when training neural networks, where each sample is often used multiple times to update the network.

An additional optimization possible on top of caching of v_{sub} in the replay buffer is to batch the calculation of v_{sub} of multiple samples together, which can be more efficient than calculating v_{sub} individually for the single sample received 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_{sub} for all samples in this buffer at once and adds the samples alongside v_{sub} to the main buffer. This intermediate buffer is then emptied and added to again every step. We use these optimizations and set the maximum size for the intermediate buffer to 1024 in experiment 1.

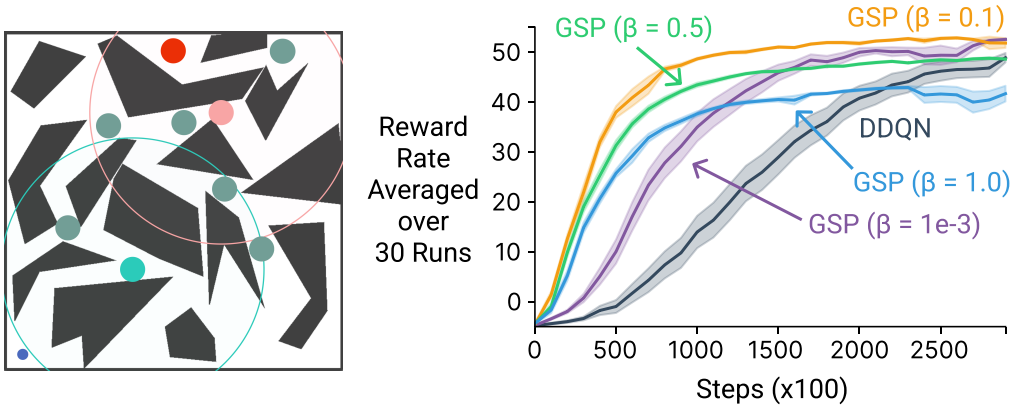

Figure 4.1: **(left)** The harder PinBall environment used in our first experiment. The dark gray shapes are obstacles the ball bounces off of, the small blue circle the starting position of the ball (with no velocity), and the red dot the goal (termination). Solid circles indicate the location and radius of the subgoals (m), with wider initiation set visualized for two subgoals (pink and teal). **(right)** Performance in this environment for GSP with a variety of β and DDQN (which is GSP with $\beta = 0$), with the standard error shown. Even just increasing to $\beta = 0.1$ allows GSP to leverage the longer-horizon estimates given by the subgoal values, making it learn much faster than DDQN. Once β is at 1, where it fully bootstraps off of potentially suboptimal subgoal values, GSP still learns quickly but levels off at a suboptimal value, as expected.

4.2 Experiment 1: Comparing GSP with Pre-learned Models and DDQN

We first investigate the utility of the models after they have been learned in a pre-training phase. The models use the same updates as they would when being learned online, and are not perfectly accurate. Pre-training the model allows us to ask: if the GSP agent had previously learned a model in the environment—or had offline data to train its model—can it leverage it to learn faster now? One of the primary goals of model-based RL is precisely this re-use, and so it is natural to start in a setting mimicking this use-case. We assume the GSP agent can do many steps of background planning, so that $\tilde{v}$ is effectively computed in early learning; this is reasonable as we only need to do value iteration for 9 subgoals, which is fast. We test GSP with $\beta \in [10^{-3}, 0.1, 0.5, 1.0]$.

We see in Figure 4.1 that GSP learns much faster than DDQN, and reaches

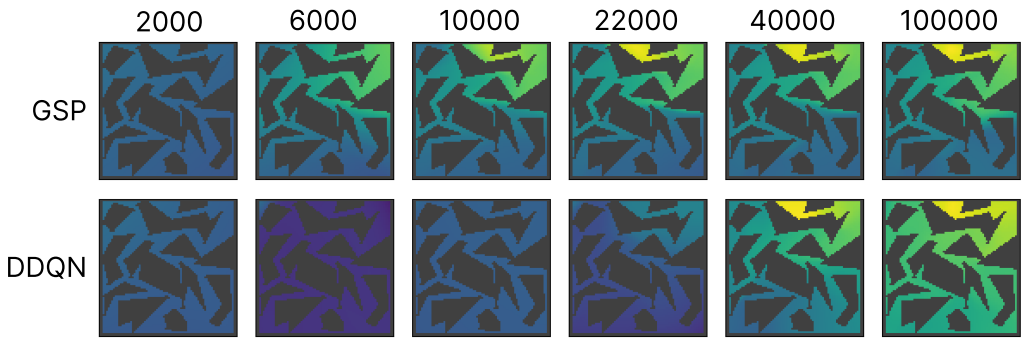

Figure 4.2: Visualizing the action-values for DDQN and GSP ($\beta = 0.1$) at various points in training. Yellow and purple indicate states with high and low value, respectively.

