Leveraging Generic Problem Structure for Efficient Reinforcement Learning

by

Kenneth John Young

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

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

In this dissertation, I investigate how we can exploit generic problem structure to make reinforcement learning algorithms more efficient. Generic problem structure means basic structure that exists in a wide range of problems (e.g., an action taken in the present does not influence the past), as opposed to structure which is specific to a particular problem (e.g., heuristics or theorems about which actions are superior in a particular game). My investigation is broken down into three major contributions.

The first contribution is to demonstrate, empirical and theoretically, that given some prior knowledge of the structure of the world, reinforcement learning methods which learn a world model can do a better job of exploiting that knowledge, to learn from experience, compared to model-free methods which learn a value function directly from experience. This validates the belief that model-based reinforcement learning improves sample efficiency by synthesizing imagined experience which generalizes beyond the data. While this belief is widely held, model generalization is an insufficient explanation because learned value functions also generalize. I address this gap with theoretical and empirical results illustrating how world model generalization is, in a sense, inherently more powerful than value function generalization.

The second contribution is an algorithm that exploits knowledge of network structure to improve credit assignment in networks of discrete stochastic neurons, where each neuron is treated as a reinforcement learning agent. Training neural networks with discrete stochastic units is challenging as backpropa-

gation is not directly applicable, nor are the reparameterization tricks often used in networks with continuous stochastic variables. I propose Hindsight Network Credit Assignment (HNCA) for gradient estimation in networks of discrete stochastic neurons. HNCA can be seen as a middle-ground between backpropagation (which is intractable for nontrivial stochastic networks) and REINFORCE (which tends to be high variance). HNCA produces unbiased gradient estimates with provably lower variance than REINFORCE. The computational cost of HNCA is on the same order as a forward pass through the network, hence learning is not a significant bottleneck. Empirical results demonstrate that HNCA significantly reduces variance in the gradient estimates compared to REINFORCE, which in turn significantly improves performance.

The third contribution is an approach to option discovery motivated by the idea that, as a consequence of the spatiotemporal locality structure of the world, optimal actions in temporally contiguous states will tend to be strongly interdependent. Motivated by this idea, I propose an approach called Option Iteration (OptIt) which distills a set of options from the results of a computationally expensive search procedure. Intuitively, OptIt aims to discover a set of options such that for every trajectory segment of some length, at least one option in the set is a good match to the improved policy which results from running a search procedure in each state in the segment. This leads to options that capture relationships among the best actions in temporally contiguous states while allowing for uncertainty in which option is best in a given situation. The resulting set of options guides the search procedure resulting in a process of iterative improvement where better options lead to better search, which in turn facilitates the discovery of better options.

There is good reason to believe that prior knowledge of problem structure is necessary to make meaningful learning possible. However, the last several

decades of research have shown that encoding specific human expertise into our systems tends to lose out in the long run compared to methods that scale well with computation and data. Acknowledging both these points, it makes sense to focus our efforts on developing methods which exploit structure that is as generic as possible, allowing the agent to learn more specific world knowledge from experience and computation. By virtue of being broadly applicable, generic structure prunes the search space of solutions more and remains relevant as more computation and data are applied to a problem. On the other hand, although specific structure might improve performance early on, it can quickly become irrelevant as it may be deduced by a combination of learning and generic structure. While it's not always clear where the line should be drawn between generic and specific, acknowledging that a trade-off exists provides a useful guideline for which research directions are worth pursuing.

Preface

Of the three main contributions in this thesis, two are based on published conference papers and one is based on a preprint available on arxiv and under review at the time of this writing. Chapter 3 is based on a paper (Young, Ramesh, et al., 2023) presented at ICML 2023 in collaboration with Aditya Ramesh, Louis Kirsch and Jürgen Schmidhuber. Chapter 5 is based on a preprint (Young & Sutton, 2023) currently available on arxiv which was written in collaboration with my supervisor, Richard Sutton. Chapter 4 is based on a paper (Young, 2022) presented at AAAI 2022 on which I am the sole author, though it undeniably benefited from discussion with my supervisor and others outlined in the acknowledgements therein.

The delivery of good [machine learning] is to do as much nothing as possible.

— Samuel Shem, The House of God (originally referring to medical care).

Acknowledgements

I would like to thank my supervisor Richard Sutton for pushing me to think clearly while giving me the time and resources to explore my ideas. Rich's book and course are the reason I became interested in reinforcement learning in the first place. His insight, and his commitment to deep and clear thinking to probe the root of a problem, have been a constant source of inspiration and direction.

Besides Rich, I owe thanks to many of the other professors at the University of Alberta for their help and feedback, and for all they've contributed to creating an environment where ideas can flow freely. Mike Bowling, Martha White, Adam White, Matt Taylor, and Martin Meuller have all provided invaluable feedback and encouragement on my work at various points in my journey. Csaba Sepezvari showed me that so much more insight can come from careful mathematical analysis in reinforcement learning than I would have believed. Ryan Hayward was also instrumental in my early research by introducing me to the game of Hex which, aside from being a beautiful game, provided me with a formative research direction.

I would like to thank my parents, and my sister for boundless encouragement and support during my thesis research, as well as for encouraging the life-long curiosity and perseverance which allowed me to follow this path in the first place.

I would also like to thank Tian Tian for providing both emotional and intellectual support throughout this process.

I owe tremendous thanks to all the past and current member of the RLAI group for their friendship, numerous stimulating discussions, and feedback.

I also wish to thank my collaborators at IDSIA, Aditya Ramesh, Louis

Kirsch, and Jürgen Schmidhuber for all their help in my recent work and for broadening my research perspectives.

Finally, I wish to thank NSERC and Alberta Innovates for providing me with funding which has allowed me to focus on my research.

Contents

List of Tables

List of Figures

Chapter 1

Generic Problem Structure in Reinforcement Learning

Reinforcement learning (RL) refers to the problem faced by a goal-directed agent interacting with an initially unknown environment. At a high level, an agent is a system which takes in a stream of experience, processes it, and refines its own behaviour in response with the aim of influencing its future stream of experience to have some desirable properties. The challenges that arise from this simple setup are vast and multifaceted. While deciding how to act in a complex world is challenging in itself, an RL agent faces the additional challenge that it doesn't even know, a priori, the dynamics of the world it occupies. An effective agent must explore to gather data, make effective use of that data to learn enough about the world to serve its purposes, and choose how to act based on that knowledge, all while making the best of limited computational resources.

To make these challenges tractable, we as practitioners must carefully scope the kinds of problems we aim to address. First and foremost, we assume there is something to learn, some regularity of the experience stream that means the future looks similar to the past, and an agent's actions have consistent consequences. We as practitioners must define the framework that makes learning possible and allows us to compare and reason about different approaches, empirically and/or analytically. It is also important to periodically question both our foundational algorithms and our underlying assumptions, explicit or implicit. Through such questioning, we can build an understanding of how our

assumptions inform our algorithms and visa-versa and how we can push the boundaries of each to strike a balance of generality and practicality.

The basic question guiding this dissertation is

> Can reinforcement learning agents be designed to take advantage of generic problem structure to achieve more efficient learning and planning?

What constitutes "generic problem structure" is, of course, open-ended and subjective. I use this phrase to emphasize that while I believe encoding some knowledge of the underlying structure of the world into our agents will be necessary to develop agents with broad capabilities[1], I am more interested in encoding structure that will be true for a wide range of tasks and environments than task-specific structure. One algorithm I see as a canonical example of exploiting generic problem structure in RL is temporal difference (TD) learning (Sutton, 1988). TD learning, and related methods like Q-learning (Watkins et al., 1992), essentially exploit the fact that the future is independent of the past given the present, and likewise, action taken in the present can influence the future but not the past. When framed this way, this seems so obvious as to be trivial. However, incorporating this basic structure into algorithms has been a crucial step in the development of efficient RL agents. Overall, I believe the most useful structure to hardcode into our agents is the most general. As a consequence of this generality, such structure may also be easy to overlook because as humans we take it for granted. Hence, when we seek to develop and analyze algorithms it is worth taking the time to think carefully about our assumptions, even—or perhaps especially—those that seem the most obvious.

The meaning of "efficient" in the above thesis question is also worth unpacking. I use it to refer to both computational and sample efficiency. Better sample efficiency in the context of RL roughly means an agent requires fewer environment interactions to reach a given level of performance. If a problem has some known structure, it's natural to believe we can use it to improve

[1]Though oftentimes this structure may not be intentionally encoded into the agents but rather as a consequence of design choices derived from human intuition, which is itself motivated by some knowledge of the world.

sample efficiency, as utilizing prior knowledge means we can reduce the space of hypotheses we need to consider. Improving computational efficiency on the other hand roughly means an agent requires less computation to reach a particular level of performance. Computational efficiency could be measured, for example, in terms of wall-clock time on a particular system or the number of primitive arithmetical operations.

There are often tradeoffs between sample efficiency and computational efficiency, however, they are also intimately related. For example, if an agent can sample from a perfect model of the world, it can in principle use it to compute an optimal policy without observing any real-world samples. However, the process of querying the model may be computationally expensive, hence applying techniques which reduce the number of samples required for learning from environment interactions could also reduce the computational requirement to compute a strong policy even when a perfect model of the world is available.

This thesis investigates a number of techniques which can be used to exploit generic problem structure to improve sample and/or computational efficiency when designing RL algorithms. In Chapter 4 and Chapter 5 this investigation involves proposing novel algorithms. Chapter 3, on the other hand, focuses on improving our understanding of how existing model-based RL algorithms can make better use of underlying problem structure compared to model-free approaches.

Next, I will outline the three major contributions of this dissertation with a particular emphasis on how they relate to the central question of exploiting generic problem structure in RL.

The Benefits of Model-Based Generalization (Chapter 3). Model-Based RL is widely believed to have the potential to improve sample efficiency by allowing an agent to synthesize large amounts of imagined experience using a learned model of the environment. A common explanation is that a model can generalize from real experience to synthesize plausible imagined experience which can ultimately help an agent to learn a better policy from less real experience. However, the approximate value function learned in conventional

model-free RL algorithms can also generalize. Hence, a natural question is what makes model generalization better than value generalization?

I contribute a simple theorem which motivates how learning a model can allow an agent to capitalize on problem structure that is inherently unavailable to conventional value-based learning even when analogous structure is encoded into the model and value function classes. I also contribute illustrative experiments suggesting that this difference is important empirically.

A motivating example of problem structure used in this chapter is factored structure, where the environment states are assumed to consist of a number of separate factors such that the dynamics of each factor are a function of only a small number of other factors. However, the main theoretical results do not assume a particular kind of problem structure, but instead aim to show that model-based methods are inherently better able to exploit any structure that is known.

Hindsight Network Credit Assignment (Chapter 4). Backpropagation is a ubiquitous method for training neural networks. Backpropagation exploits the network structure to efficiently propagate gradient information to determine how to update parameters to improve an objective. However, backpropagation is not directly applicable to discrete stochastic neural networks, that is networks of units which select output values from a discrete set of choices with probability determined by their inputs.

I introduce an algorithm called Hindsight Network Credit Assignment (HNCA), a principled way to efficiently propagate gradient information in such discrete stochastic neural networks. Despite the underlying stochasticity of the network, HNCA provides unbiased gradient estimates. By taking advantage of the known connectivity structure of the network, HNCA provably reduces variance compared to REINFORCE, which does not exploit the network structure. This variance reduction leads to significantly better performance in practice while maintaining per-sample computational efficiency similar to that of backpropagation.

Option Iteration (Chapter 5). Discovering useful temporal abstractions, in the form of options, is widely thought to be key to applying RL and

planning to increasingly complex domains. I Introduce a simple approach, called Option Iteration (OptIt), to iteratively discover options by amortizing the results of a computationally expensive planning procedure.

OptIt is motivated by the perspective that one possible benefit of options is to allow an agent to represent dependencies in the joint distribution over optimal actions for temporally contiguous states. To give a motivating example, an agent approaching an intersection while driving in an unfamiliar city may be unsure whether it should turn left, turn right, or go straight. However, it should almost always rule out the large space of different ways to run off the road. Thus, we can narrow down the huge space of possible policies to a set of three temporally extended options and be fairly sure that one of them is optimal when restricted to a short time horizon into the future.

OptIt can be seen as a way to capture the idea that the world is structured such that the optimal action at a given time is likely to be highly informative about the optimal action for states in close temporal proximity.

Chapter 2

Background

This thesis focuses on exploring the potential of exploiting generic problem structure to improve the performance of reinforcement learning and planning algorithms. To understand the individual contributions, some general background on reinforcement learning and other machine learning topics is required. Readers familiar with the individual topics should be able to skip the corresponding sections and still understand the rest of the dissertation. Sections 2.3 and 2.4 present a somewhat nonstandard perspective on variational inference and its potential for learning sample models for RL which may be of independent interest even for a reader with a good understanding of those topics.

Readers interested in my contribution to understanding the benefits of model-based generalization in Chapter 3 should at least be familiar with Sections 2.1 and 2.2. Some of the experiments in that section also make use of a latent-state model, to which Sections 2.3 and 2.4 are also relevant. However, a lack of familiarity with the latter sections should not greatly harm understanding of the main ideas in the chapter.

Readers interested in my approach to assigning credit in discrete stochastic neural networks in Chapter 4 would benefit from a basic understanding of reinforcement learning, and in particular bandit problems as described in Sections 2.1. However, the main contribution is described in a fairly self-contained manner. The later experiments apply to training a variational auto-encoder, for which again it would be useful to understand the material in Sections 2.3

and 2.4.

My final contribution in Chapter 5 pertains to option discovery for planning and assumes familiarity with Sections 2.1, 2.2, and 2.5.

2.1 Reinforcement Learning

Reinforcement learning (RL) refers to the problem faced by a goal-directed agent interacting with an initially unknown environment. We formalize this interaction as a Markov decision process (MDP). An MDP $\mathcal{M}$ is defined by a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, \mu, \tilde{r})$. The agent begins in some state $S_0 \in \mathcal{S}$ drawn from the *start-state distribution* μ. At each time $t \geq 0$ an agent observes a state from the *state space* $S_t \in \mathcal{S}$ and based on this information selects an action from the *action space* $A_t \in \mathcal{A}$. Based on this action and the current state, the environment then transitions to a new state $S_{t+1} \in \mathcal{S}$ according to the *transition distribution* $p(s'|s,a) = \mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a)$, and provides a scalar reward drawn from the *reward distribution* $R_{t+1} \sim \tilde{r}(S_t, A_t)$. As the state-action conditional expectation of the reward distribution is usually the most important, I will give it its own name and notation, the *reward function* $r(S_t, A_t) = \mathbb{E}[R_{t+1}|S_t, A_t]$. I will assume $\mathcal{S}$ and $\mathcal{A}$ both contain finitely many elements for simplicity, but note that this can be relaxed.

The agent's behaviour is specified by a policy $\pi(a|s)$, which is a distribution over $a \in \mathcal{A}$ for each $s \in \mathcal{S}$. The agent's goal is, roughly, to obtain as much reward as possible in the long run. There are a few different ways this goal can be formalized. In this thesis, I will focus mainly on the episodic setting where there is a special terminal state $\perp$ from which no more reward is possible. The termination time T is the random time at which $\perp$ is reached. Note that T may be infinite for trajectories which never reach $\perp$. The sum of rewards obtained from some time t until T is called the return at time t $G_t = \sum_{k=t+1}^{T} R_k$. In this case, the agent's goal is to learn a policy π which maximizes the total expected return G_0. In cases where T is infinite, G_t may be ∞, $-\infty$, or undefined.

We define the state-value function under a particular policy π as $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$, that is the expected return from time t given the agent starts in

state s and follows the policy π until termination. Note that some additional assumptions are needed for the value function to exist in general. One sufficient condition is that T is finite with probability one under every policy π. Another possible condition is that all infinite trajectories correspond to $G_t = -\infty$, in which case $v_\pi(s)$ will also be $-\infty$ if π has a nonzero probability of generating an infinite trajectory. In cases where it is defined, $v_\pi(s)$ is known to be the unique solution to the Bellman equation (Bellman, 1957)

$$v_\pi(s) = \sum_a \pi(a|s) \left(r(s,a) + \mathbb{E}[v_\pi(S_{t+1}) | S_t = s, A_t = a] \right),$$

where the value of the terminal state $\perp$ is defined to be zero.

The optimal state-value function $v^\star(s) = \max_\pi v_\pi(s)$ is defined as the maximum state value over all policies π. $v^\star(s)$ is known to be the unique solution to the Bellman optimality equation

$$v^\star(s) = \max_a \left(r(s,a) + \mathbb{E}[v^\star(S_{t+1}) | S_t = s, A_t = a] \right).$$

Similarly, the action-value function of a policy is defined as $q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$, the expected return if action a is selected in state s and policy π is followed from that point forward. The optimal action-value function $q^\star(s,a) = \max_\pi q_\pi(s,a)$ is defined as the maximum action value over all policies π. $q^\star(s,a)$ is known to be the unique solution to the Bellman optimality equation

$$q^\star(s,a) = r(s,a) + \mathbb{E}[\max_{a'} q^\star(S_{t+1}, a') | S_t = s, A_t = a],$$

where the value of all actions in the terminal state $\perp$ are defined to be zero.

Learning approximations to state and/or action-value functions is often used as an intermediate step toward learning a good policy. Temporal difference (TD) learning (Sutton, 1988) is an approach to learning the state-value function of a particular policy based on the Bellman equation. TD learning uses an approximate state-value function $\hat{v}(\cdot;\theta)$, where θ represents some parameters of the function that we wish to tune to approximate $v_\pi(s)$. Toward this, TD learning updates θ according to the following update rule at each

time step:

$$\theta \leftarrow \theta + \alpha \left(R_{t+1} + \hat{v}(S_{t+1}; \theta) - \hat{v}(S_t; \theta)\right) \nabla_\theta \hat{v}(S_t; \theta),$$

where α is a step size that must be chosen as a hyperparameter of the algorithm, and actions are assumed to be sampled according to the policy we are interested in evaluating. This can be seen as taking a gradient descent step to move $\hat{v}(S_t; \theta)$ towards the target $R_{t+1} + \hat{v}(S_{t+1}; \theta)$. Note that if

$$\hat{v}(S_t; \theta) = \sum_a \pi(a|s) \left(r(s, a) + \mathbb{E}[\hat{v}(S_{t+1}; \theta) | S_t = s, A_t = a]\right)$$

for every state s, that is if $\hat{v}(S_t; \theta)$ obeys the Bellman equation, the expected update to θ is 0 for arbitrary S_t. TD learning can be seen as a gradient-descent-based approach to approximately solving the Bellman equation for state values. In this interpretation, the target $R_{t+1} + \hat{v}(S_{t+1}; \theta)$ is treated as fixed even though it also depends on the parameter θ. This subtlety complicates the analysis of the algorithm. Nonetheless, the interpretation is intuitively very useful.

Another common approach is Q-learning (Watkins et al., 1992). Q-learning is closely related to TD learning but is based on the Bellman optimality equation and learns an approximate optimal action-value function from observed transitions. Note that if we know $q^\star(s, a)$, then an optimal policy, that is a policy that achieves the highest value possible in every state, is simply $\pi^\star(a|s) = \operatorname{argmax}_a q^\star(s, a)$. Hence, selecting actions according to an approximation to $q^\star(s, a)$ can be a good way to obtain a strong policy. Q-learning uses an approximate action-value function $\hat{q}(\cdot, \cdot; \theta) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where θ represents some parameters of the function that we wish to tune to approximate $q^\star(s, a)$. Toward this, Q-learning updates θ according to the following update rule at each time step:

$$\theta \leftarrow \theta + \alpha \left(R_{t+1} + \max_a \hat{q}(S_{t+1}, a; \theta) - \hat{q}(S_t, A_t; \theta)\right) \nabla_\theta \hat{q}(S_t, A_t; \theta), \qquad (2.1)$$

where α is a step size that must be chosen as a hyperparameter of the algorithm. Actions are generally selected stochastically in a manner that favours those with high $\hat{q}(S_t, A_t; \theta)$ to strike a balance between exploiting promising

actions and exploring to improve the evaluation of others. Equation 2.1 can be seen as taking a gradient descent step to move $\hat{q}(S_t, A_t; \theta)$ toward the target $R_{t+1} + \max_a \hat{q}(S_{t+1}, a; \theta)$. Note that if

$$\hat{q}(s, a; \theta) = \mathbb{E}[R_{t+1} + \max_{a'} \hat{q}(s, a'; \theta) | S_t = s, A_t = a]$$

for every state s and action a, that is if $\hat{q}(s, a; \theta)$ obeys the Bellman optimality equation, the expected update to θ is 0 for arbitrary S_t, A_t. Q-learning can be seen as a gradient-descent-based approach to approximately solving the Bellman optimality equation for action values.

Algorithms like Q-learning are often combined with a technique called experience replay (ER; Lin, 1992). With ER, rather than performing the update in Equation 2.1 using each $(S_t, A_t, R_{t+1}, S_{t+1})$ tuple at time $t + 1$, tuples are stored in a large buffer. Random batches of tuples are then drawn from the buffer for each learning update. Compared to updating only at the time a transition is experienced, this has the benefit of more rapidly propagating value information throughout the state space. Mnih et al. (2015) applied Q-learning with ER and neural-network function approximation in the Deep Q-networks (DQN) algorithm. DQN demonstrated impressive performance in the Arcade Learning Environment (Bellemare et al., 2013), a framework which allows AI agents to play Atari 2600 games.

In practice, it is common to use a discounted variant of algorithms like TD learning and Q-learning. In the case of Q-learning, for example, this means using an update like the following:

$$\theta \leftarrow \theta + \alpha \left(R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a; \theta) - \hat{q}(S_t, A_t; \theta) \right) \nabla_\theta \hat{q}(S_t, A_t; \theta), \quad (2.2)$$

where the *discount factor* γ is a hyperparameter between 0 and 1. When γ is less than 1 this has the effect of discounting rewards proportionally to how many time steps in the future they occur. In this case, $\hat{q}(S_t, A_t; \theta)$ can be seen as estimating the expectation of the expected discounted return $G_t = \sum_{k=t+1}^{T} \gamma^{k-t+1} R_k$ under the optimal policy. This will lead to an agent that is more willing to give up long-term value to gain immediate reward. Sometimes γ is expressed as a property of the MDP, but I prefer to think of it as a

hyperparameter of our algorithm. By setting γ lower we reduce the variance in our updates that comes from a large sum of future values at the cost of optimizing a biased estimate of the long-term return, even though long-term return is what we ultimately care about.

Discounting also provides one approach to deal with continuing problems in which there is no terminal state and the agent environment interaction continues indefinitely, albeit not a particularly principled one (Naik et al., 2019). In this case, without discounting, the returns could be infinite and our value estimates would tend to diverge. In the continuing setting, discounting can be seen as a way to optimize a biased estimate of the average reward per time step. As the continuing setting is not a major focus of this thesis I will not discuss this setting in detail but refer the interested reader to the work of Wan et al. (2021) for some more sophisticated approaches and an overview of prior work in the area. In Chapter 3, I will use some continuing problems in my experiments, in which case I simply use discounting.

One relatively simple subset of RL problems is known as the multi-armed bandit setting, or simply the bandit setting. The name comes from the term one-armed bandit which is sometimes used to describe slot machines. The bandit setting can be framed as an RL problem in which there is only one state and termination always occurs after one step. In this setting, the transition distribution, start-state distribution, and state space are all trivial, hence a bandit problem is fully specified by just $\mathcal{A}$ and $\tilde{r}$. In the bandit setting, the action-value function simplifies to just an estimate of the expected reward associated with each action. Despite the apparent simplicity, the bandit setting still raises many interesting questions. For example, if the reward distribution $\tilde{r}$ is not deterministic there is a trade-off between choosing actions we have limited information about and those that we are fairly confident have high expected reward. The same may be true even for deterministic $\tilde{r}$ if the action space is large but has some structure that allows the agent to generalize from one action to another.

An intermediate setting between the bandit setting and the full RL problem is called the contextual bandit setting. In this setting, there can be multiple

states, but termination still occurs after one time step so the transition distribution is trivial. In this case, the agent must learn to select good actions conditioned on the state, as different states can have different expected rewards associated with a given action. However, unlike the full RL setting, feedback is always immediate and the agent need not worry about longer-term consequences of its actions.

2.2 Planning and Model-based Reinforcement Learning

Another class of approaches to the RL problem involves learning some approximation to p and r from the observed $(S_t, A_t, R_{t+1}, S_{t+1})$ tuples.[1] Having done this, an algorithm can then use computation to try to work out a good policy given the approximate learned p and r. Approaches which do this are called model-based, in contrast to algorithms like Q-learning which are called model-free. One common way to work out a good policy from a learned model involves using the model to simulate imagined experience, which can then be used in the update of a model-free algorithm.

In practice, a learned model can take many different forms. Given a particular (s, a) pair as input, a distribution model outputs some approximate representation of $p(s'|s, a)$ for every possible next state s' simultaneously, which tends to be intractable for sufficiently complex state spaces. An expectation model would output the expected next-state $\mathbb{E}[S_{t+1}|S_t = s, A_t = a]$, which must be used carefully to produce a valid algorithm (Kudashkina et al., 2021; Wan et al., 2019). A sample model learns to stochastically output states s' which approximate the distribution of $p(s'|s, a)$. Sample models often require relatively more sophisticated techniques, such as variational inference, to learn effectively.

Categorizing models as sample, distribution, or expectation models is a useful abstraction, however, note that it falls significantly short of expressing

[1] More generally we might approximate only some properties of p and r, such as expectations, however the distinction between model-based and model-free can get blurry if we try to push the definition too far.

the full space of different types of models that exist. For example, autoregressive models, which I will discuss in Section 2.3 allow for evaluation of $p(s'|s, a)$ for arbitrary s', like a distribution model, but do not represent the full distribution in closed form.

Planning refers to the set of techniques used to process a model, learned or given, to produce or improve a policy. When planning with a known model, the entire challenge is computational, we already know everything we need to compute the optimal policy, but we merely need to process it. The RL problem also includes statistical challenges in that we start off with limited knowledge of the environment and wish to gather and process limited data in order to learn about it efficiently.

Going forward, it will be useful to distinguish between two main types of planning which are often associated with model-based RL. The first type is decision-time planning, where in each visited state, an agent uses some computational procedure along with a model to work out which action to select. In decision-time planning, the policy is implicitly given by the model and (potentially stochastic) planning procedure. The other type of planning is background planning, where an agent maintains a policy or value function to select actions. In this case, planning is run in the *background*, using the model to improve the policy or value function, rather than for direct action selection. In practice, these two different kinds of planning are often blended together. The distinction between decision-time and background planning is better considered as a useful abstraction rather than a hard delineation between two entirely different approaches.

Algorithms that use background planning with a learned model commonly follow a pattern similar to Dyna (Sutton, 1990). In Dyna-Q, as described in Chapter 8 of the textbook of Sutton et al. (2020), an agent interacts with an environment in the usual way and, at each time step, performs Q-learning updates based on the observed transition. At each time step, the agent also uses the observed transition to update an approximate model of p and r. Finally, the agent uses the model to generate n additional imagined transitions, initialized from n previously visited states, and performs additional Q-learning

updates treating these imagined transitions as if they are real. A large variety of algorithms can be constructed based on minor variations of this approach. Often times an algorithm will forgo updating $\hat{q}$ based on real experience and instead use only model-generated experience. Rather than updating the model from only the most recently observed transition, it is common to store a buffer of real transitions and update the model using random batches from the buffer. Rather than using n random previously visited states and generating one-step transitions, it is common to perform multi-step model rollouts where the model-predicted next states are fed back into the model to predict the next state and reward after that.

One simple class of decision-time planning algorithms is Monte-Carlo search (MCS; Tesauro et al., 1996). Given the current state for which we'd like to select an action, MCS estimates the value of each action by running multiple simulations under some *rollout policy*. In the simplest version of MCS, each simulation can be run until termination occurs. One can also combine MCS with a learned approximate state-value function and truncate the simulations after some number of steps, using the approximate state-value function to estimate the expected return for the remainder of the episode. In either case, the action selected at the end of the search is generally the one with the highest average (estimated) return over all the simulations.

Monte-Carlo tree search (MCTS; Coulom, 2006; Kocsis et al., 2006) is another prominent class of decision-time planning approaches. MCTS takes the current state as the root and builds a tree of possible actions and resulting next states and rewards. At a high level, MCTS works by repeatedly applying a sequence of four steps, which I will collectively refer to as a *simulation*:

- **Selection**: the current tree is descended according to some *tree policy* until it reaches a leaf node where the tree policy selects a previously unexplored action. How best to select nodes in the tree is a difficult question involving an exploration-exploitation trade-off. One common approach is to use some variant of UCT (Kocsis et al., 2006), derived by treating the problem of node selection as a bandit problem and applying the low regret upper confidence bound (UCB) algorithm (Agrawal, 1995;

Katehakis et al., 1995). Recently it is common to incorporate some (usually learned) policy prior, in which case variants of the predictor+UCB (PUCB; Rosin, 2011) algorithm are often used. PUCB incorporates a principled mechanism to direct UCB sampling toward actions that are a priori expected to be good.

- **Expansion**: a previously unexplored action (and/or chance outcome in stochastic domains) is expanded to produce a new node.
- **Evaluation**: the value of the new node is estimated by some means. Historically, evaluating newly expanded nodes is often done by simulating a rollout until the end of an episode using some rollout policy.[2] More recently, it is more common to use a learned state-value function in this step (Gelly et al., 2007), either instead of, or in addition to a rollout.
- **Backup**: the newly obtained node value is backed up the tree to modify each ancestral action value. This backup propagates all the way up to the action values in the root node. Usually, each node maintains a count of the number of simulations that have passed through it, along with the average evaluation outcome for those simulations.

Whenever time runs out to select an action, or some predetermined simulation budget is expended, an action is selected based on the evaluation and/or simulation count for each action at the root node. Each of the above steps requires a number of additional design decisions to produce a concrete algorithm from the general framework of MCTS.

It is possible to apply MCTS to stochastic domains by incorporating branches corresponding to different chance outcomes in addition to different actions selected by the agent. However, if the number of possible chance outcomes is very large, the branching factor of the tree will blow up to the point where aggregating statistics for individual nodes will not be useful, as it will be rare to revisit the same node twice. As an extreme example, imagine running MCTS using a model which simulates the random motion of the leaves on trees moving in the background. Each time we expand a chance node the leaves may move slightly differently resulting in a new node expansion and evaluation,

[2] In which case, this step is often called "simulation" or "rollout" instead of evaluation.

eliminating any possibility of performing a deeper search or estimating a single, non-root, node value from multiple simulations. In this case, MCTS would essentially reduce to MCS. The issue is that MCTS does not generalize over states during the search. Alternative decision-time planning approaches which do generalize, like TD search (Silver et al., 2012) or PG search (Anthony et al., 2018), could help to address this.

Expert Iteration (Anthony et al., 2017) bootstraps the output of a search procedure, such as MCTS, to learn a parameterized policy and value function. The learned policy and value function are used to improve future search. This approach is used in AlphaZero (Silver et al., 2017) to obtain impressive empirical results in the game of Go. Expert Iteration can be seen as involving elements of both decision-time and background planning as it plans both for immediate action selection and to improve a policy and value function for improved future action selections.

I find it interesting to think that most of the prominent empirical successes of RL, even those that are generally considered model-free, can be seen as having a significant planning component. ER, for instance, can be seen as a simple model (Lin, 1992; van Hasselt et al., 2019) where experienced interactions are directly stored, and later replayed, for use in a learning update. Likewise, algorithms that employ multiple parallel actors (Mnih, Badia, et al., 2016) to generate experience for shared learning are not applicable to the standard definition of the RL problem where a single agent interacts with an environment. However, if a single agent uses a learned or given model it can potentially run multiple parallel simulations in its imagination.

2.3 Challenge of Learning Sample Models for Model-Based Reinforcement Learning

Perhaps the most natural kind of world model to learn for model-based RL is a sample model, which stochastically outputs states S' with distribution approximating $p(S'|s, a)$. Such a model effectively provides a simulator of the world, which enables us to generate imagined experience that can be used

to train an RL agent in essentially the same way as real experience is used. However, learning a sample model from data in a stochastic environment is conceptually challenging.

First, let's consider how a distribution model could be learned from data so that we can discuss the comparative difficulty of learning a sample model. Consider a dataset of N transitions (s_i, a_i, s'_i) indexed by $i \in \{0, 1, .., N-1\}$ and a parameterized distribution model $\hat{p}(s'|s, a; \theta)$, where θ is a set of learnable parameters.[3] For a distribution model, we assume $\hat{p}(s'|s, a; \theta)$ has a simple closed form with respect to s', for example, if s' is represented as a vector of binary values, $\hat{p}(s'|s, a; \theta)$ could be modelled as a vector of Bernoulli variables with means output by a neural network which takes s, a as input. As long as we can also differentiate $\hat{p}(s'|s, a; \theta)$ with respect to θ, it is straightforward to optimize the log-likelihood of the data under the model by stochastic gradient descent in the following loss:

$$\mathcal{L} = -\sum_{i=0}^{N-1} \log(\hat{p}(s'_i|s_i, a_i; \theta)). \tag{2.3}$$

We can optimize this loss by sampling individual transitions, or batches of transitions. Given the transitions are drawn from the ground truth world model and the (s_i, a_i) pairs are assumed to come from some data distribution D, this is equivalent to optimizing an unbiased empirical estimate of the expectation over the data distribution of the KL divergence between the true model and our sample model:

$$E_{S,A\sim D}[KL(p(S'|S, A), \hat{p}(S'|S, A))]$$
$$= \int_{s,a,s'} \mathbb{P}_D(S = s, A = a)(p(s'|s, a)(\log(p(s'|s, a)) - \log(\hat{p}(s'|s, a; \theta)))dsdads'.$$

The data distribution D may, for example, be the stationary distribution of an agent following a certain behaviour policy. KL divergence is always nonnegative, and zero if and only if the two distributions match exactly. Furthermore, one can bound the detriment in expected return suffered by following a policy

[3]Throughout this section I will use θ to represent an arbitrary set of all learnable parameters. When dealing with multiple parameterized functions each function might depend only on a disjoint subset.

optimized for a learned model in the true environment in terms of KL divergence between the two distributions (see, for example, the work of Ross et al. (2012)). Hence, KL divergence is a reasonable objective for model learning. However, optimizing Equation 2.3 relies on evaluating $\hat{p}(s'_i|s_i, a_i; \theta)$ for arbitrary transitions which is generally not possible for a sample model. Working around this limitation is a key challenge of learning sample models, which has been addressed with a wide variety of different techniques.

One interesting intermediate case is an autoregressive model. In this case, we assume each state s consists of M features such that $s = (s[0], ...s[M-1])$. Rather than approximating the full joint distribution of next-state features, an autoregressive model learns approximate conditional distributions for each state feature $\hat{p}(s'[i]|s'[0], ..., s'[i-1], s, a; \theta)$. Note that I have overloaded the function $\hat{p}$ here to refer to either the conditional distribution of features or of full next states.[4] The assumption is that the individual features have a simple enough form that we can explicitly represent their distribution in closed form (e.g., as probabilities for each of a finite set of values). We can then sample a random S' by sampling from each conditional distribution sequentially. It is possible to represent arbitrary transition distributions in this autoregressive form.[5] Furthermore, in this case we can evaluate $\hat{p}(s'|s, a; \theta) = \prod_{i=0}^{M-1} \hat{p}(s'[i]|s'[0], ..., s'[i-1], s, a; \theta)$ for arbitrary transitions (s, a, s'). Thus, with an autoregressive model, we can optimize the loss in Equation 2.3.

Although autoregressive models are popular, for example in large language models, they have some notable drawbacks. They require state features to be sampled sequentially, limiting the potential for parallelism. They also require a choice of closed-form distribution to model the individual features which may be limiting, for example, if the individual features can take continuous values. Finally, autoregressive models require a, potentially arbitrary, choice of the order in which state features are sampled, which can be unnatural and may

[4] I will continue to use $\hat{p}$ generically to represent learned probability distributions, the meaning should be clear from the context and this saves a lot of notation.

[5] On the other hand, this is not possible with an approximation of the simpler form $\hat{p}(s'[i]|s, a; \theta)$ as there is no way to capture correlation between features.

result in complicated conditional distributions that are difficult to model. For example, consider the challenge of autoregressively modelling the distribution of pixels in an image in arbitrary order.

One can even construct distributions where an autoregressive model faces fundamental computational challenges in representing the conditional distributions while representing the joint distribution of features is easy. For example, imagine a process which first selects a uniform random integer X and then generates $Y = f(X)$ using some function which is easy to compute but difficult to invert.[6] Now imagine we want to model $\mathbb{P}(YX)$ where YX represents the sequence of bits in the integers X and Y concatenated together. A bitwise autoregressive model of $\mathbb{P}(YX)$ would have to first sample a random output Y, and then invert f to accurately model $\mathbb{P}(X|Y)$. On the other hand, a more general model that samples from the joint distribution as a whole would be free to simulate sampling X followed by computing $Y = f(X)$, which is easy. See the work of Lin et al. (2020) for a more detailed discussion of some related issues with autoregressive models.

2.4 Latent-Variable Models and Variational Inference

An alternative class of sample model, which can help to address the challenges outlined in Section 2.3, is called latent-variable models. Latent-variable models will be the main type of learned sample model discussed in this thesis. There is a good deal of diversity within the class of latent-variable models, here I will discuss one of the simplest instantiations that could be used as a sample model for RL. The basic idea is to factor the nontrivial part of the randomness in our learned transition distribution into a separate noise variable Z, generally referred to as the *latent variable*. We then introduce a latent-variable dependent transition function $\hat{p}(s'|s, a, Z; \theta)$.

Normally, $\hat{p}(s'|s, a, Z; \theta)$ has a simple closed form as a function of s'. For

[6] More precisely, say $f(X)$ can be computed in polynomial time but $f^{-1}(Y)$ cannot. While the existence of such functions technically remains an open question, much of modern cryptography relies on assuming they exist.

example, if s' is a vector of binary values, $\hat{p}(s'|s, a, Z; \theta)$ could be represented by a neural network which takes s, a and Z as input, and outputs a vector of values between 0 and 1 representing the mean of a Bernoulli distribution for each element of s'. Note that this means the elements of a state S' sampled from this model are independent given Z. The latent variable Z itself will also be drawn from a simple closed form *prior* distribution $\hat{p}(Z|s, a; \theta)$ conditioned on the input s, a.[7] For example, $\hat{p}(Z|s, a; \theta)$ could also be represented as a neural network which takes s, a as input and outputs the mean of each element of Z, where each element of Z is once again a Bernoulli variable. I will use vectors of independent Bernoulli variables as a running example in what follows but other choices such as vectors of independent Gaussians are possible and common.

Since $\hat{p}(s'|s, a, Z; \theta)$ is parameterized by a generic neural network capable of representing complex transformations $\hat{p}(s'|s, a; \theta) \doteq \int_z \hat{p}(s'|s, a, z; \theta)\hat{p}(z|s, a; \theta)$ can capture complicated dependencies between elements of S' despite $\hat{p}(s'|s, a, Z; \theta)$ being a simple factored distribution. In order to draw a sampled S' from this model, we sample $Z \sim \hat{p}(Z|s, a; \theta)$ first, then compute $\hat{p}(s'|s, a, Z; \theta)$, and finally sample $S' \sim \hat{p}(S'|s, a, Z; \theta)$. The basic idea here is quite natural, we wish to represent a complex distribution, so we take a universal function approximator like a neural network and use the neural network to map an arbitrary, sufficiently rich, noise variable Z to a sample from the distribution we wish to represent.

The next question is: how do we train a latent-variable model? We'd like to optimize $\hat{p}(s'|s, a; \theta)$ with respect to the loss in Equation 2.3, however just evaluating $\log(\hat{p}(s'|s, a; \theta))$, in this case, requires us to integrate over Z which in the case where Z is a vector of Bernoulli variables means summing over combinatorially many values. We want Z to be large enough to capture fairly general dependencies among the elements of S', in which case explicit integration will usually be intractable. This is where the idea of variational

[7] One could also use a distribution $\hat{p}(z; \theta)$ which does not depend on x, or $\hat{p}(z)$ which is fixed instead of learned. These choices may however limit the representational power of the model as I will discuss shortly.

inference comes in. Next, I will explain how I think about the idea behind variational inference, and in particular, the way it is applied in variational auto-encoders (VAEs; Kingma and Welling, 2014; Rezende et al., 2014). My explanation is somewhat nonstandard, but I find it to be an intuitive way to understand the underlying idea.

Computing $\hat{p}(s'|s,a;\theta)$ requires an intractable integration over Z, but computing $\hat{p}(s'|s,a,z;\theta)$ given z is easy, it's just a forward pass through a neural network. If we imagine the ground truth transition distribution was also generated by first sampling Z, and we could observe the Z associated with each transition, we could minimize an empirical estimate of $\mathbb{E}_{S,A\sim D}\left[KL(p(S',Z|S,A),\hat{p}(S'|S,A,Z;\theta)\hat{p}(Z|S,A;\theta))\right]$, over a dataset of transitions. This is equivalent to minimizing the following loss:

$$\begin{aligned}
\mathcal{L} &= \sum_{i=0}^{N-1} \left(\log(p(s'_i,z_i|s_i,a_i)) - \log(\hat{p}(s'_i|s_i,a_i,z_i;\theta)\hat{p}(z_i))\right) \\
&= \sum_{i=0}^{N-1} \left(\log(p(s'_i,z_i|s_i,a_i)) - \log(\hat{p}(s'_i|s_i,a_i,z_i;\theta)) - \log(\hat{p}(z_i|s_i,a_i;\theta))\right) \\
&\propto -\sum_{i=0}^{N-1} \left(\log(\hat{p}(s'_i|s_i,a_i,z_i;\theta)) + \log(\hat{p}(z_i|s_i,a_i;\theta))\right),
\end{aligned}$$

where z_i is the observable latent variable associated with each transition. In the final line, I've dropped a term which does not depend on θ. This is a stronger requirement than only matching the distributions with respect to S'. If we manage to make $KL(p(S',Z|s,a),\hat{p}(S'|s,a,Z;\theta)\hat{p}(Z|s,a;\theta)) = 0$ this also guarantees $KL(p(S'|s,a),\hat{p}(S'|s,a;\theta)) = 0$ since if the joint distributions over S',Z are the same then logically the marginal distribution over S' must also be the same.

Unfortunately, there is no reason to believe the true transition distribution is actually generated by first sampling a latent variable Z, and even if it was, this latent variable is certainly not observable as required by the above objective. Variational inference works around this issue by augmenting the true transition distribution with an *approximate posterior* distribution

$\hat{q}(z|s,a,s';\theta)$ which predicts the probability that the noise variable Z took a particular value z given the observed transition. Often $\hat{q}(z|s,a,s';\theta)$ is parameterized as a neural network which outputs a distribution with the same form as $\hat{p}(z|s,a;\theta)$. If $\hat{p}(z|s,a;\theta)$ is a vector of uniform Bernoulli variables, $\hat{q}(z|s,a,s';\theta)$ could be parameterized as an equal length vector of Bernoulli variables with means output by a neural network, which now takes s, a, and s' as input. We can now consider minimizing an empirical approximation to $\mathbb{E}_{S,A}[KL(\hat{q}(Z|S,A,S';\theta)p(S'|S,A), \hat{p}(S'|S,A,Z;\theta)\hat{p}(Z|S,A;\theta))]$, let's start by expanding as follows:

$$\begin{aligned}
&\mathbb{E}_{S,A}[KL(\hat{q}(Z|S,A,S';\theta)p(S'|S,A), \hat{p}(S'|S,A,Z;\theta)\hat{p}(Z|S,A;\theta))] \\
&= \mathbb{E}_{S,A,S',Z}[\log(\hat{q}(Z|S,A,S';\theta)p(S'|S,A)) - (\log(\hat{p}(S'|S,A,Z;\theta)\hat{p}(Z|S,A;\theta))] \\
&\propto \mathbb{E}_{S,A,S',Z}[\log(\hat{q}(Z|S,A,S';\theta)) - \log(\hat{p}(S'|S,A,Z;\theta)) - log(\hat{p}(Z|S,A;\theta))] \\
&= \mathbb{E}_{S,A,S'}[\mathbb{E}_{Z\sim\hat{q}(Z|S,A,S';\theta)}[\log(\hat{q}(Z|S,A,S';\theta)) - \log(\hat{p}(S'|S,A,Z;\theta)) \\
&\qquad\qquad - \log(\hat{p}(Z|S,A;\theta))]]. \qquad (2.4)
\end{aligned}$$

Note that, in the second last line, I have dropped the term $\mathbb{E}_{S,A,S',Z}[\log(p(S'|S,A))]$ which does not depend on θ and thus is not relevant to its optimization. The final expression in Equation 2.4 is an expectation over the distribution of observed transitions, of another expectation involving functions we can evaluate in closed form. It's starting to look like something we can optimize, but I'll defer the question of how precisely to do it until a little later. First, let's think about whether this is a reasonable thing to do.

Even though we are using a learned distribution $\hat{q}(z|s,a,s';\theta)$ instead of some ground truth distribution over Z, we can still say that if we managed to reduce $KL(\hat{q}(Z|s,a,S';\theta)p(S'|s,a), \hat{p}(S'|s,a,Z;\theta)\hat{p}(Z|s,a;\theta))$ to zero for all s,a this will suffice to ensure that $KL(p(S'|s,a), \hat{p}(S'|s,a;\theta))$ is zero as well. In fact, we can make a stronger statement than this. In particular, KL divergence is additive in the following sense:

$$\begin{aligned}
KL(p(Y|X)p(X), q(Y|X)q(X)) &= KL(p(X), q(X)) + \mathbb{E}_x[KL(p(Y|X), q(Y|X)] \\
&\geq KL(p(X), q(X)).
\end{aligned}$$

In our situation, this gives

$$KL(\hat{q}(Z|s,a,S';\theta)p(S'|s,a),\hat{p}(S'|s,a,Z;\theta)\hat{p}(Z|s,a;\theta)) \geq KL(p(S'|s,a),\hat{p}(S'|s,a;\theta)).$$

Thus, by minimizing the left-hand side we are minimizing an upper bound on the right-hand side. In other words, however low we can make the left-hand side, we can guarantee the right-hand side is even lower. For this reason, the (negation of the) inner expectation in Equation 2.4 is often called the evidence lower bound (ELBO).

But why should we believe it's even possible to achieve $KL(\hat{p}(Z|s,a,s';\theta)p(S'|s,a),\hat{p}(S'|s,a,Z;\theta)\hat{p}(Z|s,a;\theta))$ close to zero given our restrictive choice of approximate posterior? A particular choice of $\hat{p}(s'|s,a,z;\theta)$ and $\hat{p}(z|s,a;\theta)$ will correspond to a certain posterior over z which we can work out using Bayes theorem:

$$\hat{p}(z|s',s,a;\theta) = \frac{\hat{p}(s'|s,a,z;\theta)\cdot\hat{p}(z|s,a;\theta)}{\int_{z'}\hat{p}(s'|s,a,z';\theta)\cdot\hat{p}(z'|s,a;\theta)dz'}.$$

To push the KL divergence to zero we would require $\hat{p}(z|s',s,a;\theta) = \hat{q}(z|s',s,a;\theta)$ for all z. Unfortunately, this posterior need not have the factored form we imposed on $\hat{q}(z|s,a,s';\theta)$, i.e. there may be dependence among the elements of z. The situation is improved by noting that there need not be one unique $\hat{p}(s'|s,a,z;\theta)$ which minimizes $KL(p(S'|s,a),\hat{p}(S'|s,a;\theta))$, there may be many ways to map the noise variable Z to S' such that our sample model captures the true distribution.

If there exists some mapping $\hat{p}(s'|s,a,z;\theta)$ that captures $p(s'|s,a)$ when we marginalize out z while also leading to a factored posterior $p(z|s',s,a;\theta)$, then that solution will be preferred in terms of Equation 2.4. But do such solutions even exist? In general, not necessarily, but for sufficiently expressive neural networks, along with a sufficiently rich z, it will be possible to make the KL divergence arbitrarily small. This is essentially analogous to saying there is a universal approximation theorem for variational inference, which I will next demonstrate there is, at least for the case of Bernoulli vectors.

If we consider the case where all relevant distributions are Bernoullis, then S' can take finitely many possible values. In particular, if S' is a binary

vector of length n, then S' can take $N = 2^n$ distinct values. Let's index these possible values as $s'_0, ..., s'_{N-1}$. For realistic data, it is likely that the vast majority of these values will have near zero probability under the true distribution $p(S'|s, a)$ so the effective N could be much smaller. Now consider the extreme case where z is a vector of N independent Bernoulli variables. We can then associate each possible value of s' with an element of z by choosing

$$\hat{p}(s'_i|s, a, z; \theta) = \prod_{j=0}^{i-1} \mathbb{1}(z[j] = 0)\, \mathbb{1}(z[i] = 1)$$

where $\mathbb{1}$ is the indicator function. In words, we deterministically map each z to the s'_i corresponding to the index of the first 1 in z.[8] We can then set $\hat{p}(z|s, a; \theta)$ such that

$$p(s'_i|s, a) = \prod_{j=0}^{i-1} \hat{p}(z[j] = 0|s, a; \theta)\hat{p}(z[i] = 1|s, a; \theta),$$

which ensures that $\hat{p}(s'_i|s, a; \theta) = p(s'_i|s, a)$ for all i. Furthermore, the true posterior will factor as

$$\hat{p}(z|s'_i, s, a; \theta) = \prod_{j=0}^{i-1} \mathbb{1}(z[j] = 0)\, \mathbb{1}(z[i] = 1) \prod_{j=i+1}^{N-1} \hat{p}(z[j]|s, a; \theta),$$

and thus can be represented by a factored approximate posterior $\hat{q}(z|s, a, s'_i; \theta)$. Intuitively, this just says that if we observe a particular y_i generated by the above process, we know that the first $i - 1$ elements of z must have been zero, and the ith element must have been one, but we have no information about the remaining elements of z beyond the prior. Note that this construction requires the prior $\hat{p}(z|s, a; \theta)$ to be parameterized and conditioned on s, a. It's an interesting question whether an analogous construction is possible with a fixed prior $\hat{p}(z)$.

Now let's consider the question of how we optimize Equation 2.4. We will use stochastic samples of S, A, S' observed from the true unknown transition

[8] For simplicity I didn't explicitly specify what happens if z is all zero. We can either make the final element of z one deterministically so this doesn't happen, or we actually only need $N - 1$ elements of z and the all-zero vector will correspond to s'_{N-1}.

dynamics, however, we still need to deal with the inner expectation. Again, we cannot simply sample $Z \sim \hat{q}(Z|s, a, s'; \theta)$ and optimize the inside of the expectation with respect to the samples as the sampling distribution itself depends on θ. However, as long as we can evaluate, sample from, and differentiate $\hat{q}(Z|s, a, s'; \theta)$, which we guarantee by design, we can use the following general identity to obtain an unbiased gradient estimate:

$$\frac{\partial}{\partial\theta}\,\mathbb{E}_{Z\sim q(Z;\theta)}[f(Z;\theta)] = \mathbb{E}_{Z\sim q(Z;\theta)}\left[\frac{\partial \log(q(Z;\theta))}{\partial\theta} f(Z;\theta) + \frac{\partial}{\partial\theta} f(Z;\theta)\right]. \quad (2.5)$$

In our case, we would take $q(z|\theta) = \hat{q}(z|s, a, s'; \theta)$ and $f(z;\theta) = \log(\hat{q}(z|s, a, s'; \theta) - \log(\hat{p}(s'|s, a, z; \theta)) - \log(\hat{p}(z|s, a; \theta))$. We can then derive an unbiased estimator of the gradient by sampling from $q(Z;\theta)$ and optimizing the inside of the expectation for the specific sample. In many situations, however, it is possible to derive lower variance unbiased gradient estimates such as the reparameterization trick (Kingma & Welling, 2014; Rezende et al., 2014). HNCA, described in Chapter 4 is also an example of an unbiased variance reduction technique applicable to variational inference. Biased gradient estimators such as the straight-through estimator (Bengio et al., 2013) can also be used and often perform well in practice.