the same level of performance. This is the result we should expect—GSP gets to leverage a pre-trained model, after all—but it is an important sanity check that using models in this new way is effective. Of particular note is that even just increasing β from 0 (which is DDQN) to $\beta = 0.1$ provides the learning speed boost without resulting in suboptimal performance. Likely, in early learning, the suboptimal subgoal values provide a coarse direction to follow, to more quickly update the action-values, which is then refined with more learning. We can see that for $\beta = 0.5$ and $\beta = 1$, we similarly get fast initial learning, but it plateaus at a more suboptimal point. For $\beta = 10^{-3}$ very close to zero, we see that performance is more like DDQN. But even for such a small β we get improvements.

To further investigate the hypothesis that GSP more quickly changes its value function early in learning, we visualize the value functions for both GSP and DDQN over time in Figure 4.2. After 2000 steps, they are not yet that different, because there are only four replay updates on each step and it takes time to visit the state-space and update values by bootstrapping off of subgoal values. By step 6000, though, GSP already has some of the structure of the problem, whereas DDQN has simply pushed down many of its values (darker blue).

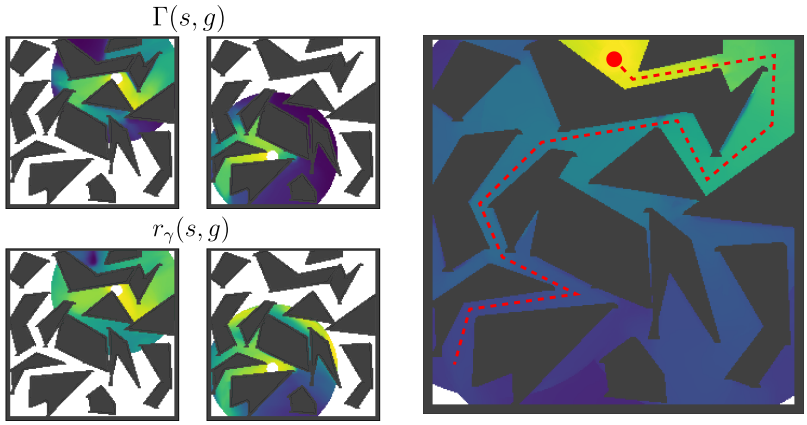

Figure 4.3: **(left)** Learned state-to-subgoal models for two different subgoals. **(right)** v_{sub} obtained from using the learned subgoal values in the projection step, as well as the trajectory that the ball must take to reach the goal. **(both)** Yellow and purple indicate states with high and low value, respectively. White indicates states where $d(s, g) = 0$.

4.3 Accuracy of the Learned Models

One potential benefit of GSP is that the models themselves may be easier to learn, because we can leverage standard value function learning algorithms. We visualize the models learned for the previous experiment, its accuracy, as well as the resulting v_{sub}.

In Figure 4.3 we see how learned state-to-subgoal models accurately learn the structure. Each plot shows the learned state-to-subgoal for one subgoal, visualized only for the initiation set $d(s, g) > 0$. We can see larger discount and reward values predicted based on reachability. However, the models are not perfect, as seen in Figure 4.4. The models learned tend to be more accurate closer to the goal, and less accurate further away. The absolute error of Γ can be as low as 0.01 close to the goal, but increase to 0.2 and higher further away. Similarly, the absolute error for r_γ can be as low as below 10 near goals, but can increase to over 100 further away. While the magnitudes of errors are not unreasonable, they are also not very near zero. This result is actually encouraging: inaccuracies in the model do not prevent useful planning.