I have described just one example of how variational inference could be used to train a sample model for model-based RL. In another common setup, rather than using the latent variable to parameterize only the noise, one can learn a mapping $\hat{p}(z|s;\theta)$ and then model the dynamics themselves in latent space as $\hat{p}(z'|z, a; \theta)$. An additional learned distribution $\hat{p}(s'|z';\theta)$ aims to reconstruct the distribution of next states (see e.g., Watter et al. (2015), Ha et al. (2018)). The overall transition distribution is then modelled as

$$\hat{p}(s'|s, a) = \int_{z,z'} \hat{p}(z|s;\theta)\hat{p}(z'|z, a; \theta)\hat{p}(s'|z';\theta)dzdz'. \quad (2.6)$$

In this case, we are effectively trying to transform states s into a new space in which the dynamics obey the factored structure imposed by $\hat{p}(z'|z, a; \theta)$. This approach has several benefits including being able to roll out the model for multiple steps in latent space without explicitly predicting the environment state in each step. This is essentially the approach of Dreamer (Hafner et al.,

2020; Hafner et al., 2021), a variant of which will be applied in Chapter 3. I will refer to this type of model as a *latent-state* model, and refer to the associated latent variable z as the *latent state*.

In addition to potential computational benefits, latent-state models may be beneficial in terms of stability since we can choose the latent state to be relatively simple and bounded regardless of the form of the underlying environment observations. Repeatedly applying a model to images of an Atari game for example can quickly lead to compounding errors which produce predictions that look nothing like plausible images. However, if we instead iterate the model in a restricted abstract latent space, such as the vector of categorical variables used in Dreamer, it's intuitively easier to imagine each element of the latent space mapping to a plausible image. Formalizing this intuition is beyond the scope of this thesis, but it's interesting to think about. An intuitively useful property for an imperfect learned model would be to converge to some steady state, which is in turn assigned some reasonable default value estimate, in the limit of repeated application rather than blowing up.

The application of variational inference to learn sample models for RL as well as machine learning more generally is a rich and active area of research. Nevertheless, the basic principles I have covered in this section apply quite generally.

2.5 The Options Framework and Temporal Abstraction

The planning algorithms discussed in the previous sections operate in the space of individual actions. Each node produced by MCTS corresponds to a single action, and Dyna-style approaches generally use a one-step model and action-value function that estimates the expected return conditioned on just one action. It's not hard to imagine how this might be limiting if actions correspond to very fine-grained choices like individual muscle twitches. In such cases, an MCTS approach will likely exhaust its search budget before looking far enough ahead to observe any meaningful changes to the state. Dyna-style

planning will slowly back up value one step at a time, making it intractable to back up information between meaningfully distinct events. As humans, it is clear that we can plan over much larger time horizons, in terms of temporally extended choices such as executing a left turn while driving or going to the grocery store. Options provide a framework (Sutton et al., 1999) to augment an RL agent with the capability for such temporally extended reasoning.

An option n consists of a 3-tuple $(\mathcal{I}_n, \pi_n, \beta_n)$. The option's policy $\pi_n(a|s)$ gives the probability of the agent selecting action a in state s while following option n. The initiation set $\mathcal{I}_n \subseteq \mathcal{S}$ is the set of states from which the option can be initiated, in many cases this is simply set to $\mathcal{S}$, such that the option can be initiated from arbitrary states. The termination function, $\beta_n(s)$ gives the probability of option n terminating after entering state s. Once defined, options can be used essentially interchangeably with actions in planning.

We can define an option conditional transition distribution $p(s'|s,n) = \mathbb{P}_{\pi_n,\beta_n}\left(S_{\tilde{T}} = s'|S_I = s\right)$ giving the probability that option n terminates in s' when initiated in s and following π_n, where $\tilde{T}$ and I denotes the random time at which option termination occurs, and the time at which the option is initiated respectively. Similarly, an option conditional reward function can be defined as the expected cumulative reward from initiation to option termination $r(s,n) = \mathbb{E}_{\pi_n,\beta_n}\left[\sum_{k=t}^{\tilde{T}} r(S_t, A_t)\middle| S_I = s\right]$. Note that actions are also options which terminate with probability one after just one step of execution. I will sometimes use the phrase primitive action to emphasize I am talking about a single-step action.

Options have at least two possible benefits. First, options facilitate more rapid temporal propagation of value when learning or planning (Sutton et al., 1999). This includes propagating information faster when background planning, as well as allowing the agent to look farther into the future for more informed decision-time planning. Second, options can facilitate exploration (Machado et al., 2017; McGovern et al., 2001) by providing bridges between distinct regions of state space that otherwise require very precise control to traverse.

While providing a set of prespecified options to an agent can be useful, it's

interesting to ask how an agent may build a set of useful options on its own. This is the problem of option discovery and is a very active area of research. A variety of methods have been proposed for option discovery, with a number of different motivations behind them. Some approaches aim to directly optimize options to facilitate good performance (Bacon et al., 2017; Veeriah et al., 2021). Others aim to learn options which help navigate between disparate regions of state space, for example by identifying bottleneck states (McGovern et al., 2001; Stolle et al., 2002), exploiting graph-theoretic properties of the transition dynamics (Machado et al., 2017), or encouraging options to contain a lot of information about their state at termination (Eysenbach et al., 2019; Gregor et al., 2016; Harutyunyan, Dabney, Borsa, et al., 2019).

Chapter 3

The Benefits of Model-Based Generalization in Reinforcement Learning

Recall that the main focus of this thesis is on the potential to develop more efficient reinforcement learning and planning algorithms by exploiting generic problem structure. This chapter introduces the first major contribution of this thesis and the first example of how algorithmic choices can impact an agent's ability to exploit generic problem structure. In particular, I demonstrate that when some knowledge of the structure of the MDP is available, a model-free approach like Q-learning is inherently less able to exploit the known structure than a model-based approach. The experiments in this section focus particularly on environments with factored structure,[1] which has the potential to allow for significant model generalization. However, the main theoretical results are not specific to factored structure but instead show that a model-based approach can better exploit known problem structure in general.

Model-based RL, which is described in more detail in Section 2.2, refers to the class of RL algorithms which learn a model of the world as an intermediate step to policy optimization. One important way such models can be used, which will be the focus of this chapter, is to generate imagined experience for training an agent's policy (Jordan, 1988; Munro, 1987; Schmidhuber, 1990; Sutton, 1990; Werbos, 1987). ER can be seen as a simple, nonparametric,

[1]Where the state consists of a set of state variables such that the distribution of each variable at the next time step depends on only a subset of the variables in the current state.

model (Lin, 1992; van Hasselt et al., 2019) where experienced interactions are directly stored, and later replayed for learning.

ER already captures many of the benefits associated with a learned model as compared to model-free incremental online algorithms (i.e., model-free algorithms which perform a learning update using each transition only at the time it is experienced). In particular, ER allows value to be rapidly propagated from states to their predecessors along previously observed transitions, without the need to actually revisit a particular transition for each step of value propagation. Propagating value only at the time a transition is visited can make model-free incremental online algorithms wasteful of data, particularly in environments where the reward signal is sparse.

As Lin (1992) and van Hasselt et al. (2019) have discussed, it is often not obvious why we'd expect experience generated by a learned model to improve upon ER, as an ER buffer is essentially a perfect model of the world insofar as the agent has observed it. This is especially true in the tabular case, where a model does not generalize from the observed transitions. It is also true for policy evaluation in the case where the value function and model are linear (Parr et al., 2008; Sutton et al., 2008). In this case, learning the least-squares linear model from the data, and then finding the TD(0) solution (Sutton, 1988) in the resulting linear MDP is identical to finding the TD(0) solution for the empirical MDP induced by the observed data. Hence, if we expect to obtain a sample efficiency benefit by using data generated by a learned model compared to ER, we should look beyond these cases.

One may argue that a parametric model that generalizes can generate a large amount of imagined experience that does not appear explicitly in the dataset. However, parametric value functions also generalize. Why should model generalization be inherently better than value function generalization? This question was already raised in the work of Lin (1992), which first introduced ER for RL. Section 3.2 gives a partial answer as a theorem which shows how learning a model as an intermediate step can narrow the space of possible value functions more than learning a value function directly from the data using the Bellman equation.

After motivating the benefit of model-based generalization theoretically, I will present an intuitive case where learning a parametric model is empirically beneficial with neural network function approximation. Subsequently, I will present extensive experiments, which highlight the sample efficiency benefits of model-based learning for online RL in cases where the environment has some underlying factored structure that can allow a learned model to generalize. I will also analyze an interesting instance I came across during these experiments where an agent using a learned model outperforms one using the perfect model due to smoothed reward and transition dynamics.

3.1 Related Work

There is a large body of empirical research showing the potential of learned models to improve sample efficiency (Asadi, 2015; Buckman et al., 2018; Curi et al., 2020; Deisenroth et al., 2011; Hafner et al., 2021; Janner et al., 2019; Kaiser et al., 2020). A relatively small body of work directly compares ER with learned models. Van Seijen et al. (2015) show an exact functional equivalence between a variant of replay with linear TD(0), and linear Dyna (Sutton, 1990). Pan et al. (2018) provide an empirical study comparing ER with learned models under a variety of search control strategies, that is, different methods for choosing which state-action pairs to update. Holland et al. (2018) empirically compare ER with a parametric model in the ALE and highlight the benefits of the model in particular when multi-step rollouts are used. Relative to these works, here I focus on understanding *how* a learned model can provide a benefit, and highlighting properties of environments where this benefit is most prominent.

Van Hasselt et al. (2019) make a strong case that ER provides many of the benefits of a learned model and argue that if a model is used only to generate experience starting from observed states it is unlikely to provide additional benefit. I investigate the comparison between ER and learned models further, and argue that there is good reason to believe a learned model can improve sample efficiency in environments with structure, such as factored dynamics.

This is true even if the model is used only to augment training data with rollouts starting from observed states.

Dong, Luo, et al. (2020) share my focus on highlighting situations where model-based RL provides a significant benefit. While I focus on the generalization benefits, they motivate the expressivity benefit of model-based RL by showing there exist MDPs where the optimal policy is exponentially more complex to represent than the dynamics.

The benefit of model smoothing, observed in some of the experiments in this chapter, may shed light on some observations in the literature. Hafner et al. (2021) suggest model smoothing as an explanation for why DreamerV2 can achieve good performance on Montezuma's Revenge without any sophisticated exploration mechanism. Holland et al. (2018) observe that their learned model sometimes outperforms the ground-truth model in Seaquest, though they do not suggest a specific explanation.

The results presented in this chapter have implications for *implicit* model-based algorithms (as defined in the survey paper of Moerland et al. (2020)) such as MuZero (Schrittwieser et al., 2020). In such algorithms, the model is not trained to predict future observations, but only task-relevant aspects of the future such as policy, value and reward. In light of Theorem 3.1, and the example in Section 3.3, it is unclear whether such techniques fully exploit the benefits of model-based learning due to the limited training signal used to constrain the model.[2] Indeed, there is empirical evidence suggesting that MuZero's sample efficiency can be improved by using additional training signals (Anand et al., 2022; Ye et al., 2021).

This chapter is loosely inspired by the literature on exploiting factored structure in MDPs (Diuk et al., 2009; Osband et al., 2014; Sallans et al., 2004; Strehl et al., 2007; Sun et al., 2019; Xu et al., 2020). As highlighted in Section 3.2, Theorem 3.1 is closely related to Theorem 2 of Sun et al. (2019), which shows that there exists a family of MDPs where a model-based approach can be exponentially more efficient than any model-free approach. I focus on

[2]Though Gehring et al. (2021) suggest that the implicit model-based parameterization itself may yield favourable gradient dynamics.

factored structure to motivate the benefit of model generalization but do not use algorithms explicitly designed to exploit this factored structure.

The benefit of model-based generalization can be seen as an example of the benefit of semi-supervised learning (Chapelle et al., 2006) in general. As another example of the latter, Ng et al. (2001) demonstrate that learning a generative model to predict $p(y, x)$ by naive Bayes and marginalizing to produce a classifier tends to outperform directly predicting $p(y|x)$ by logistic regression in the limit of low data. Model-based RL is similar in that we solve a larger problem, that is learning a world model, as an intermediate step to the more specific objective of learning a good policy. Ng et al. (2001) also find that logistic regression tends to outperform naive Bayes given sufficient data, at least when the true model is not realizable by the function class. Both Theorem 3.1 of the present chapter, and Theorem 2 of Sun et al. (2019) consider the realizable case. A more nuanced theory could highlight the trade-off between model-based and model-free learning when realizability fails or optimization is imperfect. I discuss this further in Section 3.6.

In this chapter, I focus on the advantages of using models to generate experience for policy improvement. Learned models can also be useful in other ways, for example, decision time planning for immediate action selection (Byravan et al., 2022; Chua et al., 2018; Richalet et al., 1978), improving credit assignment (Buesing et al., 2019; Heess et al., 2015; Schmidhuber, 1990), exploration (Pathak et al., 2017; Schmidhuber, 1990, 1997), representation learning (Gregor et al., 2019; Hessel et al., 2021; Lin et al., 1992), and in answering learned queries (Schmidhuber, 2015).

3.2 Theoretical Motivation for the Benefit of Model-Based Generalization

Theoretical comparison of model-based and model-free methods is challenging in that it is difficult to precisely define what makes an algorithm model-free. Given a known model class $\mathcal{M}$, Sun et al. (2019) define a model-free algorithm as one which accesses the state s only through its set of possible optimal

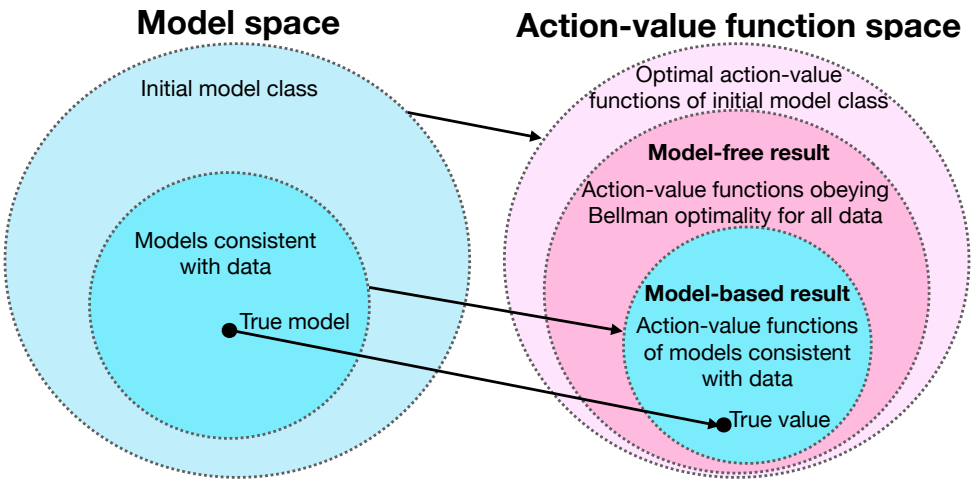

Figure 3.1: An illustration of the intuition behind Theorem 3.1. The hypothesis class over models is shown on the left and the resulting hypothesis class over optimal action-value functions is shown on the right. The diagram shows the difference between the model-based and model-free approaches to pruning optimal action-value functions based on observed data. The model-based result can be a drastically smaller set, and never larger, than the model-free result.

action-value functions $\{q^*_m(s,a)\}_{(m\in\mathcal{M},a\in\mathcal{A})}$, where $\mathcal{A}$ is the action set and q^*_m represents the optimal action-value function for a particular MDP m. Any states which have the same action values under all considered models are deemed indistinguishable to a model-free algorithm. Under this definition, they show there exist model classes where a model-based approach can be exponentially more sample efficient than any model-free approach.

One key insight of Sun et al. (2019) may be summarized as follows: even states which are indistinguishable in terms of their action values, for all MDPs in the model class, may contain distinct information about the environment dynamics. Perhaps counter-intuitively, this dynamics information which is unavailable to model-free methods can contain information that is relevant to predicting values.

Here, I present a simple theorem based on a setup similar to that of Sun et al. (2019). Relative to their general result about model-free algorithms as defined above, this result presents similar intuition in a simpler setting, while being more targeted to motivate my subsequent empirical comparison of

model-free learning with ER to using a learned model. In combination with the subsequent empirical results, this theorem can help practitioners gain more concrete intuition for how utilizing a learned parametric model can improve generalization, and thus sample efficiency.

I state the theorem informally here, and formally with proof in Appendix A.1. Figure 3.1 illustrates the key idea.

Theorem 3.1. *Consider a class of deterministic, episodic, MDPs $\mathcal{M}$ with fixed reward function, and transition function belonging to some known hypothesis class. Let H_Q be the associated class of optimal action-value functions for MDPs in $\mathcal{M}$. Now consider a dataset D of transitions. Let $H_B(D)$ be the subclass of action-value functions in H_Q which obey the Bellman optimality equation for the transitions in D and let $H_M(D)$ be the subclass of optimal action-value functions of MDPs in $\mathcal{M}$ which are consistent with D. Then the following are true:*

1. *$H_M(D) \subseteq H_B(D)$.*
2. *For any $N \in \mathbb{N}$ there exists some choices of $\mathcal{M}$ and D such that $\frac{|H_B(D)|}{|H_M(D)|} > N$.*
3. *For a tabular transition function class, that is one that includes every possible mapping from state-action pairs to next states, $H_M(D)$=$H_B(D)$.*

Intuitively, Theorem 3.1 states that, if we want to narrow down the possible optimal action-value functions from data, we can in general prune more (in fact an arbitrarily large factor more in certain cases) if we narrow down the possible models first than if we only demand that the value functions obey the Bellman optimality equation with respect to the observed data. The latter approach is closely analogous to running Q-learning to convergence on a fixed dataset D.[3] Part 3 of Theorem 3.1 states that the model-based approach offers no benefit for a tabular model class, but rather only for models which have some additional structure. In such cases, even if a model-based and model-free approach begin with the same hypothesis class, the model-free approach can

[3]Note that the latter approach meets the definition of model-free suggested by Sun et al. (2019) since if two states have the same action values in every MDP in $\mathcal{M}$ we can enforce Bellman optimality without knowing which of them was visited.

fail to leverage this structure and ultimately lose value-relevant information from the data in the process.

The proof of part 2 of Theorem 3.1 uses an example adapted from Sun et al. (2019) and illustrated in Figure 3.2. The basic idea is to set up a combination lock type problem where an a priori unknown sequence of actions leads to a reward of 1 while all other action sequences lead to a reward of 0. The problem class is such that using information about the dynamics, it is possible to work out the optimal policy from a single arbitrary trajectory. Without using the dynamics information, a model-free agent using Bellman consistency alone will only be able to prune a single possible value function per trajectory unless the data contains the optimal sequence. In words, we can construct a class of MDPs and an associated dataset such that the model-based approach can uniquely determine the optimal action-value function while the model-free approach is left with a class containing N elements for arbitrarily large N.

While somewhat contrived, the example in Figure 3.2 has simple factored structure. It isn't hard to imagine how similar situations could arise in practice. For example, one could imagine learning the basic rules of chess from a relatively small number of examples and from there working out a very strong policy by imagining different strategies using these learned rules. Other examples of problems with such simple factored structure are explored in the experimental portion of this chapter.

Theorem 3.1 provides useful intuition for how utilizing a learned parametric model can improve sample efficiency. However, there are important gaps between the assumptions of the theory and the practice of using neural network function approximation for value functions and models. With neural network function approximation, we have no guarantee of realizability (that the true model is actually in the class) though, with a sufficiently high capacity neural network, this can perhaps be assumed. Theorem 3.1 also assumes the model-based and model-free approaches have access to the same prior knowledge of the problem class. This is not true in practice where prior knowledge is encoded somewhat nebulously in the choice of neural network architecture.

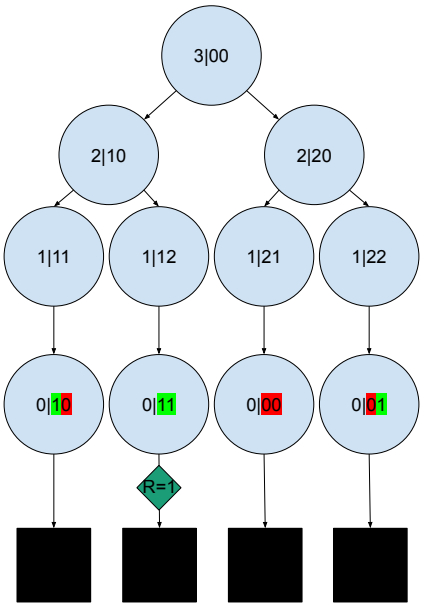

Figure 3.2: Illustration of the model class used as an example where selecting a hypothesis based on model consistency is arbitrarily more selective than Bellman consistency. The action space consists of 2 actions, left and right. The states consist of 1 component which acts as a countdown to termination along with M *passcode* components (with $M = 2$ in the figure) which act to record the action sequence executed so far. The digits take values in $\{0, 1, 2\}$ with 1 indicating the left action, 2 indicating the right action and 0 padding the future actions. When the countdown reaches 0 the correct passcode digits will switch to 1 and the incorrect digits to 0. In the following step, the episode will terminate. Note that termination always occurs at $t = M + 2$. A reward of 1 will be given on termination if and only if all passcode digits are 1, and the reward is otherwise 0. Different models in the class vary only in the initially unknown passcode.

Theorem 3.1 says nothing about the performance of an arbitrary model or value function parameterization on a single problem instance. For example, in the limiting case where the true value function is known a priori, no learning is necessary and any value function learned by a model-based approach may well be worse. Instead, Theorem 3.1 suggests that if we design a model architecture with favourable generalization properties for a problem class of interest, a model-based approach with this architecture will provide a sample efficiency benefit which cannot be replicated simply by encoding analogous generalization properties into a value function.

Theorem 3.1 also doesn't directly say anything about the relative per-

formance we can expect when using a model-based approach compared to a model-free approach. It tells us there exist datasets and model classes for which the model-based approach can rule out more value functions, but how does this affect the performance of an agent using this information to choose actions? Moreover, what if rather than an arbitrary dataset, we allow the agent to select actions in order to generate its own data to learn about the environment? I shed light on both these questions in the following Theorem, which is very closely related to Theorem 2 of Sun et al., 2019. Again, I state the theorem informally here, and formally with proof in Appendix A.1.

Theorem 3.2. *Consider a class of deterministic, episodic, MDPs $\mathcal{M}$ with fixed reward function, and transition function belonging to some known hypothesis class. Assume an agent is able to interact with the MDP in an online fashion selecting actions at each time step. Let D_t be the dataset of all transitions observed up to time t during this interaction. Let $H_B(D_t)$ be the subclass of action-value functions in H_Q which obey the Bellman optimality equation for the transitions in D_t and let $H_M(D_t)$ be the subclass of optimal action-value functions of MDPs in $\mathcal{M}$ which are consistent with D_t. For any $\delta \in (0, 1]$ and $N \in \mathbb{N}$, there exists $\mathcal{M}$ such that the following are true:*

1. *For any agent which selects actions at each time t based only on $H_B(D_t)$ and the current state S_t, there is some MDP in $\mathcal{M}$ such that with probability at least $1 - \delta$ the return for all episodes up to episode N will be* 0.
2. *There exists an agent which selects actions at each time t based on $H_M(D_t)$ and S_t such that the return for all episodes after the first is guaranteed to be* 1*, which is optimal.*

Theorem 3.2 highlights that learning a model doesn't only allow us to narrow down the possible optimal action value functions faster; in addition, forcing an agent to make decisions based only on the value information that can be determined using Bellman consistency can mean it takes an arbitrary factor more samples to achieve reasonable performance.

The remainder of this chapter focuses on demonstrating empirically how

the intuition underlying these theorems is relevant to standard RL algorithms with simple neural-network function approximation. In the next section, I will highlight a simple and intuitive case where similar intuition leads a learned model to have a clear empirical benefit despite the fact that I do not encode any problem-specific structure into the model.

3.3 A Simple Case where Model-Based Generalization is Useful

There have been many examples of empirically successful model-based approaches recently (e.g. Hafner et al. (2021) and Schrittwieser et al. (2020)). However, owing to the many design choices involved in such algorithms, it can be hard to establish where the benefits are actually coming from. In this section, I present an illustrative example where learning a parametric model from data, and then learning an action-value function within that model, has a clear advantage over learning an action-value function directly from the data. This example will also explore another gap between Theorem 3.1 and model-based RL practice. Namely, in practice, we don't compute the exact value function under the learned model. Instead, one common strategy is to train a value function on model-generated rollouts initialized from states in the ER buffer. The example presented here shows straightforwardly how, while even single-step rollouts can be useful, using multi-step model rollouts can provide an additional advantage. To keep this example simple and intuitive, I consider an offline RL setting with hand-selected datasets with varying coverage of the dynamics.

The environment in this section consists of a 3x3 grid world with a goal in the top left corner and arbitrary walls (the location of which is fixed within an episode). Reward is -1 at each step until the goal is reached, at which point the episode terminates. Actions consist of standing still or moving in a cardinal direction. Observations are flat binary vectors consisting of one-hot-encodings of the agent's position and the goal position (though goal position is fixed here), and two binary vectors indicating, respectively, the presence and

absence of walls in each cell.