It is informative to visualize v_{sub}. We can see in Figure 4.3 that the general structure is correct, matching the optimal path, but that it indeed looks sub-

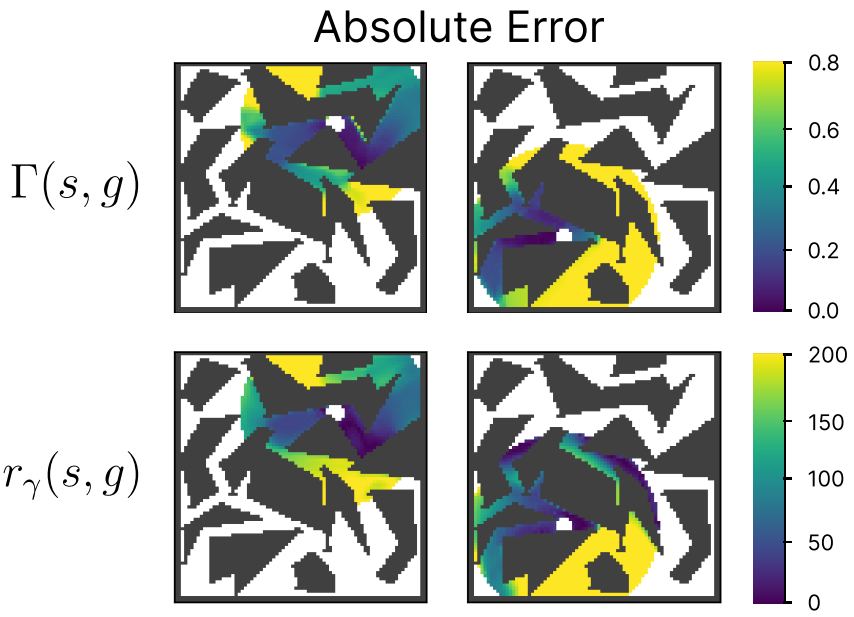

Figure 4.4: A heatmap of the absolute error of Γ and r_γ for two different subgoal models learned at various (x, y) against the ground truth obtained by rolling out the learned option policy at different (x,y) locations with 0 velocity. While the absolute error from states near subgoals can be quite low, they increase substantially as the state gets further away. White indicates states where $d(s, g) = 0$.

optimal compared to the final values computed in Figure 4.2 by DDQN. This inaccuracy is likely due to some inaccuracy in the models and to the fact that subgoal placement is not optimal. This explains why GSP has lower values particularly in states near the bottom, likely skewed downwards by v_{sub}.

Finally, we test the impact on learning using less accurate models. After all, the agent will want to start using its model as soon as possible, rather than waiting for it to become more accurate. We ran GSP using models learned online, using only 50k, 75k and 100k time steps to learn the models in a simpler PinBall environment. We then froze the models and allowed GSP to learn with them. We can see in Figure 4.5 that learning with too inaccurate of a model—with 50k—fails, but already with 75k performance improves considerably and with 100k we are already nearly at the same level of optimal performance as the pre-learned models. This result highlights it should be feasible to learn and use these models in GSP, all online.

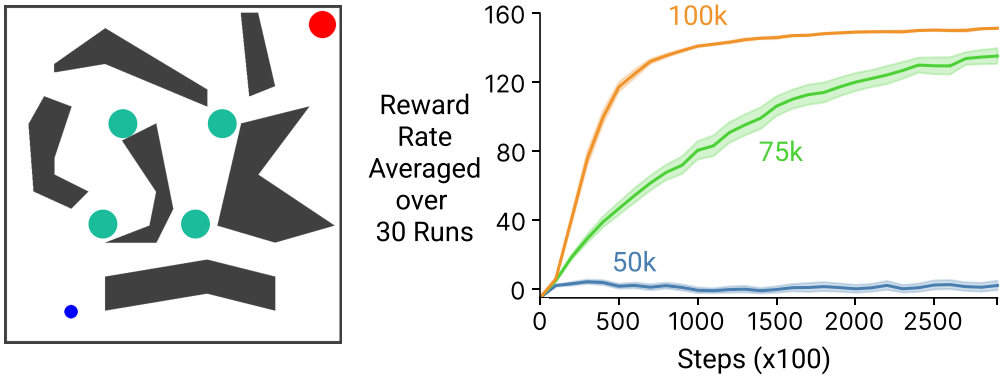

Figure 4.5: **(left)** The simple PinBall environment. **(right)** The impact on planning performance using frozen models with differing accuracy (shading shows the standard error).

4.4 Experiment 2: Adapting in Nonstationary PinBall

Now we consider another typical use-case for model-based RL: quickly adapting to changes in the environment. We let the agent learn in PinBall for 50k steps, and then switch the goal to a new location for another 50k steps. Goal information is never given to the agent, so it has to visit the old goal, realize it is no longer rewarding, and re-explore to find the new goal. This non-stationary setting is harder for DDQN, so we use a simpler configuration for PinBall, shown in Figure 4.6.

We can leverage the idea of exploration bonuses, introduced in Dyna-Q+ [36]. Exploration bonuses are proportional to the last time that state-action was visited. This encourages the agent to revisit parts of the state-space that it has not seen recently, in case that part of the world has changed. For us, this corresponds to including reward bonus r_{bonus} in the planning and projection steps: $\tilde{v}(g) = \max_{g' \in \bar{\mathcal{G}}: \tilde{d}(g,g')>0} \tilde{r}_\gamma(g,g') + \tilde{\Gamma}(g,g')\left(\tilde{v}(g') + r_{\text{bonus}}(g')\right)$ and $v_{\text{sub}}(s) = \max_{g \in \bar{\mathcal{G}}: d(s,g)>0} r_\gamma(s,g) + \Gamma(s,g)\left(\tilde{v}(g) + r_{\text{bonus}}(g)\right)$. Because we have a small, finite set of subgoals, it is straightforward to leverage this idea that was designed for the tabular setting. We use $r_{\text{bonus}}(g) = 1000$ if the count for g is zero, and 0 otherwise. When the world changes, the agent recognizes that it has changed, and resets all counts for the subgoals. Similarly, both

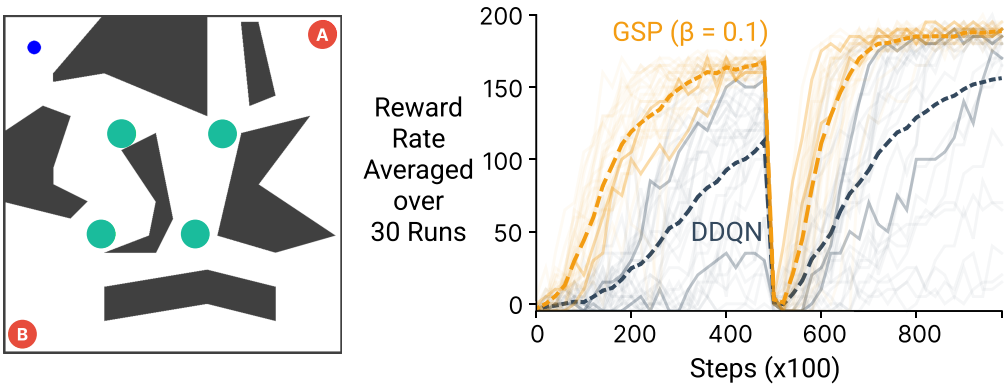

Figure 4.6: **(left)** The Non-stationary PinBall environment. For the first half of the experiment, the agent terminates at goal A while for the second half, the agent terminates at goal B. **(right)** The performance of GSP ($\beta = 0.1$) and DDQN in the environment. The mean of all 30 runs is shown as the dashed line. The 25th and 75th percentile run for each algorithm are also highlighted. We see that GSP with exploration bonus was able to adapt more quickly when the terminal goal switches compared to the baseline DDQN algorithm where goal values are not used.

agents (GSP and DDQN) clear their replay buffers.

The GSP agent can recognize the world has changed, but not how it has changed. It has to update its models with experience. The state-to-subgoal models and subgoal-to-subgoal models local to the previous terminal state location and the new one need to change, but the rest of the models are actually already accurate. The agent can leverage this existing accuracy.

In Figure 4.6, we can see both GSP and DDQN drop in performance when the environment changes, with GSP recovering much more quickly. It is always possible that an inaccurate model might actually make re-learning slower, reinforcing incorrect values from the model. Here, though, updating these local models is fast, allowing the subgoal values to also be updated quickly. Though not shown in the plot, GSP without exploration bonuses performs poorly. Its model causes it to avoid visiting the new goal region, so preventing the model from updating, because the value in that bottom corner is low.

4.5 Exploring a Natural Alternative: Incorporating Options into Dyna

A simple alternative to GSP, while still planning with temporally abstract models, is to incorporate options directly into Dyna as additional actions. In this section, we explore this alternative, describe some possible limitations, and compare it to GSP and DDQN.