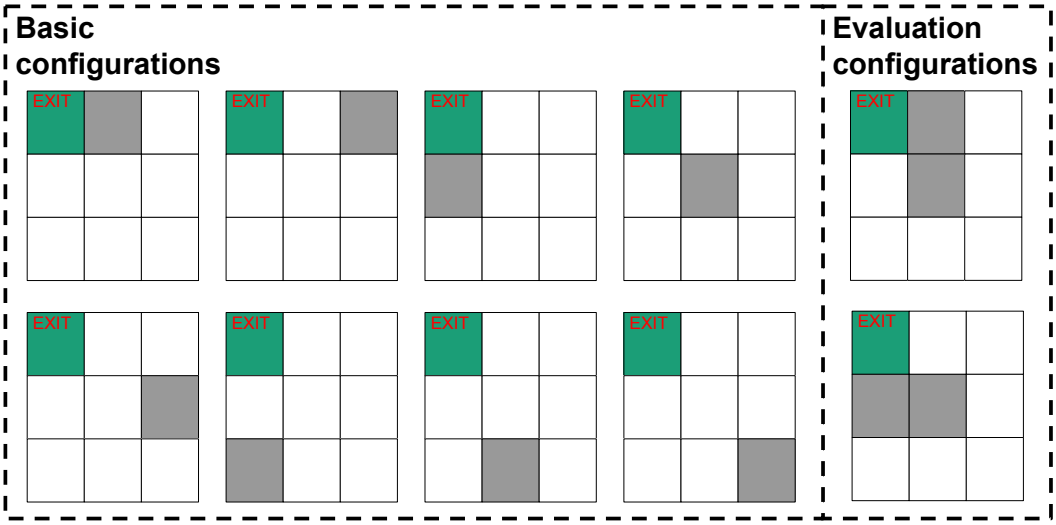

Figure 3.3: Maze layouts included in the basic and evaluation sets. All datasets include every transition in each of the basic configurations, but different sets of transitions in the evaluation configurations.

For explanatory purposes, I break the data in the training datasets provided to the agents into a *basic set* and an *evaluation set*. The basic set consists of all possible transitions with wall layouts having a single wall within one of the 8 non-goal cells. Every training dataset contains this entire basic set, in addition to some limited data from the evaluation set. The evaluation set consists of transitions with wall layouts of 2 walls in one of the configurations illustrated in Figure 3.3. I consider training datasets with 4 different levels of evaluation set data coverage:

- **All-Evaluation**: All possible transitions within the evaluation wall layouts.
- **Path-to-Goal**: Only evaluation layout transitions which follow the path to the goal.
- **Single-Cell**: Only the transitions starting from the cell furthest from the goal (with respect to the only open path).
- **No-Evaluation**: No transitions from the evaluation set.

I compare model-based and model-free algorithms in this setting. Both approaches use DQN for behaviour learning. The model-free approach trains

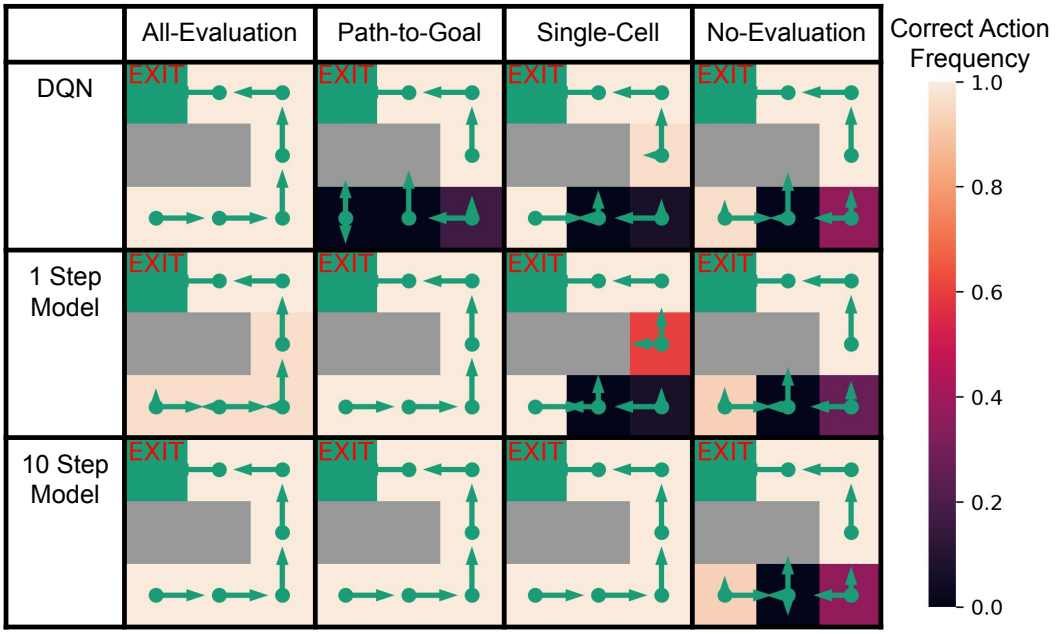

Figure 3.4: Frequency (over 30 random seeds) of trained agents' greedy policy selecting the correct action in cells of the displayed maze with different evaluation set coverage during training. The length of the green arrows indicates the frequency of greedy policies picking the corresponding action.

DQN with examples from the dataset. For the model-based approach, I use a simple feedforward neural-network model and train DQN on model-generated transitions. The model takes an observation as input and outputs a predicted reward, Bernoulli termination probability and vector of Bernoulli probabilities that each feature is active in the next state. This model is sufficient to represent the true dynamics since the environment has deterministic transitions. I train the model on transitions from the dataset and train the action-value function on model rollouts initialized from states in the dataset. I trained one model-based agent with single-step rollouts and another with 10-step rollouts. Each agent is trained for 1 million training steps with the total number of (real or imagined) transitions used in each DQN update held fixed across all agents. See Appendix A.5 for further details on the experiment setup.

Each agent is evaluated by checking the frequency out of 30 independent runs with which the greedy action under the learned value function is optimal in each cell of an evaluation layout. I say the agent has failed if there exists any cell in which the majority of runs select the wrong greedy action. I next

discuss how I expect each agent to perform with each level of evaluation set coverage.

All-Evaluation: I expect that all agents will succeed in the case where data is given for all transitions in the evaluation layouts. Here, a model-free agent has access to all the data needed to back up value from the goal and determine the value of each state-action pair, and a model-based agent has no opportunity to generate useful novel transitions which do not already appear in the dataset.

Path-to-Goal: I expect model-free DQN to fail in the Path-to-Goal case. Consider the lower evaluation wall layout in Figure 3.3. For the bottom middle cell, there is no data in the Path-to-Goal dataset for actions besides moving right. For every basic-set layout, moving right is worse than moving up. I predict the model-free agent will incorrectly generalize from the basic set to conclude that the right action is also worse in this new configuration.

I expect both model-based agents to succeed with Path-to-Goal data. All missing transitions are one step away from the available data, but require a different action selection. I predict that the outcome of these missing actions can be learned by generalization from the basic set. To determine the effect of an action, it suffices to look at the agent's current location and whether there is a wall in the cell it is attempting to enter. I predict the agent will be able to learn this basic structure from the training data, even in the absence of specific data about the case where there are two walls. Note that this hypothesis implies a nontrivial prediction about how this simple model will generalize. Factored structure is not hard-coded in the model, thus it is also plausible that the model predictions will be arbitrarily bad for the unobserved evaluation transitions.

Single-cell: I expect the single-step model to fail when only transitions from a single evaluation cell are included in the dataset. In this case, the model rollouts have no chance to reconstruct anything not already available explicitly in the dataset given all single-step transitions from the far cell are included and this is likely to be insufficient for reasons already explained. However, the 10-step model has the potential to succeed. If the model generalizes as

expected, it can start in the far cell available in the dataset and roll out a trajectory which discovers the full path to the goal.

No-Evaluation: Finally, all agents should fail in the case where there is no evaluation data available at all. The models will have no opportunity to provide the learned value function with example transitions from the evaluation layout given model rollouts are initialized with states from the dataset. Note that the situation may be different for an agent using a generative start-state model to produce plausible states for the start of rollouts which need not explicitly appear in the dataset.[4] Using the model to plan for immediate action selection at evaluation time, as in decision-time planning, would also help here as the agent could directly plan a response for the previously unseen state.

In Figure 3.4, we observe that all the above predictions are confirmed. I reiterate that this is a nontrivial empirical result. It relies on the simple model, a feedforward neural network, with no explicit bias toward factored solutions, generalizing in a particular way to state-action pairs that do not explicitly appear in the dataset. At least in this case, the model indeed seems to generalize in a way that provides a significant advantage over ER alone.[5] This experiment also straightforwardly illustrates how even a model with 1-step rollouts can be helpful, by sampling counterfactual actions or, in the case of stochastic environments, counterfactual chance outcomes. However, multi-step rollouts can succeed in situations where there is insufficient data for one-step rollouts to be helpful.

[4]This raises the question of how the start-state model should generalize. If trained to maximize data likelihood, it could overfit and only generate states from the training set.

[5]I also verified that the model predictions are near perfect with respect to predicting the agent position after each state-action pair across all transitions in the evaluation configurations.

3.4 Favourable Environments for Online Model-Based Learning

I next describe three environments which exemplify properties that should make online learning with a parametric model particularly useful. I aim for the following environment characteristics:

1. Simple factored structure in the state space that I expect should be easy to learn for a model with reasonable generalization properties.
2. Return which depends sharply on the policy, such that a randomly behaving agent won't have much of a learning signal for policy improvement. This should make model-generated experience more useful, as Bellman backups alone will tend to be mostly uninformative.
3. Occasional random transitions to rewarding states (or terminal states when termination is desirable). This allows the model-based agent to learn about the reward function even while behaving highly suboptimally.

The third characteristic does not contradict the second as an agent can occasionally obtain some reward while behaving suboptimally but have difficulty obtaining more. This characteristic may seem contrived, but I argue that it is quite natural. For example, one can imagine an agent gathering edible plants for a long time before working out how to grow their own. It is often much harder to discover reward, without ever observing it, than to work out how to reconstruct rewarding circumstances using general knowledge of the transition dynamics. On a practical note, there is significant work on exploration with sparse (or no) reward (Amin et al., 2021; Schmidhuber, 1991a, 1991b, 2010; Thrun, 1992) which is orthogonal to my focus here. Hence, I mitigate the issue by making it easier to learn the reward function. I ablate these spontaneous transitions to rewarding states in Appendix A.9 to test the impact of this choice.

In addition to the above, the environments I investigate allow scaling of problem complexity to test the limitations of different approaches. The environments are also Markov, use binary features, and are largely deterministic,

so simple models can work well, though I also investigate more sophisticated latent-variable models. Next, I describe the environments (see Appendix A.2 for details).

ProcMaze (Figure 3.5, left): Procedurally generated grid world mazes. The maze itself, along with the start state and goal state, is randomized in each episode. Negative reward is given for each step until the goal is reached. Complexity is scaled by increasing the grid size.

ButtonGrid (Figure 3.5, middle): A 5 by 5 grid with randomly placed buttons. An agent can move around and, if it hits a button, will toggle it on or off. If all buttons are on, a reward is given and button locations are randomized. Random behaviour will tend to randomly perturb the buttons, a precise policy is required to set them all to on. Complexity is scaled by increasing the number of buttons.

PanFlute (Figure 3.5, right): A minimal example of an environment with combinatorial complexity of optimal behaviour, but simple factored transition structure. PanFlute consists of n *pipes* of cells where each pipe evolves independently. Each action directly activates the cell at the bottom of one pipe, after which the activation will propagate up the pipe, one step at a time, and dissipate after reaching the end cell of the pipe. A reward is received if the cells at the end of all pipes are simultaneously active, which can only be achieved by choosing each of the n actions in a certain order, a probability of $1/n^n$ under random behaviour. Complexity is scaled by increasing the number of pipes n.

I expect ER alone to be of limited utility in each of these environments as each of them requires precise control to obtain significantly more reward than random behaviour, especially as the problem complexity is scaled up in each case. Since each environment includes random transitions to rewarding states, an agent can easily learn that these states are good, but until it reaches the rewarding state by its own actions, it won't be able to learn much from the states which precede rewarding states.

On the other hand, each environment has factored structure that a model-based agent can learn, and subsequently use to imagine many novel, plausible,

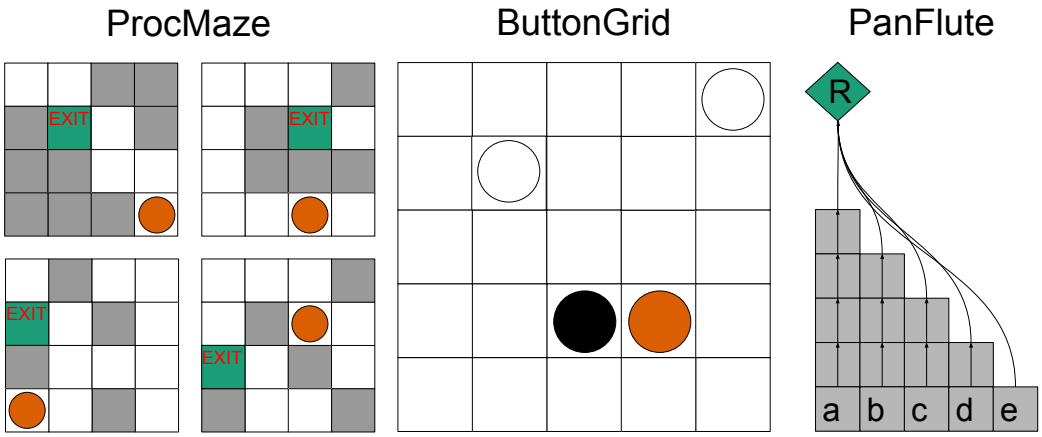

Figure 3.5: **Left**: Examples of states in the ProcMaze environment of size 4 with the agent shown in orange. **Middle**: An example state of the ButtonGrid environment with 3 buttons. The agent is shown in orange and the buttons in black (off) and white (on). **Right**: An instance of the PanFlute environment with 5 pipes. The agent directly activates cells (a,b,c,d,e) through its actions after which the activation propagates up the associated pipe, one step at a time, and dissipates at the end.

states in its rollouts. In ProcMaze, an agent moving into a specific empty space will have the same effect regardless of the rest of the maze layout, and attempting to move into a wall will always block it. In ButtonGrid, the connectivity of the grid is independent of the button layout, and stepping on a button in a specific cell will have the same effect regardless of the layout of the rest of the buttons. In PanFlute, each action always has the same effect, and each pipe evolves according to dynamics which are unaffected by the other pipes.

To better contextualize the results for environments with factored structure, I will also present results in an open grid world with a goal in one corner which I refer to as OpenGrid. The agent location is simply represented by a one-hot vector (effectively tabular) so there is really no structure to exploit in OpenGrid. The learned model must essentially memorize every individual transition to learn the dynamics. Further details of this unstructured environment are available in Appendix A.3.

3.5 Beneficial Model-Based Generalization for Online RL

I now empirically evaluate the performance of model-based and model-free learning algorithms on the environments described in Section 3.4. I experiment with variants of each environment with a range of complexities to test how different approaches scale to more complex environments.

All tested approaches use DQN for behaviour learning but vary in the source of training examples for DQN. The model-free approach draws transitions randomly from an ER buffer for training. I test several types of learned model. The first is the simple feedforward neural network model introduced in Section 3.3. The second is a latent-variable model (Ha et al., 2018; Schmidhuber, 1997; Watter et al., 2015) inspired by Dreamer (Hafner et al., 2020; Hafner et al., 2021), but with two major differences to simplify the approach and reduce confounding factors in my experiments. In particular, I use DQN instead of actor-critic and, since the considered environments are Markov, I forgo the recurrent network of Dreamer and model single-step stochastic transitions with no memory. I experiment with Gaussian-latent and categorical-latent variables. See Appendix A.4 for further details of these models. Finally, as a strong baseline, I include a perfect model, which uses the ground truth environment dynamics but is otherwise the same as the simple-model agent.

As in Section 3.3, I control for the total number of updates, and the total number of (real or imagined) transitions used in each update. In particular, model-free DQN is updated on a batch of 320 transitions from the ER buffer while all model-based approaches use 32 model rollouts of length 10, beginning in a state from the ER buffer. In Appendix A.9, I perform an ablation in which 1-step rollouts are used instead and find that longer rollouts are generally helpful. In each update, the model is trained using the same batch of transitions which initialize the rollouts. All agents use a softmax behaviour policy and are evaluated under the greedy policy. Action-value learning uses Equation 2.2, with real or imagined transitions, using a discount factor of 0.9.

I experiment with 2 different data regimes to get a more complete picture

of how each approach scales with available data. In the *high-data* regime, I use one update per real environment step and train for a total of 1 million steps. In the *low-data* regime, I use 10 updates per step and train for 100 thousand steps. Note that the total number of updates is the same in each case.

Experiment Design: For each combination of agent, environment, and data regime I performed an extensive grid search over the action-value-function step size and softmax-exploration temperature. I judged these hyperparameters to be the most likely to impact the relative performance of different methods. This grid search was performed for an intermediately complex version of each environment (size 4 ProcMaze, 4 button ButtonGrid, and 7 pipe PanFlute) and the same hyperparameters were used for the other complexity levels. I evaluated each hyperparameter setting based on mean final performance of the greedy policy over 30 random seeds. I was able to run 30 seeds efficiently in parallel on a single GPU using automatic batching in JAX (Bradbury et al., 2018). Other hyperparameters, including model step size, were fixed to reasonable defaults (see Appendix A.6), not tuned for any specific approach. In Appendix A.7, I report hyperparameter sensitivity results from this grid search.

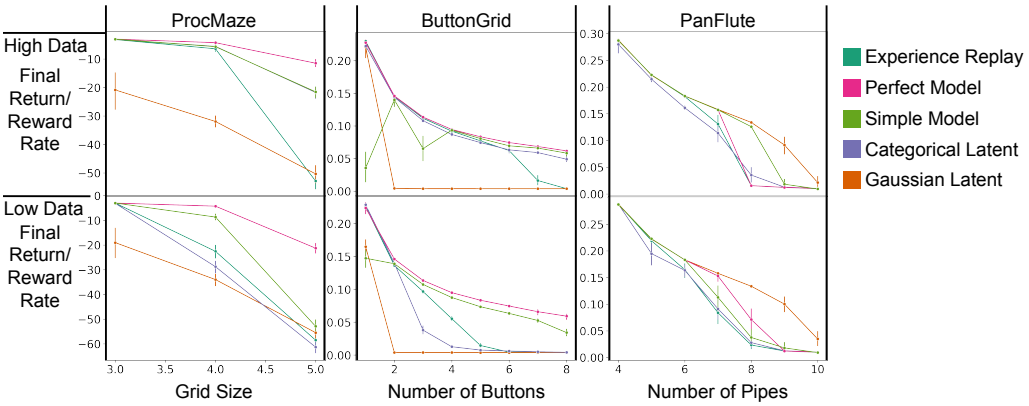

Figure 3.6: Final performance of greedy policy for the three structured environments in two different data regimes. Error bars show 95% confidence intervals.

Results: I present results for the three structured environments in Figure 3.6 (see Appendix A.8 for learning curves). As the results for PanFlute differ substantially from the results for ButtonGrid and ProcMaze, I will dis-

cuss them separately. For ButtonGrid and ProcMaze in the high-data regime, the simple model and categorical-latent model both significantly outperform ER for sufficiently complex environment instances. The Gaussian-latent model generally performs quite poorly, which corroborates the results of Hafner et al. (2021) that categorical latents tend to work better in the discrete control setting. Oddly, the simple model also performs much worse for 1 and 3 buttons in ButtonGrid in the high-data regime. This may be because fewer buttons mean fewer examples where a button occupies each particular cell, which makes it harder for the simple model to learn the underlying dynamics.

In the low-data regime, the simple model outperforms ER to a greater extent, while the performance of the categorical-latent model degrades significantly. This can perhaps be understood by noting that the hypothesis class of the simple model is simpler (the latent-variable model can model correlated features, while the simple model cannot) and thus is able to generalize well from less data when the simple class is sufficient. Performance of the simple model for 1 and 3 buttons improves when moving to the low-data regime. Likely, this indicates underfitting in the high-data regime which is helped by training more on each example. Overall, the results for the simple model and categorical-latent model in the high-data regime, and the simple model in the low-data regime, show a clear indication of the sample efficiency benefit that can be obtained by using a learned model in these environments.

Results for PanFlute are qualitatively different. Most surprisingly, the Gaussian-latent model and the simple model outperform the perfect model in some of the harder problem instances. This is intuitively strange and seems to indicate that model errors somehow improve performance.

Why do Some Learned Models Outperform the Perfect Model on PanFlute? I hypothesize that the smoother reward and dynamics learned by the model serve as a powerful exploration heuristic. The true reward function is nonzero, and thus the agent receives a learning signal, only in the rare event that every pipe-end is active. The model might instead learn a smoothed reward function, proportional to the number of pipe-ends activated. The agent could then learn incrementally to activate more pipe-ends, gradually improving

toward the correct sequence.

To test this hypothesis, I looked at the models learned at 10,000 time steps, in 9-pipe PanFlute (high data). I generated a large amount of random trajectory data with a policy that selects actions in alphabetical order with 80% probability and uniformly randomly otherwise to get a good mix of different numbers of active pipe-ends. I bin this data by the number of active pipe-ends and then look at the average model-predicted reward for each bin. Note that the ground truth reward is zero for all except the 9 active pipe-end bin. To test for favourable smoothing in the transition dynamics, I used the same data. This time, I bin the data by the number of pipe-ends active at the *next* step and look at the probability under the model that all pipe-ends are active at the next time step.

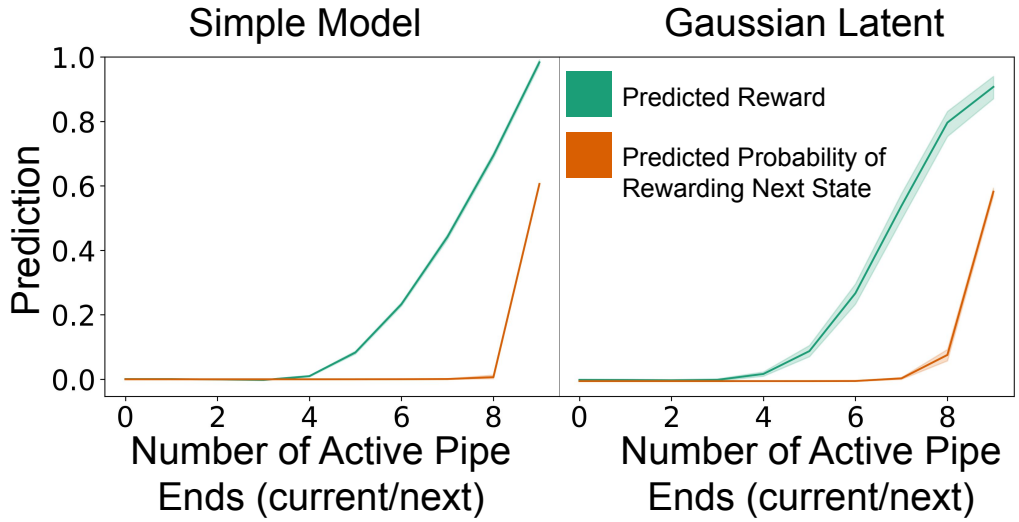

Figure 3.7: Model prediction of reward and probability that all pipe-ends are active at the next time step as a function of the true number of current and next pipe-ends active respectively on 9-pipe PanFlute. Error bars show 95% confidence intervals.

The results, for both predicted reward and predicted probability that all pipe-ends will be active at the next time step ("predicted probability of rewarding next state" in the figure) are displayed in Figure 3.7. I observe that the models indeed tend to learn smoother rewards than the ground truth in a way that might provide a useful exploration heuristic. The Gaussian-latent

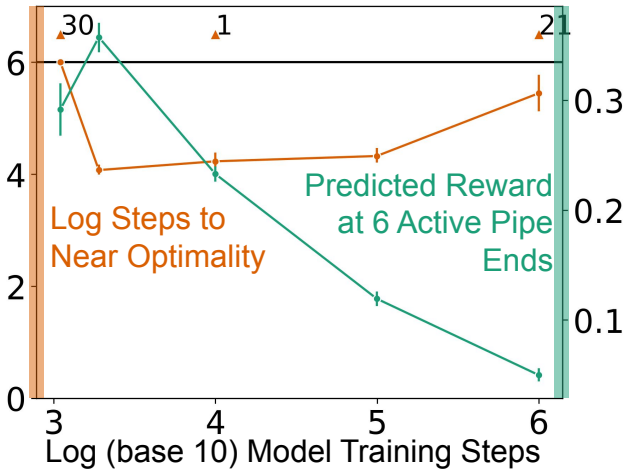

Figure 3.8: In orange, the number of steps to reach near-optimal performance (95% of maximum possible reward rate) on 9-pipe PanFlute when using a frozen simple model trained for a variable number of steps. Numbered arrows indicate the number of seeds out of 30 which failed to reach near-optimal performance within 1 million steps. In green, the predicted reward for 6 active pipe-ends, displayed as a surrogate for the amount of model smoothing. Error bars show 95% confidence intervals.

model additionally learns smoothed transition dynamics, predicting all pipe-ends will activate with appreciable probability when in reality only most will, this may provide additional benefit in this problem.

As an additional test of the benefit of model errors, I used simple models frozen at various points in training to train a value function from scratch for 1,000,000 time steps. I plot when near-optimal performance of the greedy policy is first reached. The results, shown in Figure 3.8 clearly show that a model trained for an intermediate amount of time is most useful for reaching good performance quickly. I also plot the mean predicted reward for states with 6 active pipe-ends for each model as an indication of smoothing, as expected, this decreases with more model training.

The smoothing effect highlighted in these results is interesting for two reasons. First, it may lead model-based algorithms to perform better than expected, acting as a confounding variable when interpreting results. Second, it may be genuinely useful. One could even design algorithms which learn policies within relaxed versions of a model, as a way to drive exploration. However, as there are surely other situations where model smoothing is harmful, this

would need to be done with care.

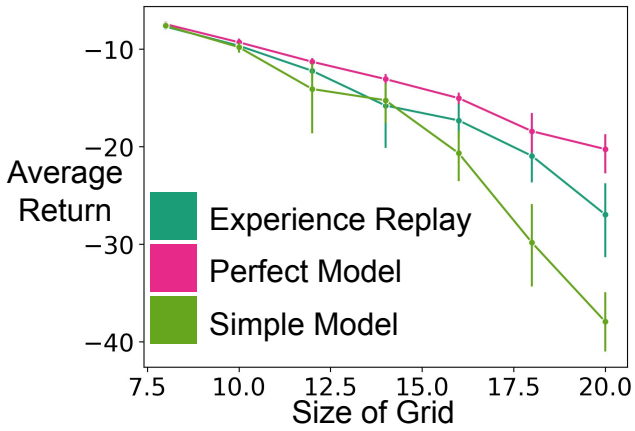

Figure 3.9: Final performance of greedy policy for OpenGrid in low-data regime. Error bars show 95% confidence intervals.