To incorporate options into Dyna, we expand the action space to include options and learn the option and action value function $Q : \mathcal{S} \times (\mathcal{A} \cup \mathcal{O}) \to \mathbb{R}$. If an option o is selected when taking a greedy action according to Q, then the first action given by π_o is executed. The model in Dyna needs to include option models, which allows the agent to reason about accumulated rewards under an option, and outcome states after executing an option. Otherwise, the framework is identical to Dyna. It is a simple, elegant extension on Dyna that allows for planning with temporal abstraction.

However, this approach has several limitations. One limitation is that as we include new options—more abstraction—our value function needs to reason over more actions. Our proposed approach allows us to obtain the benefits of abstraction, without modifying the form of the policy. The model itself is like an option model, but it is used to directly reason about values for low-level states and actions. Another limitation is that the model in Dyna is the standard state-to-state model. Though Dyna with options has not been extended to function approximation — somewhat surprisingly — the natural extension suffers from similar problems of model errors and the use of expectation models as standard Dyna.

We compare Dyna with options, DDQN, and GSP in the simple PinBall environment. For Dyna with options, we use the subgoal-conditioned models pre-learned by GSP as options models and set the predicted next state of each option to $(x_g, y_g, 0, 0)$. We found Dyna with options difficult to get working. Instead, we used a modified version that only plans over options. This avoids learning and using primitive action models. We see in Figure 4.7 that this modified variant actually outperformed DDQN, but learned slower than GSP.

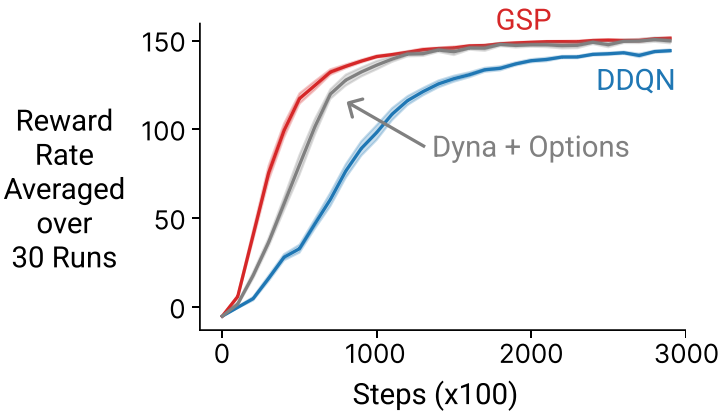

Figure 4.7: The performance of Dyna with options, GSP ($\beta = 0.1$), and DDQN in the simple PinBall environment. Dyna with options is depicted by the grey line. Dyna with options learns slower than GSP with its best beta parameter, but faster than DDQN, and ultimately achieves the same performance as GSP with its best configuration. Results are over 30 seeds as before. Dyna with options is implemented with separate step size and Polyak averaging rate hyperparameters for the separate primitive action and option value networks, (0.001, 0.1) and (0.005, 0.05), respectively.

Chapter 5

Conclusion and Future Work

In this thesis, we introduced a new planning framework, called Goal-Space Planning (GSP). The key idea is to plan in a much smaller space of subgoals, and use these (high-level) subgoal values to update state values using subgoal-conditioned models. We show that, in the PinBall environment, that (1) the subgoal-conditioned models can be accurately learned using standard value estimation algorithms and (2) GSP can significantly improve speed of learning, over Double DQN. The formalism avoids learning transition dynamics and iterating models, two of the sources of failure in previous model-based RL algorithms. GSP provides a new approach to incorporate background planning to improve action-value estimates, with minimalist, local models and computationally efficient planning.

Many new technical questions are introduced along with this new formalism. We have only tested GSP with pre-learned models and assumed a given set of subgoals. Our initial experiments learning the models online, from scratch, indicate that GSP can get similar learning speed boosts. Using a simple recency buffer, however, accumulates transitions only along the optimal trajectory, sometimes causing the models to become highly inaccurate part-way through learning, causing GSP to fail. An important next step is to incorporate smarter strategies, such as curating the replay buffer, to learn these models online. The other critical open question is in subgoal discovery. We somewhat randomly selected subgoals across the PinBall environment, with a successful outcome; such an approach is unlikely to work in many en-

vironments. In general, option discovery and subgoal discovery remain open questions. 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 the agent plan more efficiently.

There are also other open questions. The approach we took in this work, iterating through each subgoals during an update and learning a separate model for each, is unlikely to scale as the number of subgoals grow. Another avenue of improvement is further considering how to select β, the tradeoff between using fast changing subgoal values and the slower but eventually optimal standard bootstrap target. It is possible that we could adaptively change β over time depending on the agent's state of knowledge about the environment to adaptively rely more on subgoal values when the current value estimates are inaccurate, and more on real experience when fine tuning behaviour.