What Happens when There is no Structure for the Learned Model to Exploit? We can compare the above, largely positive, results for the benefit of model-based generalization with the results for OpenGrid shown in Figure 3.9. In contrast to the more favourable environments, here we see that the simple model becomes worse relative to ER as the environment complexity increases. This is reasonable as OpenGrid is essentially tabular and thus the model has no ability to extrapolate beyond the data. The best it can do is memorize the transitions that are already in the ER buffer and the limitations of finite model capacity and imperfect optimization prevent it from doing so perfectly. See Appendix A.3 for further details and parameter sensitivity curves for OpenGrid.

3.6 The Drawbacks of Model-Based Learning

One clear limitation of Theorem 3.1 is that we assume the model and value function are effectively constructed using the same initial knowledge of the world. When this is not the case, it is obvious that if a particular value-function class is a better fit to a problem class of interest than a particular model class, we may be better off using a model-free method with the superior function class. However, I see this limitation as somewhat uninteresting since the point

of the analysis is to highlight how a model-based method can allow us to make inherently better use of problem structure to extract more information from the data.[6] More interesting is to consider in what sense a model-free method might be preferred even when we assume parity of world knowledge.

If realizability does not hold, or if the optimization process of the world model fails to find the best model, despite it existing in the function class, then model-free learning may be preferred, even when the model and value function share analogous function classes. In particular, by fitting the value function directly, model-free learning can better use limited function approximate resources given sufficient data. In the extreme case where one assumes that we have enough data such that the empirical MDP is effectively a perfect model, then it's probably preferable to fit a value function to that MDP than to the imperfect learned model. If we ultimately care about the accuracy of our value function, then optimizing for value-function accuracy directly avoids an extra step of unnecessary error propagation.

In practice, there will usually be a trade-off in choosing a model-based method over a model-free method with ER. Learning a model can extract more information from limited data. However, if the model or its optimization process cannot perfectly fit the data, a value-based method will be preferable when there is sufficient data that the empirical MDP becomes a better model than the learned one. One could also employ hybrid methods where the value function is trained using a mix of real and model-generated transitions to get some of the benefits of both. Nevertheless, the results presented here give good reason to believe that model-based methods are preferable given sufficient computation and function approximation resources.

[6]Though there is a related question of whether it is somehow practically easier to encode useful structure into a value function than a model, and if so what could be done to change that?

3.7 Discussion

This chapter introduced the first major contribution of this thesis, and the first example of how our design decisions can influence a reinforcement agent's ability to take advantage of generic problem structure. In particular, I demonstrated that model-based agents are in a sense inherently better able to take advantage of known structure. Theorem 3.1 demonstrates that for a model class with some known structure, we can generally narrow down the possible value functions more with a model-based approach than a model-free approach. I also provided empirical evidence that the intuition behind this theorem holds in practice when we use neural network function approximation in domains with factored structure. In such cases, I have verified through extensive experiments that a model-based method can maintain strong performance as the complexity of the environment increases beyond the point where an analogous model-free approach fails. As an aside I demonstrated how, by smoothing the reward and/or transition dynamics, experience generated by a learned model can provide a useful signal for exploration that can sometimes lead to better performance than even a perfect model. Overall, I believe that the work presented in this chapter can help to ground future work in model-based RL in a better understanding of how learned models can improve sample efficiency. An interesting direction for future work lies in better understanding the inductive biases that allow simple neural network models to generalize in a way that allows them to efficiently learn factored structure, and how more sophisticated architectures could improve on this.

Chapter 4

Hindsight Network Credit Assignment: Efficient Credit Assignment in Networks of Discrete Stochastic Units

The central focus of this thesis is on demonstrating how we can design RL agents which take advantage of generic problem structure. In the previous chapter, I demonstrated how model-based RL algorithms can do an inherently better job of exploiting known problem structure than model-free algorithms. This chapter introduces the second major contribution of this thesis and the second example of how known problem structure can be exploited to develop better algorithms. Relative to the previous chapter which provides a fairly general comparison of two broad classes of algorithms, the present chapter provides a more specific example of how known problem structure can be exploited. In particular, I consider the challenging problem of training a network of stochastic units, each of which can be considered an agent trying to optimize a shared objective. I provide a novel algorithm which demonstrates how, by using our knowledge of the network structure, we can achieve a drastic performance improvement compared to a more general algorithm that does not exploit this structure.

Using discrete stochastic units within neural networks is appealing for several reasons, including representing multimodal distributions, modelling discrete choices, providing regularization and facilitating exploration. However,

training such units efficiently and accurately presents challenges, as backpropagation is not directly applicable, nor are the reparameterization tricks (Kingma & Welling, 2014; Rezende et al., 2014) that are typically used with continuous stochastic units. Despite these challenges, discrete stochastic units have played an important role in recent empirical successes in both text-to-image generation (Ramesh et al., 2020) and model-based RL (Hafner et al., 2021). Hence, techniques for efficiently training networks of discrete stochastic units have the potential to be of significant practical interest.

In this chapter, I introduce an unbiased, and computationally efficient estimator for the gradients of stochastic units which provably reduces gradient estimate variance compared to REINFORCE. This estimator works by assigning credit to each unit based on how much it impacts the outputs of its immediate children. My approach is inspired by Hindsight Credit Assignment (HCA; Harutyunyan, Dabney, Mesnard, et al., 2019) for RL, hence I call it Hindsight Network Credit Assignment (HNCA).

In addition to the immediate application to stochastic neural networks, I believe the insights presented in this chapter can help pave the way for new ways of thinking about efficient credit assignment in stochastic compute graphs, including perhaps the RL setting.

4.1 Related Work

As already mentioned, HNCA is inspired by HCA. In particular, HNCA was initially motivated by the idea of assigning credit to possible actions based on the estimated probability of each action conditioned on some information about the future, instead of the single action that was actually selected. This basic motivation is shared with HCA, however, the way this idea is instantiated in HNCA is quite different.

More directly related to HNCA is the literature on estimating gradients in networks of discrete stochastic units. Prior work has proposed a number of techniques for producing either biased or unbiased estimates in such networks. Bengio et al. (2013) propose an unbiased REINFORCE (Williams,

1992) style estimator, as well as a biased but low variance estimator which replaces a random variable with its expectation during backpropagation. Tang et al. (2013) propose an expectation-maximization procedure which maximizes a variational lower bound on the loss. Mnih et al. (2014) propose several techniques to reduce the variance of a REINFORCE style estimator, including subtracting a learned baseline and normalizing by a moving average standard deviation. Maddison et al. (2017) and Jang et al. (2017) each propose a biased estimator based on a continuous relaxation of discrete outputs. Tucker et al. (2017) use such a continuous relaxation to derive a control variate for a REINFORCE style estimator, resulting in a variance-reduced *unbiased* gradient estimator. Grathwohl et al. (2018) and Gu et al. (2018) also explore the use of control variates with discrete random variables. Yin et al. (2019) provide a variance-reduced unbiased estimator, called ARM, based on a particular reparameterization and antithetic sampling. Dong, Mnih, et al. (2020) further reduce the variance of ARM by marginalizing over the reparameterization step.

Perhaps the most closely related work is the local expectation gradients (LEG) approach of Titsias et al. (2015). In fact, the gradient estimator used in HNCA can be seen as an instance of the LEG estimator. However, the generic expression for the LEG estimator makes it unclear when and how it can be efficiently computed. This has led to suggestions in the literature that LEG tends to be too computationally expensive to be practical (Mnih & Rezende, 2016; Tucker et al., 2017).

This chapter extends the work of Titsias et al. (2015) in several ways. First, while LEG may be computationally expensive in the general case, for the common case of a network of Bernoulli units, with firing probability parameterized by a linear transformation of their inputs followed by a nonlinear activation, HNCA uses an efficient message-passing procedure.[1] In this case, the resulting computational cost is similar to that of Backpropagation. This efficiency allows us to straightforwardly apply HNCA to multi-layer Bernoulli networks, while the analysis and experiments of Titsias et al. (2015) focus

[1] A similar procedure applies to units with softmax activation, though I do not explore this empirically in this thesis.

on single-layer (fully factorized) stochastic networks. I further demonstrate that a simple baseline subtraction, similar to that employed by Mnih et al. (2014), drastically improves performance when applying HNCA to multi-layer networks. While Titsias et al. (2015) focus on the case where the agent has access to the function being optimized, I also present HNCA in a contextual bandit setting where an agent operates online, outputting an action at each time step and observing a single sampled reward as a result. Interestingly, in the contextual bandit setting, we can still compute local expectations for each hidden unit without the need to resample the reward. Finally, I prove that HNCA provides a variance reduction over REINFORCE.

In taking inspiration from RL to train networks of stochastic units, HNCA is related to work on CoAgent Networks (Kostas et al., 2020; Thomas et al., 2011) that formalizes framing stochastic networks as collectives of interacting RL agents.

4.2 HNCA in a Contextual Bandit Setting

I first formulate HNCA in a contextual bandit setting. In this setting, an agent interacts with an environment in a series of time steps.[2] At each time step, the environment provides an i.i.d. random context $X \in \mathcal{X}$ (for example the pixels of an image). The agent then selects an action from a discrete set $A \in \mathcal{A}$ (for example a guess of what class the image belongs to). Finally, the environment responds with a real-valued reward $R \sim \tilde{r}(X, A)$, where $\tilde{r}$ is an unknown reward distribution (for example a reward of 1 for guessing the correct class and 0 otherwise). The agent's goal is to select actions which result in as much reward as possible.

I will consider an agent which consists of a network of stochastic computational units. Let Φ be a random variable corresponding to the output of a particular unit. For each unit, Φ is drawn from a parameterized *policy* $\pi_\Phi(\phi|b) \doteq \mathbb{P}(\Phi = \phi | \operatorname{pa}(\Phi) = b)$ conditioned on $\operatorname{pa}(\Phi) = b$, its parents in the

[2] As it will not be necessary to distinguish multiple time steps in this chapter, I suppress the time step in notation, for example writing the context as X instead of X_t. All random variables are defined for a single arbitrary time step.

network.[3] Each unit's policy is differentiably parameterized by a unique set of parameters $\theta_\Phi \in \mathbb{R}^d$. A unit's parents $\text{pa}(\Phi)$ may include the output of other units, as well as the context X. I focus on the case where Φ takes values from a discrete set. I will use $\text{ch}(\Phi)$ to refer to the children of Φ, that is, the set of outputs of all units for which Φ is an input.[4] I assume the network has a single output unit, which selects the action A sent to the environment.

The goal is to tune the network parameters to increase $\mathbb{E}[R]$. Towards this, I will construct an unbiased estimator of the gradient $\frac{\partial \mathbb{E}[R]}{\partial \theta_\Phi}$ for the parameters of each unit, and update the parameters according to the estimator.

Directly computing the gradient of the output probability with respect to the parameters for a given input, as we might do with backpropagation for a deterministic network, is generally intractable for discrete stochastic networks. Instead, we can define a local REINFORCE estimator, $\hat{G}^{\text{RE}}_\Phi \doteq \frac{\partial \log(\pi_\Phi(\Phi|\,\text{pa}(\Phi)))}{\partial \theta_\Phi} R$. It is well known that $\hat{G}^{\text{RE}}_\Phi$ is an unbiased estimator of $\frac{\partial \mathbb{E}[R]}{\partial \theta_\Phi}$ (see Appendix B.1 for a proof). However, $\hat{G}^{\text{RE}}_\Phi$ tends to have high variance.

One can think of the estimator $\hat{G}^{\text{RE}}_\Phi$ as treating each unit in the network as a separate sub-agent running REINFORCE to optimize its own policy. While they all optimize the same reward, the units have no awareness of their position in the larger network. The only information each unit receives is its own input and the reward. Based on this feedback alone they optimize their own policy to favour outputs that tend to result in higher reward. The random decisions of all the other units in the network are observed only as noise in the input and reward signal.

HNCA Gradient Estimator: HNCA exploits the causal structure of the network to assign credit to each unit's output based on how it impacts the output of its immediate children. Assume Φ is a nonoutput unit and define $\text{mb}(\Phi) \doteq \{\text{ch}(\Phi), \text{pa}(\Phi), \text{pa}(\text{ch}(\Phi)) \setminus \Phi\}$ as a notational shorthand. Note that $\text{mb}(\Phi)$ is a Markov blanket (Pearl, 1988) for Φ, meaning that conditioned on

[3] Expectations and probabilities are taken with respect to all random variables in the network, and the context.

[4] I may also apply $\text{ch}(\cdot)$ or $\text{pa}(\cdot)$ to sets, in which case it has the obvious meaning of the union of the elementwise applications.

$\text{mb}(\Phi)$, Φ is independent of all other variables in the network as well as the reward R. Beginning from the expression for $\hat{G}_\Phi^{\text{RE}}$, we can rewrite $\frac{\partial \mathbb{E}[R]}{\partial \theta_\Phi}$ as follows:

$$\begin{aligned}
\frac{\partial \mathbb{E}[R]}{\partial \theta_\Phi} &\overset{(a)}{=} \mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\,\text{pa}(\Phi)))}{\partial \theta_\Phi} R\right] \\
&\overset{(b)}{=} \mathbb{E}\left[\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\,\text{pa}(\Phi)))}{\partial \theta_\Phi} R \middle| \text{mb}(\Phi), R\right]\right] \\
&\overset{(c)}{=} \mathbb{E}\left[\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\,\text{pa}(\Phi)))}{\partial \theta_\Phi} \middle| \text{mb}(\Phi)\right] R\right] \\
&\overset{(d)}{=} \mathbb{E}\left[\sum_\phi \frac{\mathbb{P}(\Phi=\phi|\,\text{mb}(\Phi))}{\pi_\Phi(\phi|\,\text{pa}(\Phi))} \frac{\partial \pi_\Phi(\phi|\,\text{pa}(\Phi))}{\partial \theta_\Phi} R\right],
\end{aligned} \tag{4.1}$$

where (a) follows from the unbiasedness of $\hat{G}^{\text{RE}}$, (b) applies the law of total expectation, (c) pulls R out of the expectation and then uses the fact that $\text{mb}(\Phi)$ forms a Markov blanket for Φ, thus we can drop the conditioning on R without loosing anything, and (d) expands the inner expectation over Φ and rewrites the log gradient. This idea of taking a local expectation conditioned on a Markov blanket is similar to the LEG estimator proposed by Titsias et al. (2015). However, it is not immediately obvious how to compute this estimator efficiently. Titsias et al. (2015) provide a more explicit expression and empirical results for a fully factorized variational distribution. Here, I will go beyond this case to provide a computationally efficient way to compute the inner expression for more general networks of stochastic units. To begin, I apply Theorem 1 from Chapter 4 of the probabilistic reasoning textbook of Pearl (1988), which implies that

$$\mathbb{P}(\Phi=\phi|\,\text{mb}(\Phi)) = \rho_\Phi(\phi)\pi_\Phi(\phi|\,\text{pa}(\Phi)). \tag{4.2}$$

where $\rho_\Phi(\phi) = \frac{\prod_{C\in\text{ch}(\Phi)} \pi_C(C|\,\text{pa}(C)\setminus\Phi,\Phi=\phi)}{\sum_{\phi'} \pi_\Phi(\phi'|\,\text{pa}(\Phi)) \prod_{C\in\text{ch}(\Phi)} \pi_C(C|\,\text{pa}(C)\setminus\Phi,\Phi=\phi')}$. Intuitively, $\rho_\Phi(\phi)$ is the relative counterfactual probability of the children of Φ taking the value they did had Φ been fixed to ϕ. See Appendix B.2 for a full proof. Substituting this result into the expression within the expectation in Equation 4.1, we get that the following is an unbiased estimator of $\frac{\partial \mathbb{E}[R]}{\partial \theta_\Phi}$:

$$\hat{G}_\Phi^{\text{HNCA}} \doteq \sum_\phi \rho_\Phi(\phi) \frac{\partial \pi_\Phi(\phi|\,\text{pa}(\Phi))}{\partial \theta_\Phi} R, \tag{4.3}$$

which I call the HNCA estimator. Equation 4.3 applies only to Φ for which $\mathrm{ch}(\Phi) \neq \emptyset$, which excludes the output unit A. In the bandit experiments, I use the REINFORCE estimator $\hat{G}_\Phi^{\mathrm{RE}}(\phi)$ for the output unit. Later, I will show how to improve upon this if we have access to the reward function.

HNCA assigns credit to a particular output choice ϕ based on the relative counterfactual probability of its children's outputs had ϕ been chosen, independent of the actual value of Φ. Intuitively, this reduces variance, because each potential output choice of a given unit will get credit proportional to the difference it makes further downstream. On the other hand, REINFORCE credits whatever output happens to be selected, whether it makes a difference or not. This intuition is formalized in the following theorem:

Theorem 4.1. $\mathbb{V}(\hat{G}_\Phi^{HNCA}) \leq \mathbb{V}(\hat{G}_\Phi^{RE})$*, where* $\mathbb{V}(\vec{X})$ *stand for the elementwise variance of random vector* $\vec{X}$*, and the inequality holds elementwise.*

Theorem 4.1 follows from the law of total variance by the proof available in Appendix B.3.

Efficient Implementation of HNCA: HNCA can be implemented as a message-passing procedure. A forward pass propagates information from parents to children to compute the network output. A backward pass passes information from children to parents to compute the HNCA estimator. The computational complexity of this procedure depends on how difficult it is to compute the numerators of $\rho_\Phi(\phi)$. We could naively recompute $\pi_C(C|\,\mathrm{pa}(C) \setminus \Phi, \Phi = \phi)$ from scratch for each possible ϕ. When C corresponds to a Bernoulli unit, which computes its output probability as a linear function of its inputs followed by sigmoid activation, this would require time $\mathcal{O}(|\,\mathrm{pa}(C)|N_\Phi)$, where N_Φ is the number of possible values Φ can take (2 if Φ is also Bernoulli). To do this for every parent of every unit in a Bernoulli network would thus require $\mathcal{O}(2\sum_\Phi |\,\mathrm{pa}(\Phi)|^2)$. This is much greater than the cost of a forward pass, which takes on the order of the total number of edges in the network, or $\mathcal{O}(\sum_\Phi |\,\mathrm{pa}(\Phi)|)$. This contrasts with backpropagation where the cost of the backward pass is on the same order as the forward pass, an appealing property, which implies that learning is not a bottleneck.

Algorithm 1 HNCA (Bernoulli unit)

1: Receive $\vec{x}$ from parents
2: $l = \vec{\theta} \cdot \vec{x} + b$
3: $p = \sigma(l)$
4: $\phi \sim Bernoulli(p)$
5: Pass ϕ to children
6: Receive $\vec{q_1}, \vec{q_0}, R$ from children
7: $q_1 = \prod_i \vec{q_1}[I]; \quad q_0 = \prod_i \vec{q_0}[i]$
8: $\bar{q} = pq_1 + (1-p)q_o$
9: $\vec{l_1} = l + \vec{\theta} \odot (1 - \vec{x}); \quad \vec{l_0} = l - \vec{\theta} \odot \vec{x}$
10: $\vec{p_1} = (1-\phi)(1-\sigma(\vec{l_1})) + \phi\sigma(\vec{l_1}); \quad \vec{p_0} = (1-\phi)(1-\sigma(\vec{l_0})) + \phi\sigma(\vec{l_0})$
11: Pass $\vec{p_1}, \vec{p_0}, R$ to parents
12: $\vec{\theta} = \vec{\theta} + \alpha\sigma'(l)\vec{x}\left(\frac{q_1 - q_0}{\bar{q}}\right) R$
13: $b = b + \alpha\sigma'(l)\left(\frac{q_1 - q_0}{\bar{q}}\right) R$

Algorithm 1: The forward pass in lines 1-5 takes input from the parents and uses it to compute the fire probability p and samples $\phi \in \{0, 1\}$. The backward pass receives two vectors of probabilities $\vec{q_1}$ and $\vec{q_0}$, each with one element for each child. Each element represents $\vec{q}_{0/1}[i] = \mathbb{P}\left(C_i | \text{pa}(C_i) \setminus \Phi, \Phi = 0/1\right)$ for a given child $C_i \in \text{ch}(\Phi)$. Line 7 takes the product of child probabilities to compute $\prod_i \pi_{C_i}(C_i | \,\text{pa}(C_i) \setminus \Phi, \Phi = 0/1)$. Line 8 computes the associated normalizing factor. Lines 9 and 10 use the logit l to efficiently compute a vector of probabilities $\vec{p_1}$ and $\vec{p_0}$. Each element corresponds to a counterfactual probability of ϕ if a given parent's value was fixed to 1 or 0. Here $\odot$ represents the elementwise product. Line 11 passes information to the unit's children. Lines 12 and 13 finally update the parameter using $\hat{G}_{\Phi}^{\text{HNCA}}$ with step-size hyperparameter α.

Luckily, we can improve on this for cases where $\pi_C(C | \,\text{pa}(C) \setminus \Phi, \Phi = \phi)$ can be computed from $\pi_C(C | \,\text{pa}(C))$ in less time than computing $\pi_C(C | \,\text{pa}(C))$ from scratch. This is indeed the case for linear Bernoulli units, for which the policy can be written $\pi_\Phi(1|\vec{x}) = \sigma(l_\Phi(1|\vec{x}))$ where $l_\Phi(1|\vec{x}) = \vec{\theta} \cdot \vec{x} + b$, $\vec{x}$ is the binary vector consisting of all parent outputs, b is a scalar bias, $\vec{\theta}$ is the parameter vector for the unit, and σ is the sigmoid function. Note that $\pi_\Phi(0|\vec{x})$ is simply $1 - \pi_\Phi(1|\vec{x})$. Now, say we wish to compute the counterfactual probability of $\Phi = 1$ given $\vec{x}[i] = 1$, and we already have $l_\Phi(1|\vec{x})$ from the forward pass. Regardless of the actual value of $\vec{x}_i$ we can use the following

identity:

$$\pi_\Phi(1|\vec{x} \setminus \vec{x}[i], \vec{x}[i] = 1) = \sigma(l_\Phi(1|\vec{x}) + \vec{\theta}[i](1 - \vec{x}[i])).$$

This requires only constant time, whereas computing $\pi_\Phi(\phi|\vec{x})$ requires time proportional to the length of $\vec{x}$. This simple idea is crucial for implementing HNCA efficiently. In this case, we can compute the numerator terms for every unit in a Bernoulli network in $\mathcal{O}(\sum_\Phi |\operatorname{pa}(\Phi)|)$ time. This is now on the same order as computing a forward pass through the network. Computing $\hat{G}_\Phi^{\text{HNCA}}$ for a given Φ from these numerator terms requires multiplying a scalar by a gradient vector with the same size as θ_Φ. For a Bernoulli unit, θ_Φ has $\mathcal{O}(|\operatorname{pa}(\Phi)|)$ elements, so this operation adds another $\mathcal{O}(\sum_\Phi |\operatorname{pa}(\Phi)|)$, maintaining the same order of complexity.

Algorithm 1 shows an efficient implementation of HNCA for Bernoulli units. Note that, for ease of illustration, the pseudocode is implemented for a single unit and a single training example at a time. In practice, I use a vectorized version which works with vectors of units that constitute a layer, and with minibatches of training data.

In Section 4.2, I will apply HNCA to a model consisting of a number of hidden layers of Bernoulli units followed by a softmax output layer. Appendix B.4 provides an implementation and discussion of HNCA for a softmax output unit. Note that the output unit itself uses the REINFORCE estimator in its update, as it has no children, which precludes the use of HNCA. Nonetheless, the output unit still needs to provide information to its parents, which do use HNCA. Using a softmax unit at the output, we can still maintain the property that the time required for the backward pass is on the same order as the time required for the forward pass. If, on the other hand, the entire network consisted of softmax nodes with N choices each, the HNCA backward pass would require a factor of N more computation than the forward pass, I discuss this in Appendix B.4 as well.

Contextual Bandit Experiments: I evaluate HNCA against REINFORCE in terms of gradient variance and performance on a contextual bandit

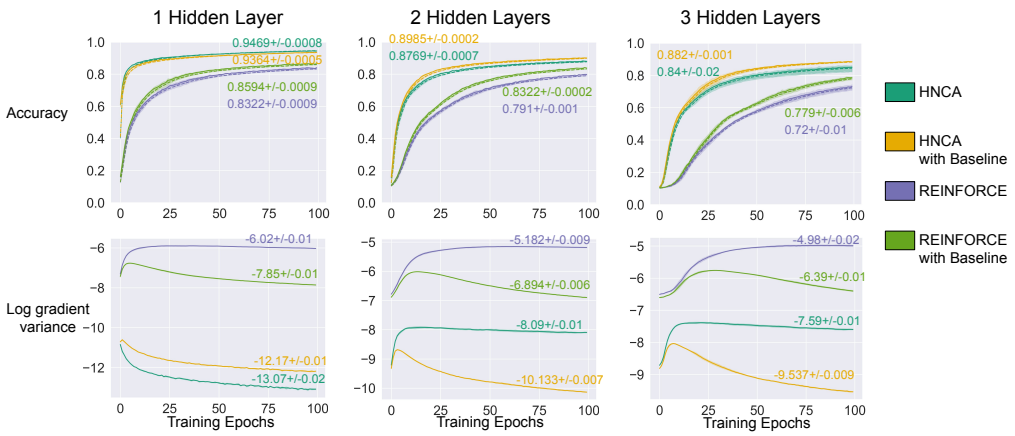

Figure 4.1: Training stochastic networks on a contextual bandit version of MNIST. Each line represents the average of 5 random seeds with error bars showing 95% confidence interval. Final values (train accuracy for the left plots) at the end of training are written beside each line. The left column shows the online training accuracy (or equivalently the average reward) as a dotted line, and the test accuracy as a solid line (though they essentially overlap). The right column shows the natural logarithm of the mean gradient variance. Mean gradient variance is computed as the mean of the per-parameter empirical variance over examples in a training batch of 50. I find that, for each network depth, HNCA drastically reduces gradient variance, resulting in significantly improved performance on this task.

version of MNIST (LeCun et al., 2010), with the standard train-test split. Following Dong, Mnih, et al. (2020), input pixels are dynamically binarized, meaning that at each epoch they are randomly fixed to 0 or 1 with probability proportional to their intensity. For each training example, the model outputs a prediction and receives a reward of 1 if correct and 0 otherwise. I use a fully connected, feedforward network with 1, 2 or 3 hidden layers, each with 200 Bernoulli units, followed by a softmax output layer. I train using ADAM optimizer (Kingma & Ba, 2014) with the step-size fixed to 10^{-4} and batch size of 50 for 100 epochs. Step-size and layer-size hyperparameters follow Dong, Mnih, et al. (2020) for simplicity. I map the output of the Bernoulli units to one or negative one, instead of one or zero, as I found this greatly improved performance in preliminary experiments. I report results for HNCA and REINFORCE, both with and without an exponential moving average baseline subtracted from the reward. I use a discount rate of 0.99 for the moving

average.