References

[1] Arthur Aubret, Laetitia matignon, and Salima Hassas. DisTop: Discovering a Topological representation to learn diverse and rewarding skills. *arXiv:2106.03853 [cs]*, 2021.

[2] Alex Ayoub, Zeyu Jia, Csaba Szepesvari, Mengdi Wang, and Lin Yang. Model-Based Reinforcement Learning with Value-Targeted Regression. In *International Conference on Machine Learning*, 2020.

[3] Andre Barreto, Diana Borsa, Shaobo Hou, Gheorghe Comanici, Eser Aygün, Philippe Hamel, Daniel Toyama, Jonathan hunt, Shibl Mourad, David Silver, and Doina Precup. The Option Keyboard: Combining Skills in Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2019.

[4] André Barreto, Shaobo Hou, Diana Borsa, David Silver, and Doina Precup. Fast reinforcement learning with generalized policy updates. *Proceedings of the National Academy of Sciences*, 117(48), 2020.

[5] Veronica Chelu, Doina Precup, and Hado P van Hasselt. Forethought and hindsight in credit assignment. In *Advances in Neural Information Processing Systems*, 2020.

[6] Rohan Chitnis, Tom Silver, Joshua B. Tenenbaum, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Learning Neuro-Symbolic Relational Transition Models for Bilevel Planning. *arXiv:2105.14074 [cs]*, 2021.

[7] Rohit K. Dubey, Samuel S. Sohn, Jimmy Abualdenien, Tyler Thrash, Christoph Hoelscher, André Borrmann, and Mubbasir Kapadia. SNAP:Successor Entropy based Incremental Subgoal Discovery for Adaptive Navigation. In *Motion, Interaction and Games*, 2021.

[8] Scott Emmons, Ajay Jain, Misha Laskin, Thanard Kurutach, Pieter Abbeel, and Deepak Pathak. Sparse graphical memory for robust planning. In *Advances in Neural Information Processing Systems*, 2020.

[9] Amir-massoud Farahmand. Iterative Value-Aware Model Learning. In *Advances in Neural Information Processing Systems 31*, 2018.

[10] Amir-massoud Farahmand, Andre M S Barreto, and Daniel N Nikovski. Value-Aware Loss Function for Model-based Reinforcement Learning. In *International Conference on Artificial Intelligence and Statistics*, 2017.

[11] Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. TreeQN and ATreeC: Differentiable Tree-Structured Models for Deep Reinforcement Learning. In *International Conference on Learning Representations*, 2018.

[12] Robert Gieselmann and Florian T. Pokorny. Planning-Augmented Hierarchical Reinforcement Learning. *IEEE Robotics and Automation Letters*, 6(3), 2021.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[14] Christopher Hoang, Sungryull Sohn, Jongwook Choi, Wilka Carvalho, and Honglak Lee. Successor Feature Landmarks for Long-Horizon Goal-Conditioned Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2021.

[15] Chad Hogg, U. Kuter, and Hector Muñoz-Avila. Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning. In *AAAI Conference on Artificial Intelligence*, 2010.

[16] Zhiao Huang, Fangchen Liu, and Hao Su. Mapping state space using landmarks for universal goal reaching. In *Advances in Neural Information Processing Systems*, 2019.

[17] Taher Jafferjee, Ehsan Imani, Erin Talvitie, Martha White, and Micheal Bowling. Hallucinating Value: A Pitfall of Dyna-style Planning with Imperfect Environment Models. *arXiv:2006.04363 [cs, stat]*, 2020.

[18] Khimya Khetarpal, Zafarali Ahmed, Gheorghe Comanici, David Abel, and Doina Precup. What can I do here? A Theory of Affordances in Reinforcement Learning. In *International Conference on Machine Learning*, 2020.

[19] Junsu Kim, Younggyo Seo, and Jinwoo Shin. Landmark-Guided Subgoal Generation in Hierarchical Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2021.

[20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[21] G.D. Konidaris and A.G. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in Neural Information Processing Systems*, 2009.

[22] Nathan Lambert, Kristofer Pister, and Roberto Calandra. Investigating Compounding Prediction Errors in Learned Dynamics Models. *arXiv:2203.09637 [cs]*, 2022.