Figure 4.1 shows the results, in terms of performance and gradient variance, for gradient estimates generated by HNCA and REINFORCE. I find that HNCA provides drastic improvement in terms of both gradient variance and performance over REINFORCE. Note that performance degrades with number of layers for both estimators, reflecting the increasing challenge of credit assignment. Subtracting a moving average baseline generally improves performance of both algorithms, except for HNCA in the single hidden layer case. The comparison between the two algorithms is qualitatively similar whether or not a baseline is used.

In Appendix B.5, I demonstrate that HNCA can also be used to efficiently train a stochastic layer as the final hidden layer of an otherwise deterministic network, this could be useful, for example, for learning a binary representation.

4.3 Optimizing a Known Function

I introduced HNCA in a setting where the reward function was unknown, and dependent only on the input context and the output of the network as a whole. Here, I extend HNCA to optimize the expectation of a known function f, which may have direct dependence on every unit. This new setting includes the problem of optimizing a discrete hierarchical VAE, which I will explore in my experiments, among other discrete optimization problems. I refer to this extension as f-HNCA. This setting is similar to the setting explored by Titsias et al. (2015), and f-HNCA differs from LEG mainly in its computationally efficient message-passing implementation, which in turn facilitates its application to multi-layer stochastic networks.

I assume the function $f = \sum_i f^i$ is factored into a number of *function components* f^i, which I index by i for convenience. This factored structure has two benefits, the first is computational. In particular, it will allow us to compute counterfactual values for each component with respect to changes to its input separately. The second is for variance reduction by realizing that we only need to assign credit to function components that lie downstream of the

unit being credited. A similar variance-reduction approach is also used by the NVIL algorithm of Mnih et al. (2014).

Each function component f^i is a deterministic function of a subset of the outputs of units in the network, as well as possibly depending directly on some parameters. Thus, $f^i = f^i(\widetilde{\text{pa}}(f^i); \theta^i)$, where θ^i is a set of real-valued parameters which may overlap with the parameters θ_Φ for some subset of units in the network, and $\widetilde{\text{pa}}(f^i)$ is the set of nodes in the network which act as input to f^i. Formally, f^i without arguments will refer to the random variable corresponding to the output of the associated function. I use the notation $\widetilde{\text{pa}}$, distinct from pa, to make it clear that function components are not considered nodes in the network. Likewise, I will use $\widetilde{\text{an}}$ to denote the ancestors of a function component, which includes $\widetilde{\text{pa}}(f^i)$ as well as all ancestors of $\widetilde{\text{pa}}(f^i)$.

The goal in this setting is to estimate the gradient of $\mathbb{E}[f]$ in order to maximize it by gradient ascent. By linearity of expectation, we can define unbiased estimators for $\frac{\partial \mathbb{E}[f^i]}{\partial \theta_\Phi}$ and sum over i to get an unbiased estimator of the full gradient.

HNCA with a Known Function: I now discuss how to extend the HNCA estimator to construct an estimator of $\frac{\partial \mathbb{E}[f]}{\partial \theta_\Phi}$ for a particular unit Φ in this setting. We begin by considering the gradient for a single function component $\frac{\partial \mathbb{E}[f^i]}{\partial \theta_\Phi}$. First, note that we can break the gradient into indirect and direct dependence on θ_Φ:

$$\frac{\partial \mathbb{E}[f^i]}{\partial \theta_\Phi} = \mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\operatorname{pa}(\Phi)))}{\partial \theta_\Phi} f^i\right] + \mathbb{E}\left[\frac{\partial f^i}{\partial \theta_\Phi}\right]. \tag{4.4}$$

This uses the same identity mentioned in Section 2.4 in the context of estimating gradients in a variational auto-encoder. The direct gradient $\frac{\partial f^i}{\partial \theta_\Phi}$ is zero whenever $\theta_\Phi \cap \theta^i = \emptyset$. When it is nonzero, $\frac{\partial f^i}{\partial \theta_\Phi}$, can be computed directly given I assume access to f^i. From this point on, I will focus on the left expectation.

The main added complexity in estimating $\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\operatorname{pa}(\Phi)))}{\partial \theta_\Phi} f^i\right]$, compared to the contextual bandit case, arises if f^i has a direct functional dependence on Φ. In this case, we can no longer assume that f^i is separated from Φ by $\text{mb}(\Phi)$. Luckily, this is straightforward to patch. Let $f^i_\Phi(\phi)$ be the random variable defined by taking the function $f^i(\widetilde{\text{pa}}(f^i); \theta^i)$ and substituting the specific value

ϕ instead of the random variable Φ into the arguments while keeping all other $\widetilde{\text{pa}}(f^i)$ equal to the associated random variables. By design, $f^i_\Phi(\phi)$ is independent of Φ given $\text{mb}(\Phi)$, which allows us to define the following unbiased estimator for $\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\,\text{pa}(\Phi)))}{\partial \theta_\Phi} f^i\right]$ (see Appendix B.6 for the full derivation):

$$\hat{G}^{f\text{-HNCA},i}_\Phi(\phi) \doteq \sum_\phi \rho_\Phi(\phi) \frac{\partial \pi_\Phi(\phi|\,\text{pa}(\Phi))}{\partial \theta_\Phi} f^i_\Phi(\phi), \tag{4.5}$$

where $\rho_\Phi(\phi)$ is as in Equation 4.2. As $\rho_\Phi(\phi)$ is defined with respect to $\text{ch}(\Phi)$, this estimator is only applicable if Φ has children (i.e. $\text{ch}(\Phi) \neq \emptyset$). In fact, even if Φ has children, we can ignore them if they have no downstream connection[5] to f^i, as such children cannot influence f^i. Thus, if $\text{ch}(\Phi) \cap \widetilde{\text{an}}(f^i) = \emptyset$, I instead define $\hat{G}^{f\text{-HNCA},i}_\Phi(\phi) \doteq \sum_\phi \frac{\partial \pi_\Phi(\Phi|\,\text{pa}(\Phi))}{\partial \theta_\Phi} f^i_\Phi(\phi)$. In Appendix B.8, I extend Theorem 4.1 to apply to f-HNCA, showing that using $\hat{G}^{f\text{-HNCA},i}_\Phi$ results in a variance-reduced estimator for $\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\,\text{pa}(\Phi)))}{\partial \theta_\Phi} f^i\right]$ compared to REINFORCE. The full f-HNCA estimator is defined by summing up these components and accounting for any direct functional dependence of f on network parameters:

$$\hat{G}^{f\text{-HNCA}}_\Phi \doteq \sum_\phi \frac{\partial \pi_\Phi(\phi|\,\text{pa}(\Phi))}{\partial \theta_\Phi} \left(\rho_\Phi(\phi) \sum_{i:\text{ch}(\Phi)\cap\widetilde{\text{an}}(f^i)\neq\emptyset} f^i_\Phi(\phi) \quad + \sum_{i:\text{ch}(\Phi)\cap\widetilde{\text{an}}(f^i)=\emptyset} f^i_\Phi(\phi) \right) + \sum_i \frac{\partial f^i}{\partial \theta_\Phi}. \tag{4.6}$$

If $\Phi \notin \widetilde{\text{an}}(f^i)$ then $\frac{\partial \mathbb{E}[f^i]}{\partial \theta_\Phi} = \mathbb{E}\left[\frac{\partial f^i}{\partial \theta_\Phi}\right]$ as Φ cannot influence something with no downstream connection. Hence, in the two leftmost sums over i in Equation 4.6, we implicitly only sum over i such that $\Phi \in \widetilde{\text{an}}(f^i)$.

In addition to the efficiency of computing counterfactual probabilities, for f-HNCA, we have to consider the efficiency of computing counterfactual function components $f^i_\Phi(\phi)$ given f^i. For function components with no direct connection to a unit Φ, this is trivial as $f^i_\Phi(\phi) = f^i$. If f^i is directly connected, then implementing f-HNCA with efficiency similar to HNCA will require that we are able to compute $f^i_\Phi(\phi)$ from f^i in constant time. This is the case if f^i is a linear function followed by some activation. For example, functions of the form $f^i = \log(\sigma(\vec{\theta} \cdot \vec{x} + b))$ which will appear in the ELBO function used in

[5]More generally, if only a subset of $\text{ch}(\Phi)$ lies in $\widetilde{\text{an}}(f^i)$ we can replace $\text{ch}(\Phi)$ in $\rho_\Phi(\phi)$ with $\text{ch}^i(\Phi) = (\text{ch}(\Phi) \cap \widetilde{\text{an}}(f^i))$, but, I will not use this in my experiments.

my variational auto-encoder (VAE; Kingma and Welling, 2014; Rezende et al., 2014) experiments. More algorithmic details can be found in Appendix B.7.

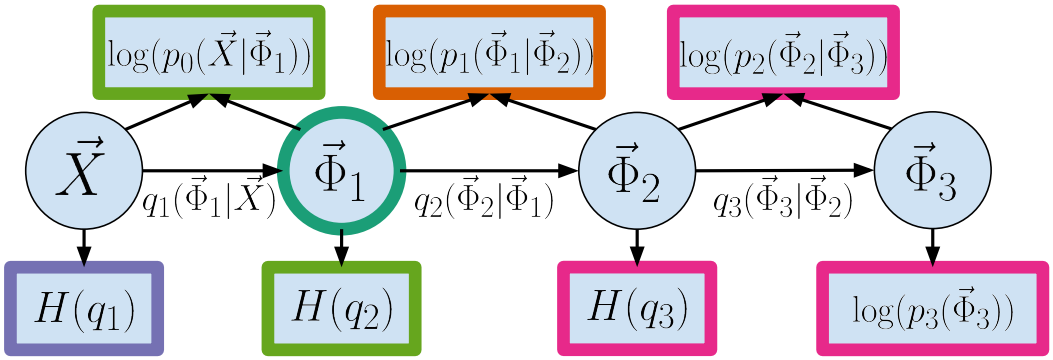

Figure 4.2: An illustration of the ELBO for a 3-layer discrete hierarchical VAE broken down into function components for f-HNCA. $\vec{X}$ is the input to be encoded, each additional circle is the latent state from a layer of the encoder network. Each rectangle is a set of function components which contribute to the ELBO. The parameters of the encoder are trained to maximize the ELBO by f-HNCA. Consider the f-HNCA estimator for $\vec{\Phi}_1$. The function components $H(q_1)$, marked in purple are upstream of $\vec{\Phi}_1$, however, $H(q_1)$ depends directly on θ_{q_1} and thus $\frac{\partial H(q_1)}{\partial \theta_{q_1}}$ is nonzero, so the entire contribution of $H(q_1)$ to the gradient estimate $\hat{G}_\Phi^{f\text{-HNCA}}$ will come from this gradient. The function components marked in green have only direct connection with $\vec{\Phi}_1$, so they will receive credit via $\hat{G}_\Phi^{f\text{-HNCA},i}(\phi) \doteq \sum_\phi \frac{\partial \pi_\Phi(\phi|\,\mathrm{pa}(\Phi))}{\partial \theta_\Phi} f_\Phi^i(\phi)$. The function components marked in orange have both direct connections and downstream connections mediated by $\vec{\Phi}_2$, so they will receive credit via Equation 4.5. Finally, the variables marked in pink have only mediated connections to $\vec{\Phi}_1$ through $\vec{\Phi}_2$, so $f_\Phi^i(\phi) = f^i$, the estimator for these variables essentially reduces to the original HNCA estimator defined in Equation 4.3.

Variational Auto-encoder Experiment: Here, I demonstrate how the f-HNCA approach can be applied to the challenging task of training a discrete hierarchical VAE. Consider a VAE consisting of a generative model (decoder) p and an approximate posterior (encoder) q, each of which consist of L discrete stochastic layers. Samples $\vec{X}$ are generated by p as

$$\vec{X} \sim p_0(\vec{X}|\vec{\Phi}_1), \vec{\Phi}_1 \sim p_1(\vec{\Phi}_1|\vec{\Phi}_2), ..., \vec{\Phi}_L \sim p_L(\vec{\Phi}_L),$$

while q approximates the posterior $\mathbb{P}(\vec{\Phi}_L|X)$ as a distribution which can be sampled as

$$q_L(\vec{\Phi}_L|\vec{\Phi}_{L-1}), \vec{\Phi}_{L-1} \sim q_{L-1}(\vec{\Phi}_{L-1}|\vec{\Phi}_{L-2}), ..., \vec{\Phi}_1 \sim q_1(\vec{\Phi}_1|\vec{X}),$$

where, each p_i and q_i represents a vector of Bernoulli distributions, each parameterized as a linear function of their input (except the prior $p_L(\vec{\Phi}_L)$ which takes no input, and is simply a vector of Bernoulli variables with learned means). Call the associated parameters θ_{p_i} and θ_{q_i}. We can train such a VAE by maximizing a lower bound on the log-likelihood of the training data, that is an evidence lower bound (ELBO) similar to the one discussed in Section 2.4. In this case, we can write the ELBO as $\mathbb{E}[f_E]$ where

$$f_E \doteq \log(p_0(\vec{X}|\vec{\Phi}_1)) + \sum_{l=1}^{L-1} \log(p_l(\vec{\Phi}_l|\vec{\Phi}_{l-1})) \\ + \log(p_L(\vec{\Phi}_L)) + H(q_1(\Phi_1|\vec{X})) + \sum_{l=1}^{L-1} H(q_{l+1}(\Phi_{l+1}|\vec{\Phi}_l)), \tag{4.7}$$

where H is the entropy of the distribution, and the expectation is taken with respect to the encoder q and random samples $\vec{X}$. Each $\vec{\Phi}_i$ is sampled from the associated encoder q_i. Note that each term in Equation 4.7 is a sum over elements in the associated output vector, we can view each element as a particular function component f^i. The resulting compute graph is illustrated in Figure 4.2.

I compare f-HNCA with REINFORCE and several stronger methods for optimizing an ELBO of a VAE trained to generate MNIST digits. I focus on strong, unbiased, variance-reduction techniques from the literature that do not require modifying the architecture or introduce significant additional hyperparameters. Since HNCA falls into this category, this allows for straightforward comparison without the additional nuance of architectural and hyperparameter choices. Specifically, I compare HNCA with REINFORCE leave one out (REINFORCE LOO; Kool et al., 2019) and DisARM (Dong, Mnih, et al., 2020). Note that in the multi-layer case, both DisARM and REINFORCE LOO require sampling an additional partial forward pass beginning from each layer, which gives them a quadratic scaling in compute cost with the number of layers. By contrast, HNCA requires only a single forward pass and a backward pass of similar complexity.

Initially, I found that f-HNCA outperformed the other tested methods in the single-layer discrete VAE case, but fell short in the multi-layer case. How-

ever, I found that a simple modification that subtracts a layer-specific scalar baseline, similar to that used by Mnih et al. (2014), significantly improved the performance of f-HNCA in the multi-layer case. Specifically, for each layer, I maintain a scalar running average of the sum of those components of f with mediated connections (those highlighted in pink and orange in Figure 2) and subtract it from the leftmost sum over i in Equation 4.5 to produce a centred learning signal.[6] I use a discount rate of 0.99 for the moving average.[7] I refer to this variant as f-HNCA with baseline. I also tested subtracting a moving average of all downstream function components in REINFORCE to understand how much this change helps on its own. It's not obvious how to apply a running average baseline to the other tested methods, as they already use alternative means to center the learning signal a naive moving average baseline would have expectation zero.

As in Section 4.2, I use dynamic binarization and train using ADAM optimizer with step-size 10^{-4} and batch size 50. Following Dong, Mnih, et al. (2020), the decoder and encoder each consist of a fully connected, stochastic feedforward neural network with 1, 2 or 3 layers. Each hidden layer has 200 Bernoulli units. I train for 840 epochs, approximately equivalent to the 10^6 updates used by Dong, Mnih, et al. (2020). For consistency with prior work, I use Bernoulli units with a zero-one output. For all methods, I train each unit based on downstream function components, as opposed to using the full function f. See Appendix B.9 for more implementation details.

Figure 4.3, shows the results in terms of ELBO and gradient variance, for gradient estimates generated by f-HNCA and the other methods tested. As in the contextual bandit case, I find that f-HNCA provides drastic improvement over REINFORCE. f-HNCA also provides a significant improvement over all other methods for the single-layer discrete VAE, but underperforms the other strong methods in the multi-layer case. On the other hand, f-HNCA with baseline significantly improves on the other tested methods in all cases. REIN-

[6] Using such a baseline for components without mediated connections would analytically cancel.

[7] I used the first value I tried, I did not tune it.

FORCE with baseline outperforms ordinary f-HNCA in the multi-layer cases. Hence, this baseline subtraction is a fairly powerful variance-reduction technique for REINFORCE, with strong complementary benefits with f-HNCA. In Appendix B.10, I additionally report multi-sample test-set ELBOs for the final trained networks, which reflect the same performance ordering as the training set ELBOs. In Appendix B.11, I perform an ablation experiment on f-HNCA with baseline and find that the choice of whether to exclude children when $\mathrm{ch}(\Phi) \cap \widetilde{\mathrm{an}}(f^i) = \emptyset$ has a significant performance impact, while the additional impact of excluding upstream function components is fairly minimal.

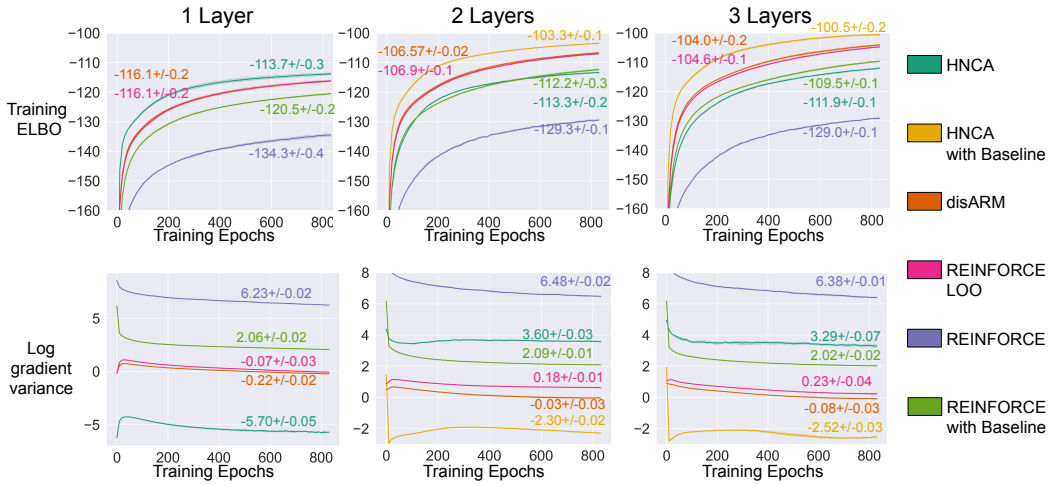

Figure 4.3: Training discrete VAEs to generate MNIST digits. Each line represents the average of 5 random seeds with error bars showing 95% confidence interval. Final values at the end of training are written near each line in matching colour. The left column shows the online training ELBO. The right column shows the natural logarithm of the mean encoder gradient variance. Mean gradient variance is computed as the mean over parameters and batches of the per-parameter empirical variance over examples in a training batch of 50. f-HNCA outperforms all other tested methods in the single-layer case, but underperforms in the multi-layer cases. f-HNCA with baseline outperforms the other methods in the multi-layer case. f-HNCA with baseline is excluded from the single-layer results as there are no mediated connections.

4.4 Discussion

This chapter introduced the second major contribution of this thesis, HNCA, which is also the second example of how our design decisions can influence a reinforcement agent's ability to take advantage of generic problem structure. In particular, if we know that the reinforcement learning agents in question are part of a network of discrete stochastic units, we can apply HNCA to provide a significant improvement in credit assignment capability compared to the naive approach.

HNCA is inspired by Hindsight Credit Assignment (Harutyunyan, Dabney, Mesnard, et al., 2019), and can be seen as an instance of local expectation gradients, extending the work of Titsias et al. (2015) by providing a computationally efficient message passing algorithm and extension to multi-layer networks of stochastic units. The computationally efficient approach of HNCA directly addresses concerns in the literature that LEG is inherently computationally expensive (Mnih & Rezende, 2016; Tucker et al., 2017). I proved that HNCA is unbiased, and that it reduces variance compared to REINFORCE. Empirically, I showed that HNCA outperforms strong methods for training a single-layer Bernoulli VAE, and when subtracting a simple moving average baseline also outperforms the same methods for the case of a multi-layer Hierarchical VAE.

It's worth highlighting that efficient implementation of HNCA is predicated on the ability to efficiently compute counterfactual probabilities or function components when a single input is changed. This is not always possible, for example, if f is the result of a multi-layer deterministic network. An example of this situation is the nonlinear discrete VAE architecture explored by Dong, Mnih, et al. (2020) and Yin et al. (2019) where the encoder and decoder are nonlinear networks with a single stochastic Bernoulli layer at the outputs. However, as I show in Appendix B.5, HNCA can be used to train a final Bernoulli hidden layer at the end of a nonlinear network.

In addition to optimizing a known function of the output of a stochastic network, I showed that HNCA can be applied to train the hidden layers of a multi-layer discrete network in an online learning setting with an unknown

reward function. REINFORCE LOO and DisARM, which rely on the ability to evaluate the reward function multiple times for a single training example, cannot.

Future work could explore combining HNCA with other methods for complimentary benefits. One could also explore extending HNCA to propagate credit multiple steps which would presumably allow further variance reduction but presents challenges, as the relationships between more distant nodes in the network become increasingly complex.

HNCA provides insight into the challenges of credit assignment in discrete stochastic compute graphs, which has the potential to have an impact on future approaches.

Chapter 5

Iterative Option Discovery for Planning, by Planning

Recall, once again, that the central focus of this thesis is on demonstrating how we can design RL agents which take advantage of generic problem structure. In keeping with this theme, the previous chapter introduced HNCA as a specific example of an approach which exploits generic problem structure to design a more efficient algorithm. In the case of HNCA, the exploited structure was the connectivity of a network of interacting units. This chapter introduces another example, where in this case the exploited structure is the tendency for optimal actions in temporally contiguous states to be interdependent. In particular, I outline a perspective that an important benefit of learning a set of options for planning is that options facilitate modelling dependency between actions in temporally contiguous trajectory segments in a way that a single policy does not. Based on this perspective, I propose an approach to learning a set of options to maximize the likelihood of actions selected by a computationally expensive planner over temporally contiguous trajectory segments. I demonstrate that using options learned in this manner during planning results in improved performance compared to using a single policy trained analogously.

Consider the planning problem faced by a human approaching a four-way intersection while driving in an unfamiliar city. Figuring out precisely the correct action requires complex reasoning to incorporate knowledge of local geography and the desired destination. However, despite this complexity, there

is a relatively small set of short-term temporally extended behaviours which might be useful. Namely, we might want to turn left, turn right, or go straight. On the other hand, we can usually rule out the enormous space of behaviours which would lead us to run off the road. This highlights the motivation for Option Iteration (OptIt). The optimal policy may be a very complicated, and thus difficult to learn, function of the current state. Nonetheless, there may be a small set of potentially useful short-term behaviours which are comparatively easy to learn and broadly applicable. Motivated by this example, OptIt aims to discover a set of options such that in every state visited, at least one option in the set is good over some future horizon.

OptIt learns options with an approach similar to Expert Iteration (Anthony et al. (2017); ExIt), which is itself essentially a version of approximate policy iteration. A closely related approach was used to obtain impressive results in the game of Go by AlphaZero (Silver et al., 2017). Given some search algorithm which takes a prior policy and uses some computation to generate an improved policy, ExIt iteratively updates the prior policy to better match the output of the search procedure. In doing so, ExIt amortizes the cost of search by distilling the results into a relatively inexpensive neural network. This results in a virtuous cycle in which each search results in an improved policy, which improves the learned policy, which then improves future searches.

Intuitively, learning a set of policies such that at least one is good in each state is likely easier since it allows the algorithm to hedge its bets. Without learning the nuances of a previously unseen state s well enough to decide which option is best, an algorithm may still be able to discover a set such that at least one option included in the set is likely to be good when initiated in s. The learned set of options can then be used to improve search by allowing actions to be evaluated under a variety of different plausible behaviours rather than a single learned policy.

5.1 Related Work

The work presented in this chapter builds on the options framework of Sutton et al. (1999). In particular, I focus on the problem of option discovery, where rather than providing an agent with a set of prespecified options, we wish to design algorithms that allow agents to build a set of useful options on their own.

A variety of methods have been proposed for option discovery, with a number of different motivations behind them. Some approaches aim to directly optimize options to facilitate good performance on a task or distribution of tasks (Bacon et al., 2017; Frans et al., 2018; Veeriah et al., 2021). Others aim to learn options which help navigate between disparate regions of state space, for example by identifying bottleneck states (McGovern et al., 2001; Stolle et al., 2002), exploiting graph-theoretic properties of the transition dynamics (Klissarov et al., 2023; Machado et al., 2017), or encouraging options to contain a lot of information about their state at termination (Eysenbach et al., 2019; Gregor et al., 2016; Harutyunyan, Dabney, Borsa, et al., 2019). A large body of work considers learning hierarchical policies in which a high-level policy, trained to maximize task performance, outputs subgoals that a low-level policy is trained to achieve (Dayan et al., 1992; Hafner et al., 2022; Vezhnevets et al., 2017).

Jinnai et al. (2019) and Wan et al. (2022) share my focus on discovering options that facilitate planning, albeit using value iteration rather than decision-time planning. Co-Reyes et al. (2018) jointly learn a latent conditioned policy and trajectory level model such that the model predicts the trajectory resulting from the policy when conditioned on a particular latent state. The resulting latent state and model are then used in a decision-time planning procedure, similar to the way the learned set of options is used in OptIt.

There is also a body of work focusing on discovering sets of options which fit a dataset of unsegmented expert demonstrations (Fox et al., 2017; Kipf et al., 2019; Krishnan et al., 2017; Shankar et al., 2020; Zhu et al., 2022).

OptIt can be seen as an instance of this where the demonstration data is dynamically generated by a search procedure and the discovered options are used to improve the search.

This chapter focuses on option discovery in a single-task setting, while much of the literature focuses on the multi-task setting. The distinction is not always clear-cut but to give some examples: Frans et al. (2018), Veeriah et al. (2021), and Wan et al. (2022) all expressly focus on finding options which are useful across a distribution of related tasks as opposed to one specific task. In a sense, the multi-task setting is more natural for option discovery as one can imagine learning a set of options which capture temporal regularities in the optimal policy shared across the tasks. For a single task, finding a single globally optimal policy is sufficient, which makes it less obvious why we'd want to learn a set of behaviours. In Section 5.2, I motivate the benefit of options in a single, complex, task by the desire to quantify uncertainty in the joint distribution over actions. One can think of this as being loosely related to the multitask setting. Effectively, the assumption of multiple ground truth tasks, each with a distinct optimal policy, is replaced with a distribution over optimal policies induced by uncertainty, despite there being only a single ground truth task.

OptIt computes an improved policy and value function estimate based on Monte-Carlo evaluation under a set of possible options. This is an instance of Generalized Policy Improvement, introduced by Barreto et al. (2017).

As already discussed, OptIt is inspired by the ExIt (Anthony et al., 2017) and the closely related approach used in AlphaZero (Silver et al., 2017). Recently, Zahavy et al. (2023) have demonstrated that optimizing a diverse set of players, encoded as different latent-state inputs to a single network, using a quality-diversity objective can significantly improve the performance and robustness of AlphaZero.

5.2 Options as a Way to Represent the Joint Distribution Over Future Optimal Actions

While temporal abstraction is widely regarded as key to scaling RL algorithms to increasingly complex problems, empirical results are mixed. Likewise, it is not always clear how or why learning a set of options should lead to improvement over simply learning a single strong policy, especially in a single-task setting. For example, Bacon et al. (2017) found that without additional regularization, learned options tend to collapse down to single-step primitive actions. For this reason, before we suggest new techniques for option discovery, it is useful to articulate how specifically we believe learning a set of options could be helpful in a sufficiently complex single-task setting. Here, I expand on the intuition presented in the chapter introduction to articulate a view that options can allow us to capture information about the joint distribution over optimal actions in a way that learning a single strong policy does not. My reasoning is reminiscent of the case for learning joint predictions over labels presented by Osband et al. (2021), as well as related to posterior sampling for RL (Osband et al., 2013; Strens, 2000).

To begin, define the entropy-regularized optimal policy

$$\pi^*_\beta(a|s) = \frac{\exp(q^*(s,a)/\beta)}{\sum_{a'} \exp(q^*(s,a')/\beta))}, \tag{5.1}$$

where $q^*(s,a)$ is the unknown optimal action-value function of the true MDP and the entropy-regularization factor β is a hyperparameter. Imagine we have observed a dataset D of samples from π^*_β in various states. Given this data, along with some prior over regularized optimal policies, we can in principle use Bayes theorem to determine a posterior over regularized optimal policies $\mathbb{P}(\pi^*_\beta = \pi|D)$.[1] Now, imagine would like to provide a planner with an approximation to the distribution of regularized optimal-policy actions to guide its search for a good action in this state. Say we have the choice between specifying this approximation as a single policy, or as a mixture distribution

[1]Note that while there may be many unregularized optimal policies sharing the same action-values, using a regularized version as in Equation 5.1 makes the solution unique for a given MDP.

over N different policies. The single policy assigns some probability $\pi(a|s)$, and thus given a sequence of K states $\vec{s} = (s_0, s_1, ...s_{K-1})$ it will assign a joint probability to action sequence $\vec{a} = (a_0, a_1, ...a_{K-1})$ as follows

$$\hat{p}(\vec{a}|\vec{s}) = \prod_{k=0}^{K-1} \pi(a_k|s_k). \tag{5.2}$$

On the other hand, given an approximate posterior consisting of a mixture of N policies $\pi_n(a|s)$ for $n \in \{0, ..., N-1\}$ weighted by some $\rho(n)$, we get the following joint probability over actions for $\vec{s}$

$$\hat{p}(\vec{a}|\vec{s}) = \sum_{n=0}^{N-1} \rho(n) \prod_{k=0}^{K-1} \pi_n(a_k|s_k). \tag{5.3}$$

It's clear that Equation 5.3 includes Equation 5.2 as a special case where $\rho(n) = \mathbb{1}(n = 0)$ and $\pi_0 = \pi$. Furthermore, as Equation 5.3 allows nontrivial dependency between regularized optimal-policy actions in different states, it can in general be a significantly better representation of the true posterior predictive distribution

$$\mathbb{P}(\vec{A} = \vec{a}|\vec{S} = \vec{s}, D) = \int_{\pi^*_\beta} \mathbb{P}(\pi^*_\beta = \pi|D) \prod_{k=0}^{K-1} \pi(a_k|s_k) d\pi^*_\beta.$$

Where $\vec{A}$ is a random vector containing actions selected by the entropy regularized optimal policy in the states contained in $\vec{S}$. To give a simple example, consider a tabular environment, called Compass, in which an agent starts each episode in a random location on a 2-dimensional, square, grid of cells with K cells in total. In each cell, the agent has the choice to move up, down, left or right. Termination occurs upon reaching one of the 4 edges of the grid, with a large positive reward on one edge and a large negative reward on the other three. The positive reward is a priori equally likely to lie on any edge. In addition, assume there is a minor negative reward for each time step such that remaining on the grid indefinitely is suboptimal. Thus, the optimal policy will always go in the same direction for every state in a particular grid instance. In addition, assume the state includes an integer $m \in \{0, ..., M-1\}$ which indicates the identity of the grid. For a given m, the positive reward will always

be on the same side, but it is a priori unknown which indices correspond to which side. The states of this environment can be represented by $s = (k, m)$ where k indexes all the cells within the grid and m is the index of the grid.

Now assume, based on some combination of prior knowledge and data, the posterior over optimal policies[2] captures the fact that for a fixed, unobserved, grid the correct behaviour is either to always go left, right, up or down with equal probability. Now, consider approximating the posterior predictive distribution for the vector set of states $\vec{s}$ such that $\vec{s}[k] = (k, \tilde{m})$ for all k grid cells and a single previously unobserved grid index $\tilde{m}$. In particular, consider the minimal cross-entropy approximation using either a single policy as in Equation 5.2 or a mixture of policies as in Equation 5.3. With a single policy, we aim to minimize

$$\begin{aligned}
&\mathbb{E}_{\vec{A}\sim\mathbb{P}(\vec{A}|\vec{S}=\vec{s},D)}\left[-\log\left(\prod_{k=0}^{K-1}\pi(A_k|(k,\tilde{m}))\right)\right] \\
&\quad=\sum_{k=0}^{K-1}\mathbb{E}_{A_k\sim\mathbb{P}(A_k|S_k=(k,\tilde{m}),D)}\left[-\log(\pi(A_k|(k,\tilde{m})))\right].
\end{aligned}$$

Where $\vec{A} = (A_0, ..., A_{K-1})$ is a random sequence of actions drawn from the true posterior predictive distribution. Since the marginal likelihood $\mathbb{P}(A_k|S_k = (k, \tilde{m}), D)$ is 0.25 for each action, the best we can do is to set $\pi(A_k|(k, \tilde{m})) = 0.25$ uniformly. This yields a total cross-entropy of $K\log(4)$. On the other hand, using a mixture of four policies we can exactly represent the predictive posterior over $\vec{s}$. In particular, we can set $\rho(0) = \rho(1) = \rho(2) = \rho(3) = 0.25$, and $\pi_0(a|(k, \tilde{m}))) = \mathbb{1}(a = left)$, $\pi_1(a|(k, \tilde{m}))) = \mathbb{1}(a = right)$, $\pi_2(a|(k, \tilde{m}))) = \mathbb{1}(a = up)$, $\pi_3(a|(k, \tilde{m}))) = \mathbb{1}(a = down)$ for all k. Since this exactly matches the true predictive posterior it gives the best possible cross-entropy of $\log(4)$, K times lower than what is achievable from fitting a single policy, and equal to the true entropy. In turn, this corresponds to a 4^d times higher probability of the action sequence that takes the agent directly to the edge with a positive reward, where $d \leq \sqrt{K}$ is the distance from the agent's current location to the edge with positive reward. Roughly speaking, a planner using this mixture policy to generate rollouts will need to consider exponentially fewer trajecto-

[2]I consider unregularized optimal policies in this case.

ries on average to find the optimal sequence compared to one using the best-fit single policy.

Compass also illustrates the potential benefit of searching for a set of options such that at least one in the set is good only for some short horizon into the future in each state. The same four option policies suffice to represent the posterior predictive distribution over all $s = (k, \tilde{m})$ for any fixed $\tilde{m}$. On the other hand, representing the predictive posterior jointly for all possible k and m would require 4^M policies, as we'd have to represent every joint configuration of optimal action for each index m. Furthermore, assuming we are using the set of options for planning locally over a particular time horizon, we need not accurately represent the predictive posterior globally, but only locally for the states likely to be encountered during planning.

I postulate that the kind of local structure that exists in Compass, with strong dependency among the optimal actions for temporally contiguous states, is likely to be present in many problems of interest. In particular, due to spatial locality, temporally contiguous states will tend to involve interaction with a highly overlapping set of subsystems in the world and thus any information we gain about these subsystems will be likely to provide information about the optimal behaviour in many such states. Note, however, that such local dependency need not hold a priori. One can easily construct posterior distributions such that the optimal action in one state is uninformative about the optimal action in the states that follow temporally while providing information about the optimal action in faraway states. Thus, such local structure represents a nontrivial assumption about the kinds of problems we are interested in.

5.3 Option Iteration

Rather than learning a single strong policy, OptIt aims to learn a set of policies such that, in every state we encounter, at least one policy in the set is good for some horizon into the future. To learn such a set, I adopt an approach similar to ExIt by optimizing for agreement between the learned set of option policies and the results of a relatively computationally expensive search. In particular,

OptIt maintains a set of N option policies for $n \in \{0, ..., N-1\}$, I denote the probability of sampling action a in state s with option n as $\pi_n(a|s;\theta)$.[3]

For each contiguous trajectory segment of K steps, we aim to have a weighted mixture of the option policies in our learned set which is a close match to the results of a search procedure jointly for all K steps. More precisely, consider a given trajectory segment $s_0, s_1, ..., s_{K-1}$ along with $\tilde{\pi}(\cdot|s_k)$ representing the search policy, that is, the improved policy returned by running a search procedure in s_k. OptIt will optimize the following loss for randomly sampled segments of K trajectory steps from a replay buffer

$$\mathcal{L} = \mathbb{E}_{A_k \sim \tilde{\pi}(A_k|s_k)} \left[-\log \left(\sum_{n=0}^{N-1} \rho(n|s_0;\theta) \prod_{k=0}^{K-1} \pi_n(A_k|s_k;\theta) \right) \right], \tag{5.4}$$

where $\rho(n|s_0;\theta)$ is a learned policy over options conditioned on only the first state in the sequence. Using a weighting conditional on only s_0 reflects the fact that, when planning, we must select an option conditional on only s_0. If instead ρ was conditioned on the whole trajectory segment, we may end up with one of two different options being best depending on random transitions following s_0. This is undesirable given we wish to use the options for planning from the current state and thus want some option to be good in expectation conditional on the current state, rather than only conditional on some random future states. Equation 5.4 is exactly the cross-entropy between the search policy for each state in the sequence and the joint distribution over actions induced by the weighted mixture of option policies.

Note that A_k is sampled independently from the action actually executed in the environment. In practice, I stochastically sample $A_k \sim \tilde{\pi}(A_k|s_k)$ in each update rather than optimizing the expectation in Equation 5.4 directly as the computational complexity of the latter grows as $|\mathcal{A}|^k$ and is thus intractable for moderately large k.

Even if ρ is uniform over options, Equation 5.4 reduces to the expected LogSumExp over the log-likelihood of the search action sequence under each option. LogSumExp acts as a smoothed maximization, putting more emphasis

[3]Throughout I use θ generically to represent the complete set of learnable parameters defining an agent, though in general only a subset will be used by each function.

on higher-value elements. Thus, relative to simply optimizing the average log-likelihood over options, Equation 5.4 will tend to favour making options that are already a good match better. The emphasis on good options will be even stronger if ρ is well learned. This motivates the intuition that OptIt aims to learn a set of options such that, for every state we encounter, at least one is good for some horizon into the future.

There are two key differences between OptIt and ExIt. First, ExIt optimizes a single policy rather than a set. Second, ExIt optimizes independently in each state rather than for trajectory segments. The analogue to Equation 5.4 for ExIt is

$$\mathcal{L} = \mathbb{E}_{A\sim\tilde{\pi}(A|s)} \left[-\log(\pi(A|s;\theta)) \right].$$

Note the importance of optimizing over trajectory segments in OptIt. When optimizing for single-step agreement, using a set of options is unlikely to be beneficial as the weighted mixture itself would collapse into a single policy.

Option Iteration with Monte Carlo Search: OptIt could be integrated with any planning algorithm that uses search to compute an improved policy in each visited state, including Monte-Carlo tree search (MCTS; Coulom, 2006; Kocsis et al., 2006). However, Monte-Carlo search (MCS; Tesauro et al., 1996) provides a particularly appealing use case as the addition of learned options could help to mitigate some of its inherent weaknesses. Unlike MCTS, MCS does not perform policy improvement in non-root nodes during the search, each action is only evaluated under a specific rollout policy. On the other hand, MCS offers certain advantages relative to MCTS. In particular, MCS is straightforward to parallelize and also straightforward to apply to stochastic environments.

Performing MCS with a strong learned set of options can help mitigate the drawbacks and allow us to capitalize on the benefits inherent in its simplicity. By searching in a joint space of actions and options rather than just primitive actions, we can evaluate each action under various possible behaviours rather than a single learned rollout policy. Unless one policy in the set is best everywhere, finding the best action under the best policy in a set for each

Algorithm 2 MCS with Options

Input: $\theta, \bar{\sigma}, s$
for a in $\mathcal{A}$ **do**
  for n in $0 : N-1$ **do**
    $M = simulation\_budget/(|\mathcal{A}|N)$
    $\hat{Q}_n(a,s) \leftarrow 0$
    for j in $0 : M-1$ (in Parallel) **do**
      Sample $s_1 \sim p(\cdot|s,a)$
      Simulate $\pi_n(\cdot|\cdot;\theta)$ for $K-1$ steps from s_1
      Get simulated trajectory $(s_0, a_0, r_1, ..., s_K)$
      $G \leftarrow \sum_{k=0}^{K-1} \gamma^k r_{k+1} + \gamma^K v(s_K;\theta)$
      $\hat{Q}_n(a,s)$ += G/M
    end for
  end for
end for
$\tilde{p}(a,n) \propto \exp(\hat{Q}_n(a,s)/(\bar{\sigma}\beta))$
$\tilde{\pi}(a|s) \leftarrow \sum_n \tilde{p}(a,n)$
$\tilde{a} \leftarrow \operatorname{argmax}_a \max_n \hat{Q}_n(a,s)$
$\tilde{v} \leftarrow \max_{a,n} \hat{Q}_n(a,s)$
return $\tilde{a}, \tilde{v}, \tilde{\pi}$

encountered state will give us better actions overall than selecting the best action under any single policy from the set. If the set of learned options is sufficiently rich to capture the plausible future behaviours then it may not be necessary to explicitly build a tree over future trajectories. I will apply OptIt on top of MCS in my experiments in this thesis leaving its application to other search algorithms for future work.

I run MCS in the joint space of options and initial actions. Effectively, this approach searches for the best combination of initial action and subsequent behaviour from the set of available learned options. MCS with any finite set of options and actions effectively reduces the search problem to a finite-armed bandit setting. We could thus choose any finite-armed bandit algorithm to select options and actions during search. To avoid complex interactions between the bandit algorithm and policy-learning procedure, which could confound my main focus, I make a simple choice. Namely, I fix the total number of simu-

lations used in each state and allocate this simulation budget evenly amongst the joint space of options and initial actions. For example, if we allocate 1000 total simulations with 5 options and 4 actions, then for each option n and action a we would perform $1000/(4 \cdot 5) = 50$ simulations where initial action a is selected and the policy of option n is followed thereafter for the remainder of the rollout length. Applying OptIt with more sophisticated choices of bandit algorithm such as PUCB (Rosin, 2011) or Sequential Halving (Danihelka et al., 2021; Karnin et al., 2013) is an interesting direction for future work.

While we run MCS in the joint space of options and actions, we ultimately use the search results only to select an action to execute and provide an improved policy over actions alone as a target for option learning. To do this, I first compute a joint distribution over options and actions based on the entropy regularized average return resulting from the simulations:

$$\tilde{p}(a, n) \propto \exp(\hat{Q}_n(a, s)/(\bar{\sigma}\beta)),$$

where $\hat{Q}_n(a, s)$ is the average K step bootstrapped return for simulations where the initial action is a and option n is followed thereafter, β is an entropy regularization hyperparameter, and $\bar{\sigma}^2$ is an exponentially weighted average of the variance of returns for recent rollouts used to reduce sensitivity to β across problems. I next compute the search policy over actions by marginalizing out the options $\tilde{\pi}(a|s) = \sum_n \tilde{p}(a, n)$, the action $\tilde{a}$ returned by the search to actually execute in the environment is $\tilde{a} = \operatorname{argmax}_a \max_n \hat{Q}_n(a, s)$. I also maintain a value estimate, which is used only in computing bootstrapped returns during search, the target for which is simply $\tilde{v} = \max_{a,n} \hat{Q}_n(a, s)$.

Pseudocode for my implementation of MCS with options is displayed in Algorithm 2. Since I consider episodic environments, I define $v(\perp; \theta) = 0$ for the terminal state.

After searching in each state, the resulting $\tilde{\pi}$ and $\tilde{v}$ are stored in a replay buffer along with the current state s. The option policies and the policy over option ρ are trained using Equation 5.4 on randomly sampled batches of contiguous length K trajectory segments.[4] The approximate value function is

[4]I sample start states randomly and truncate segments at terminal or timeout states. I

trained to minimize squared error relative to $\tilde{v}$.

5.4 Does Option Iteration Learn Useful Options for Monte-Carlo Search?

Here, I evaluate the ability of OptIt to discover useful options from the results of search. In all my experiments in this chapter, I use simple feed-forward neural networks for function approximation, with binary observations as input. Option policies as well as the policy over options ρ are implemented using a network with a single shared trunk with only the output layer differing, hence the difference in parameter count due to using multiple options versus a single policy is very minor. The value function is approximated using a separate feed-forward network of equal size. I begin with a simple domain as a pedagogical illustration before moving on to a more challenging and plausible planning problem.

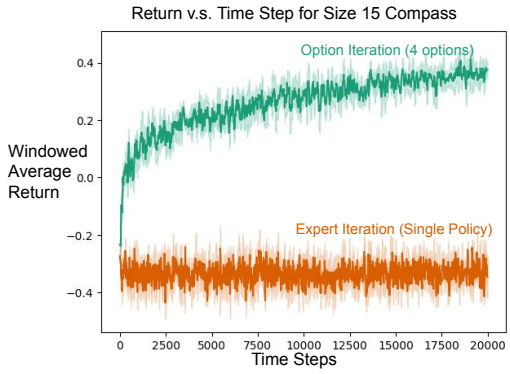

Figure 5.1: Windowed average return over training time for Monte-Carlo search with options learned via OptIt compared to Mote-Carlo Search with just primitive actions using a rollout policy learned via ExIt in 15×15 Compass. Here, and in all other figures in this chapter, I report undiscounted return up to either termination or timeout. Error bars show 95% confidence interval over 5 random seeds.

An Illustrative Domain Highlighting the Benefit of Option Iteration: I first illustrate the benefit of OptIt in a variant of the Compass domain, as described in Section 5.2. The domain used here differs slightly from that

also divide the loss by the trajectory segment length, only counting truncated steps in case of truncation.

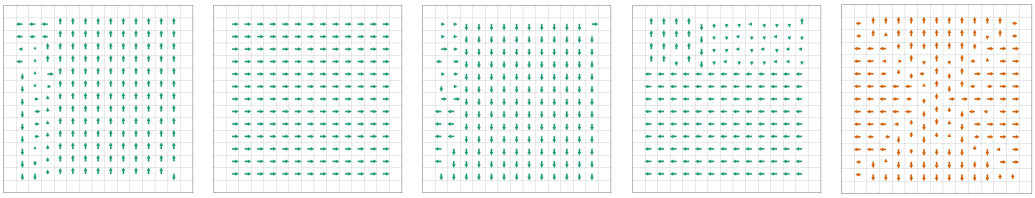

Figure 5.2: The left four plots with green arrows show the highest probability action selected by each of the 4 options learned by OptIt after 100,000 steps in the 15×15 Compass environment. The rightmost plot with orange arrows shows the action selected by the single policy learned with ExIt. In all cases, the length of the arrow indicates the probability of the action with probability 1 corresponding to the arrow touching the edge of the grid cell. The options learned by OptIt each tend to select a certain direction to reach a particular edge of the grid as quickly as possible. The single policy learned by ExIt generally just takes the shortest path to any edge of the grid.

described in Section 5.2 in that the index m identifying the grid is not provided to the agent. Instead, there is simply a reward of 1 on one edge and a reward of -1 on the other edges at random. Which edge the positive reward is located on is not observable and can be determined by the agent only by performing rollouts under the model. This is essentially analogous to the case where M is so large that it is unlikely that the agent will ever see the same grid twice. There is also a reward of $-1/(grid\ width)$ at each time step to incentivize the agent not to avoid termination indefinitely, note that even with this penalty it is still always better to move to the edge with the positive reward.

This situation is somewhat contrived in that information about which edge the reward is on is available to the world model, but not observable. However, it serves as an empirical verification of the arguments for OptIt laid out in Section 5.2. I present this as a surrogate for the more plausible situation where the optimal policy is a function of the agent's observations, but in a nuanced way that has not been well learned, due to either limited policy training or policy network capacity, but which can nonetheless be determined by simulation using the world model. Subsequent sections will evaluate OptIt in environments without such contrived partial observability.

I set the width of the grid to 15. The timeout period for an episode was set to 20 such that there was enough time to reach the positive reward from any

initialization position under the optimal policy. After the time-out period, a new episode begins regardless of whether the terminal state has been reached.[5] The rollout length for options and length K of trajectory segments used in the OptIt loss are likewise set to 20 such that it is possible to reach any edge of the grid in one rollout. In this experiment, I use a relatively low simulation budget of 50 rollouts. I test an agent using 4 options learned with OptIt against an agent operating in the space of primitive actions with a rollout policy learned from search results using ExIt. Other hyperparameter settings are available in Appendix C.5.

The resulting learning curves are displayed in Figure 5.1. OptIt is able to quickly learn approximations to the four directional options that allow the search to rapidly locate the positive reward on any of the 4 edges and from that point forward achieves strong performance. On the other hand, the single rollout policy learned using ExIt tends to just move toward whatever edge is closest (presumably to escape the negative reward per time-step as quickly as possible) which is much less useful for search. Visualizations of the learned options and policy are provided in Figure 5.2.

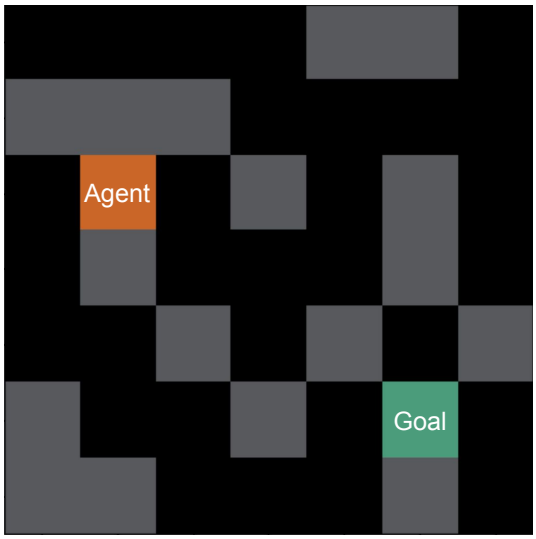

Figure 5.3: Example of a state in the 7x7 ProcMaze environment. Note that the maze layout is randomized in each episode, making the state space much larger than that of a particular maze.

[5]Note that timeouts are not applied during search rollouts.

Evaluating Option Iteration in a Challenging Planning Domain: I now empirically evaluate OptIt on a conceptually simple environment designed to present a significant planning challenge. Specifically, I use a variant of the ProcMaze environment already introduced in Chapter 3, which I will refer to as ElectricProcMaze for reasons I will explain shortly.

Unlike the Compass environment, ElectricProcMaze is fully observable. There no information a priori hidden from the policy network that affects the correct behaviour which would require the use of options to represent the joint distribution of optimal actions. However, this environment is challenging enough that learning could benefit from representing a distribution of possible behaviours before the single optimal policy is well learned.

In ElectricProcMaze, instead of remaining in place upon transitioning into a wall, the agent is allowed to move into the wall cell but with a large negative reward (analogous to an electric shock in animal experiments) set to be equal to one more than the largest possible number of steps required to reach the goal across all possible maze configurations. In addition, compared to the ProcMaze version used in Chapter 3, the no-op action was removed and random teleportation to the goal was replaced with a timeout after a fixed number of steps.