[23] Timothy A. Mann, Shie Mannor, and Doina Precup. Approximate Value Iteration with Temporally Extended Actions. *Journal of Artificial Intelligence Research*, 53, 2015.

[24] Amy McGovern and Andrew G Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *International Conference on Machine Learning*, 2001.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[26] Soroush Nasiriany, Vitchyr Pong, Steven Lin, and Sergey Levine. Planning with Goal-Conditioned Policies. In *Advances in Neural Information Processing Systems*, 2019.

[27] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. *Advances in Neural Information Processing Systems*, 2017.

[28] Yangchen Pan, Muhammad Zaheer, Adam White, Andrew Patterson, and Martha White. Organizing Experience: A Deeper Look at Replay Mechanisms for Sample-Based Planning in Continuous State Domains. In *International Joint Conference on Artificial Intelligence*, 2018.

[29] Warren B. Powell. What you should know about approximate dynamic programming. *Wiley InterScience*, 2009.

[30] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal Value Function Approximators. In *International Conference on Machine Learning*, 2015.

[31] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839), 2020.

[32] David Silver, Richard S Sutton, and Martin Müller. Sample-based learning and search with permanent and transient memories. In *International Conference on Machine Learning*, 2008.

[33] David Silver, Hado Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David Reichert, Neil Rabinowitz, Andre Barreto, and Thomas Degris. The Predictron: End-To-End Learning and Planning. In *International Conference on Machine Learning*, 2017.

[34] Satinder Singh, Andrew Barto, and Nuttapong Chentanez. Intrinsically Motivated Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2004.

[35] Martin Stolle and Doina Precup. Learning Options in Reinforcement Learning. In *Abstraction, Reformulation, and Approximation*, 2002.

[36] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

[37] Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2), 1999.

[38] Richard S Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M Pilarski, Adam White, and Doina Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *International Conference on Autonomous Agents and Multiagent Systems*, 2011.

[39] Richard S Sutton, A R Mahmood, and Martha White. An emphatic approach to the problem of off-policy temporal-difference learning. *The Journal of Machine Learning Research*, 2016.

[40] Richard S. Sutton, Marlos C. Machado, G. Zacharias Holland, David Szepesvari, Finbarr Timbers, Brian Tanner, and Adam White. Reward-Respecting Subtasks for Model-Based Reinforcement Learning. *arXiv:2202.03466 [cs]*, 2022.

[41] R.S. Sutton. Integrated architectures for learning planning and reacting based on approximating dynamic programming. In *International Conference on Machine Learning*, 1990.

[42] R.S. Sutton and A G Barto. *Reinforcement Learning: An Introduction*. MIT press, 1998.

[43] Erik Talvitie. Model regularization for stable sample roll-outs. In *Uncertainty in Artificial Intelligence*, 2014.

[44] Erik Talvitie. Self-Correcting Models for Model-Based Reinforcement Learning. In *AAAI Conference on Artificial Intelligence*, 2017.

[45] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value Iteration Networks. In *Advances in Neural Information Processing Systems*, 2016.

[46] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI conference on artificial intelligence*, 2016.

[47] Hado P van Hasselt, Matteo Hessel, and John Aslanides. When to use parametric models in reinforcement learning? In *Advances in Neural Information Processing Systems*, 2019.

[48] A Venkatraman, M Hebert, and J. Andrew Bagnell. Improving Multi-Step Prediction of Learned Time Series Models. In *AAAI Conference on Artificial Intelligence*, 2015.

[49] Yi Wan, Abhishek Naik, and Richard S. Sutton. Average-Reward Learning and Planning with Options. In *Advances in Neural Information Processing Systems*, 2021.

[50] Theophane Weber, Sebastien Racanière, David P Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, David Silver, and Daan Wierstra. Imagination-Augmented Agents for Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2017.

[51] Martha White. Unifying task specification in reinforcement learning. In *International Conference on Machine Learning*, 2017.

[52] Jason Wolfe, Bhaskara Marthi, and Stuart Russell. Combined Task and Motion Planning for Mobile Manipulation. *International Conference on Automated Planning and Scheduling*, 2010.

[53] Lunjun Zhang, Ge Yang, and Bradly C. Stadie. World Model as a Graph: Learning Latent Landmarks for Planning. In *International Conference on Machine Learning*, 2021.

[54] Tianren Zhang, Shangqi Guo, Tian Tan, Xiaolin Hu, and Feng Chen. Generating adjacency-constrained subgoals in hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, 2020.