I use ElectricProcMaze rather than ProcMaze as the latter has the property that the greedy policy with respect to the action-value function of the uniform random policy is optimal while ElectricProcMaze does not. Laidlaw et al. (2023) demonstrated that this characteristic is surprisingly common, particularly in environments where standard deep RL algorithms perform well. However, it is undesirable for this chapter, where the main focus is on policy learning for search. See Appendix C.1 for further details on this point.

I present experiments in 7x7 mazes, an example of which is displayed in Figure 5.3. I found in preliminary experiments that smaller mazes were easily solved by MCS in the space of primitive actions and hence presented little potential to benefit from options. I also use a timeout of 120 steps after which the episode is ended to avoid the agent getting stuck indefinitely on a single challenging maze.

I evaluate OptIt with MCS using 5 learned options against ExIt with a single learned policy. ExIt is implemented using the exact same code as OptIt, simply reducing the number of options to 1.[6] In order to establish a strong performance baseline for ExIt on this environment I sweep the step-size and entropy-regularization parameters and select the one with the best average return in the final 100,000 of 500,000 training steps, see Appendix C.5 for details. I then use the same hyperparameters for OptIt. Each is given the same simulation budget per step, which for OptIt is distributed evenly over all option-action combinations and for ExIt is distributed over only actions. In each case, the approximate value function is used to estimate the value of the final state in length 5 rollouts during MCS. As an additional baseline, I also tested training 5 options by optimizing the mean cross-entropy over options, that is

$$\mathcal{L} = \mathbb{E}_{A_k \sim \tilde{\pi}(A_k|s_k)} \left[-\frac{1}{N} \sum_{n=0}^{N-1} \log \left(\prod_{k=0}^{K-1} \pi_n(A_k|s_k;\theta) \right) \right].$$

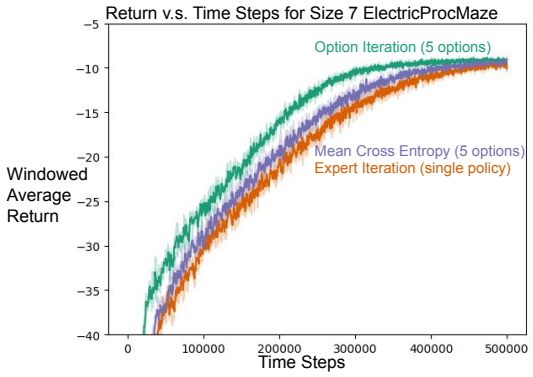

Figure 5.4: Windowed average return over training time for OptIt and baselines on size 7 ElectricProcMaze. Error bars show 95% confidence interval over 5 random seeds. The y-axis is thresholded at -40 to omit the rapid period of initial improvement. Running MCS in the space of options learned with OptIt provides a significant benefit compared to using primitive actions evaluated under the single learned rollout policy.

The results are displayed in Figure 5.4. Planning with options discovered by

[6] This means that ExIt is also trained on sequences and uses samples from the search policy rather than directly minimizing cross-entropy. I also tried implementing ExIt with independent samples and exact cross-entropy but surprisingly found this to be significantly worse, hence I report results with the same setting as OptIt for simplicity. I include an ablation study to better understand these choices in Appendix C.2.

OptIt shows a clear benefit over using a single learned rollout policy. We also see that using the OptIt loss significantly outperforms the mean-cross entropy baseline. The latter performs just slightly better than ExIt, suggesting that the specialization induced by maximizing the joint likelihood of action sequences results in a more useful set of options.

Evaluating Option Iteration in a Challenging Domain with Hierarchical Structure: The ElectricProcMaze results demonstrate that using options learned by OptIt in MCS can significantly accelerate learning in a challenging planning domain compared to using a single policy learned by ExIt. This is despite the fact that randomized mazes have no obvious hierarchical structure. We may expect to see more significant benefits in domains where there is some underlying hierarchical structure in the environment. In this section, I introduce another conceptually simple environment, built on top of ElectricProcMaze, to investigate this.

I call the hierarchical environment introduced here HierarchicalElectricProcMaze. This environment consists of a base environment (in this case ElectricProcMaze) and a controller environment. The basic idea is to create a simplified abstraction of an agent interacting with a low-level controller such that executing a single action in the base environment via the controller requires a sequence of coherent actions in the controller environment. A more complex example would be a robot learning to play chess when it is required to physically manipulate the pieces to take action in the game.

The controller environment consists of an 8×8 grid with buttons placed in the centre of each edge. Each button corresponds to a particular action in the base environment. The action space consists of moving in the 4 cardinal directions. Each primitive action moves the agent one cell in the controller environment. Upon reaching a button in the controller environment the selected action to execute in the base environment is changed to the action associated with the button. On every eighth time step in the controller environment, the currently selected base-environment action is executed and the base environment is advanced by one time step. At this time, the selected base-environment action is reset to no-op. Executing no-op will cause the

agent not to move in the base environment, hence to select an action besides no-op in the next base-environment step, the agent must once again move to a button within 8 time steps. The environment is fully observable with the observation space consisting of the base-environment observations, together with one-hot vectors indicating the agent's position in the controller environment, the currently selected base-environment action, and the remaining time steps until the next base-environment action is executed. Rewards are simply the rewards from the base environment and fixed to zero except on time steps when the base environment is advanced.

I use size 5 ElectricProcMaze as the base environment, reduced from size 7 to compensate for the added difficulty of having to control the environment indirectly. Most of the hyperparameters are maintained from the ElectricProcMaze experiment. I changed the option rollout length from 5 to 8 to match the size of the controller-environment grid, corresponding to one base-environment action per rollout. I also increased the capacity of the network as the original network worked poorly for both OptIt and ExIt in preliminary experiments indicating significant underparameterization.

For HierarchicalElectricProcMaze, I also reduced the entropy regularization factor β from 0.1 to 0.01 for Option Iteration which I found was necessary to obtain a performance benefit compared to ExIt. In light of this change, to facilitate a fair comparison, and a more complete picture of the behaviour of each approach, I performed a sweep over β values in powers of 10 and display the performance of each algorithm for the best β. The results of the sweep are available in Figure 5.5 in Appendix C.5. The results of this sweep reveal that OptIt achieves optimal performance at an order of magnitude lower level of entropy regularization than ExIt and remains more robust to further reduction in the regularization. One plausible interpretation of this result is that since OptIt is effectively learning a distribution of possible behaviours rather than fitting a single policy, it is able to maintain adequate behavioural diversity while fitting to a sharper search policy. This may in turn allow it to benefit from extracting more information from the search results without suffering as much from overfitting.

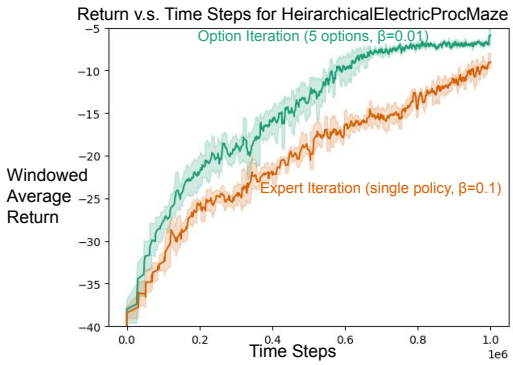

Figure 5.5: Windowed average return over training time for OptIt and baselines on HierarchicalElectricProcMaze with size 8 controller and size 5 base environment. Each algorithm is presented for its best β value from a sweep in powers of 10. Error bars show 95% confidence interval over 5 random seeds. The y-axis is thresholded at -40 to omit the rapid period of initial improvement. Running MCS in the space of options learned with OptIt provides a large benefit compared to using primitive actions evaluated under the single learned rollout policy.

The results are displayed in Figure 5.5. Planning with options discovered by OptIt shows a large benefit over using a single learned rollout policy.

Unlike the options learned for the Compass environment, I did not observe the options learned for HierarchicalElectricProcMaze to be particularly interpretable. Nevertheless, I did find the learned options to display significant behavioural diversity in an analysis which I present in Appendix C.3.

5.5 Discussion

This chapter introduced the third major contribution of this thesis, OptIt, which is also the third example of how our design decisions can influence a reinforcement agent's ability to take advantage of generic problem structure. In the case of OptIt, the exploited structure is the idea that optimal actions in temporally contiguous states tend to be highly interdependent. This in turn was postulated to be a consequence of spatial locality, whereby temporally contiguous states will tend to involve interaction with a highly overlapping set of subsystems in the world. Thus any information we gain about these subsystems will be likely to provide information about the optimal behaviour in

many such states. OptIt is a simple approach to discovering a set of options, which represent a richer distribution over trajectories for search than can be represented by a single policy, by amortizing the results of a computationally expensive search algorithm. The discovered options can then be used to improve future search, resulting in a virtuous cycle where better options lead to improved search, which in turn enables the discovery of better options. I demonstrated empirically that the options learned by OptIt provide a significant benefit over a single policy learned with ExIt when planning with MCS in challenging planning domains.

Chapter 6

Closing

In this dissertation, I presented three major contributions in the form of three examples of how generic problem structure can be exploited in RL. Generic problem structure here refers, open-endedly, to structure we expect to exist in a wide range of problems (e.g., action taken in the present does not influence the past), as opposed to structure which is highly specific to the problem at hand (e.g., heuristics or theorems about which moves are superior in a particular game). In Chapter 3, I demonstrated how model-based RL algorithms can make inherently better use of known problem structure compared to analogous model-free algorithms. In Chapter 4, I proposed the HNCA algorithm which exploits knowledge of the network structure to provide lower variance gradient estimates in networks of stochastic units. In Chapter 5, I proposed an approach to option discovery based on the intuitive idea that, as a consequence of spatial locality, the optimal actions for temporally contiguous states will tend to be highly interdependent.

In terms of the central question of this thesis

> Can reinforcement learning agents be designed to take advantage of generic problem structure to achieve more efficient learning and planning?

I believe the above three examples show that the answer is undoubtedly yes, but the question itself remains fertile ground for future work. Many of the fundamental advances in RL and machine learning more broadly can be seen as coming up with clever ways to exploit generic structure to improve sam-

ple efficiency and/or computational efficiency. Backpropagation exploits the fact that computational units are organized in a network with particular connectivity, and TD learning exploits the fact that the future is independent of the past given the present. However, despite this ubiquity, new approaches are often not framed in terms of what kind of structure they aim to exploit. This relates to the distinction between problem-focused and solution-focused research.

Problem-focused research looks first and foremost at the nature of the problem at hand, considers its structure, and then aims to tailor a solution method to the particular problem. Solution-focused research first decides on a particular class of solution methods and looks for problems where the method can be applied or asks how the methods can be improved. Of course, in practice, this distinction is not clear-cut. For example, neural networks are a particular solution method, but once one commits to using them one can see the question of how to train them as a problem to be solved, for example by backpropagation. Solution methods tend to give rise to their own unique problems.

Both problem-focused and solution-focused research can have significant value, improving established methods and looking for new places they can add value is certainly valid. However, it's important to realise that the problem and its underlying structure always comes before the solution. Every existing solution method was designed with a particular problem in mind. When faced with a new problem it's of course valid to consider the library of existing tools and think carefully about whether the problem is similar enough that they can be applied. However, when a problem is truly new one inevitably needs to fall back on considering the problem, and its underlying structure, directly to design a good solution.

I have often found problem-focused thinking to be a good base of operation to return to when challenges arise. It's easy to get focused on a particular solution method and lose sight of the problem we are ultimately trying to solve. Many times I have spent significant effort trying to make an algorithm work better only to realize that my underlying assumptions were just a little

bit off and upon carefully considering the problem at hand I should be doing things a little bit differently.

6.1 Future Directions

Aside from the general sense in which exploiting problem structure can be a valuable guide for choosing research directions, each specific contribution outlined in this thesis raises many interesting questions which could be addressed in future work.

Chapter 3 offers a strong foundation for understanding the benefits of model-based learning, but there remain many unanswered questions and significant work to be done to make the insights more actionable. For example

- Theorem 3.1 only directly addresses the case of deterministic transition dynamics, it is interesting to ask whether similarly simple insights can be made for the case of stochastic dynamics.
- Theorem 3.1 suggests that model-based methods should be preferred to model-free methods if the approaches begin with the same world knowledge. However, in practice, world knowledge is encoded somewhat nebulously, for example, in the choice of neural-network architecture. It would be interesting to look more closely at what kind of biases are implied by using a particular neural-network architecture in a model-based or model-free approach. One plausible reason for the empirical challenge of getting model-based learning methods working compared to model-free methods is that the inductive bias over MDPs implied by naively applying a neural network to learn a world model is less reasonable than when a similar network is applied to learn a value function. For example, feeding a neural network's output back into its input as is done with multi-step model rollouts will tend to result in instability, unless something is done to address it. In principle, a value function which generalizes pathologically could also lead to extreme results for examples that differ from those on which it is directly trained. However, this seems not to be a major problem for neural-network-based value

functions in practice. It would be interesting to better understand how the underlying inductive biases differ, towards improving them for the model-based approach.

- Theorem 3.1 also applies only to the realizable case where the true model is contained in the function class. It would be interesting to consider how a more nuanced analysis could be applied to the case where realizability does not hold. Such analysis should reveal a trade-off between model-based and model-free learning where the model-based approach becomes relatively better in the limit of more function approximation resources and relatively worse in the limit of more data.
- Along the same lines, it is interesting to consider how using some of the approaches for learning implicit models—which focus on modelling only task-relevant aspects of the future such as policy, value and reward—can help when realizability does not hold. I suspect an intermediate paradigm may ultimately be a better solution than either only learning task-relevant features, or attempting to model the full dynamics. One could for example attempt to model as much of the dynamics as possible given limited function approximation resources, but failing that fall back on first modelling things which are already known to be task-relevant. This could give rise to something like an exploration-exploitation trade-off for model learning.

Chapter 4 proposes a concrete approach with clearly established benefits for gradient estimation in networks of stochastic neurons. My original motivation for working on it however was partially related to the credit assignment problem in RL (see the survey of Pignatelli et al. (2023) for an overview of relevant work on this problem). To see the connection, one can imagine replacing the network of agents with some kind of factored world model and using an approach like HNCA to propagate credit for the agent's actions through the world model. I believe the key insights of HNCA could help develop methods to address the Credit Assignment Problem more generally. One key challenge which could be addressed as a step toward this would be extending HNCA to somehow propagate credit multiple steps through a network rather than just

a single step.

Chapter 5 provides a proof of concept for the approach of discovering options by amortizing the results of a computationally expensive search over multi-step trajectory segments. There are many possible directions to extend and improve this basic idea. For example

- Rather than fixing the horizon of the learned options one could learn termination functions for each option based by maximizing the likelihood of trajectories under the distribution induced by the combination of the option policies, policy over options and termination functions. In this case, the termination functions would aim to learn the optimal place to split the trajectory into segments such that in each segment the search policy was closely matched to a particular option. I discuss how this could be done in detail in Appendix C.4, building on the approach of Fox et al. (2017).
- One could explore applying OptIt in combination with more sophisticated search algorithms than MCS, such as MCTS.
- Rather than using a discrete set of options, one could specify options with an arbitrary latent variable. This would likely require more sophisticated techniques, like variational inference.
- OptIt inherently relies on the reward signal to allow the search procedure to identify good actions. This is limiting in domains with very sparse rewards, in which we might instead want to discover options that are useful for exploration even before the agent has managed to locate any reward. One simple way to address this would be to run OptIt in combination with an intrinsic reward signal.
- I encourage option diversity in OptIt only implicitly by maximizing trajectory likelihood under the ρ weighted mixture of option policies. It may be beneficial to explicitly encourage diversity, for example using a quality-diversity approach similar to that of Zahavy et al. (2023).
- On the more theoretical side, it would be interesting to analyze the convergence behaviour of OptIt. I motivated OptIt by arguing that a set of options can be a better representation of the true predictive

posterior over optimal actions than a single policy. However, the targets used to train OptIt were not samples from the optimal policy, which would not be practical to obtain, but rather samples from an improved policy derived from a search procedure using the current set of options. It would be interesting to consider how one could analytically investigate the convergence behaviour in this more realistic setting.

- Finally, OptIt evaluates options by explicitly rolling them out under the true world model. Ideally, we'd like to learn an option model to predict the mean reward and distribution of states that would result from executing each option, without actually having to simulate it. This would allow an increase in effective search depth for a given amount of computational effort at the cost of introducing model errors. In this case, one would still need a method for learning useful options, and I believe OptIt or something like it could be a reasonable approach. However, the addition of learned option models would raise interesting challenges. If options are iteratively improved as in OptIt, the option models would have to fit nonstationary targets. To address this, it may be desirable to jointly fit the options and models to each other such that the options are explicitly encouraged to remain predictable and not to deviate too quickly from the current model predictions. Furthermore, we would like to learn dynamics models for each option in the set while selecting actions according to yet another policy induced by the search procedure. This would mean dealing with the challenges of off-policy learning. However, the challenge may be mitigated somewhat if the option policies are themselves trained to match the search results as in OptIt. Overall, joint learning of options and the associated option models is an important, but challenging, area for future work.

6.2 Closing Thoughts: Where does Knowledge Come From?

The focus of this dissertation was on how incorporating basic knowledge of problem structure can improve RL systems. Incorporating some prior knowledge into our algorithms is necessary for progress. Without any prior knowledge, we have no real basis for making predictions in situations that differ in any way from those we have already seen as we can't know a priori the correct way to generalize from our observations. This is the essential idea behind various no free lunch theorems, such as those of Wolpert et al. (1997). Agents acting in the real world will never see the same situation twice, particularly if you incorporate history, hence this would leave us in a hopeless position.

While I believe including some basic knowledge is necessary to build systems which learn in any meaningful sense, at first glance, this almost seems like a chicken-and-egg problem. To learn anything meaningful, first, we have to already know something about how one situation is likely to generalise to another. So where do humans get the prior knowledge to incorporate into our learning systems? On a basic level, one could say that some of it is learned from life experience and some of it is hard-coded at birth as a result of evolution. But human learning and evolution can both be seen as algorithms for improving from experience. In the case of evolution, the improvement comes from the experience of many generations that came before us. In the presence of an infinite space of non-repeating situations, improving from experience is only possible given some prior knowledge about how one situation should generalize to another.

I think the resolution to this question is simply to realize that evolution itself does not begin with zero knowledge because it operates on a substrate derived from the universe for which it optimizes. In a sense, one could say that the initial seed of knowledge about the universe which facilitates learning about it comes from the universe itself. As an illustrative example, brains exist in three-dimensional space. This simple fact might bias them towards reasoning about three-dimensional space, which in turn is a useful bias for

reasoning about, and surviving in the world. Likewise, brains operate based on the same laws of physics which govern the rest of the world around them. These laws tend to be continuous and smooth in the sense that small changes to initial conditions result in small changes in short-term outcomes. This could bias them towards modelling smooth dynamics which again is likely to be useful in our universe.

As practitioners of machine learning, we aim to instill a minimal amount of our knowledge—derived from learning, evolution, and the inherent priors of being part of the universe—into machines to allow them to learn for themselves from their own experience of the world. Theorists distil knowledge by thinking deeply to formulate tractable problems which approximate some properties of the universe, and then try to solve them. Empiricists rely on intuition to guide their search through a large search space of possible solutions toward those which their universe-derived brain deems plausible to succeed. It's not clear how much prior knowledge we need to instil to enable meaningful learning on a reasonable time scale. Including the most basic priors can already make the difference between an impossible problem and merely a painfully slow process. Adding the right knowledge on top of this may result in significant additional improvements but it's too early to say what the right amount is for a practical generally intelligent system.

Taking neural networks as an example, the specific prior knowledge encoded depends a lot on the particular architecture. However, all neural networks are biased towards smoothly interpolating between nearby inputs. Given sufficient data, they can also develop arbitrarily sharp distinctions between inputs as appropriate but they tend towards smoothness by default. Probably the reason for the ubiquitous success of neural networks is not that they generalize particularly well, but that they generalize in the simplest way which still provides useful biases for realistic problems. Beyond that, they provide a convenient generic function approximation and by and large get out of the way and let the data speak for itself in a reasonably computationally efficient manner.

Overall, while we need to incorporate some structure into our agents, we do not necessarily need to incorporate a lot of structure, and decades of machine

learning research so far seem to reinforce the bitter lesson (Sutton, 2019) that less is usually better in the long run. With that in mind, the driving force behind the work that went into this dissertation was the search for the simplest aspects of problem structure which could provide a significant edge to a learning agent.

References

Agrawal, R. (1995). Sample mean based index policies by O(log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, *27*(4), 1054–1078.

Amin, S., Gomrokchi, M., Satija, H., van Hoof, H., & Precup, D. (2021). A survey of exploration methods in reinforcement learning. *arXiv preprint 2109.00157*.

Anand, A., Walker, J. C., Li, Y., Vértes, E., Schrittwieser, J., Ozair, S., Weber, T., & Hamrick, J. B. (2022). Procedural generalization by planning with self-supervised world models. *Proceedings of the International Conference on Learning Representations*.

Anthony, T., Nishihara, R., Moritz, P., Salimans, T., & Schulman, J. (2018). Policy gradient search: Online planning and expert iteration without search trees. *NeurIPS Deep Reinforcement Learning Workshop*.

Anthony, T., Tian, Z., & Barber, D. (2017). Thinking fast and slow with deep learning and tree search. *Advances in Neural Information Processing Systems*, *30*, 5366–5376.

Asadi, K. (2015). Strengths, weaknesses, and combinations of model-based and model-free reinforcement learning. *Department of Computing Science University of Alberta (Masters Thesis)*.

Bacon, P.-L., Harb, J., & Precup, D. (2017). The option-critic architecture. *Proceedings of the AAAI Conference on Artificial Intelligence*, 1726–1734.

Barreto, A., Dabney, W., Munos, R., Hunt, J. J., Schaul, T., van Hasselt, H. P., & Silver, D. (2017). Successor features for transfer in reinforcement learning. *Advances in Neural Information Processing Systems*, *30*, 4055–4065.

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

Bellman, R. (1957). *Dynamic programming*. Princeton University Press.

Bengio, Y., Léonard, N., & Courville, A. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint 1308.3432*.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne,

S., & Zhang, Q. (2018). JAX: Composable transformations of Python+NumPy programs. http://github.com/google/jax.

Buckman, J., Hafner, D., Tucker, G., Brevdo, E., & Lee, H. (2018). Sample-efficient reinforcement learning with stochastic ensemble value expansion. *Advances in Neural Information Processing Systems*, *31*, 8224–8234.

Buesing, L., Weber, T., Zwols, Y., Racaniere, S., Guez, A., Lespiau, J.-B., & Heess, N. (2019). Woulda, coulda, shoulda: Counterfactually-guided policy search. *Proceedings of the International Conference on Learning Representations*.

Burda, Y., Grosse, R., & Salakhutdinov, R. (2015). Importance weighted autoencoders. *arXiv preprint 1509.00519*.

Byravan, A., Hasenclever, L., Trochim, P., Mirza, M., Ialongo, A. D., Tassa, Y., Springenberg, J. T., Abdolmaleki, A., Heess, N., Merel, J., et al. (2022). Evaluating model-based planning and planner amortization for continuous control. *Proceedings of the International Conference on Learning Representations*.

Chapelle, O., Schölkopf, B., & Zien, A. (2006). *Semi-supervised learning*. The MIT Press.

Chua, K., Calandra, R., McAllister, R., & Levine, S. (2018). Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *Advances in Neural Information Processing Systems*, *31*, 4754–4765.

Co-Reyes, J., Liu, Y., Gupta, A., Eysenbach, B., Abbeel, P., & Levine, S. (2018). Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings. *Proceedings of the International Conference on Machine Learning*, 1008–1017.

Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. *Proceedings of the International Conference on Computers and Games*, 72–83.

Curi, S., Berkenkamp, F., & Krause, A. (2020). Efficient model-based reinforcement learning through optimistic policy search and planning. *Advances in Neural Information Processing Systems*, *33*, 14156–14170.

Danihelka, I., Guez, A., Schrittwieser, J., & Silver, D. (2021). Policy improvement by planning with gumbel. *Proceedings of the International Conference on Learning Representations*.

Dayan, P., & Hinton, G. E. (1992). Feudal reinforcement learning. *Advances in Neural Information Processing Systems*, *5*, 271–278.

Deisenroth, M., & Rasmussen, C. E. (2011). PILCO: A model-based and data-efficient approach to policy search. *Proceedings of the International Conference on Machine Learning*, 465–472.

Diuk, C., Li, L., & Leffler, B. R. (2009). The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 249–256.

Dong, K., Luo, Y., Yu, T., Finn, C., & Ma, T. (2020). On the expressivity of neural networks for deep reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 2847–2854.

Dong, Z., Mnih, A., & Tucker, G. (2020). DisARM: An antithetic gradient estimator for binary latent variables. *Advances in Neural Information Processing Systems*, *33*, 18637–18647.

Eysenbach, B., Gupta, A., Ibarz, J., & Levine, S. (2019). Diversity is all you need: Learning skills without a reward function. *Proceedings of the International Conference on Learning Representations*.

Fox, R., Krishnan, S., Stoica, I., & Goldberg, K. (2017). Multi-level discovery of deep options. *arXiv preprint 1703.08294*.

Frans, K., Ho, J., Chen, X., Abbeel, P., & Schulman, J. (2018). Meta learning shared hierarchies. *Proceedings of the International Conference on Learning Representations*.

Gehring, C., Kawaguchi, K., Huang, J., & Kaelbling, L. (2021). Understanding end-to-end model-based reinforcement learning methods as implicit parameterization. *Advances in Neural Information Processing Systems*, *34*, 703–714.

Gelly, S., & Silver, D. (2007). Combining online and offline knowledge in UCT. *Proceedings of the International Conference on Machine Learning*, 273–280.

Grathwohl, W., Choi, D., Wu, Y., Roeder, G., & Duvenaud, D. (2018). Backpropagation through the void: Optimizing control variates for black-box gradient estimation. *Proceedings of the International Conference on Learning Representations*.

Gregor, K., Jimenez Rezende, D., Besse, F., Wu, Y., Merzic, H., & van den Oord, A. (2019). Shaping belief states with generative environment models for RL. *Advances in Neural Information Processing Systems*, *32*, 13475–13487.

Gregor, K., Rezende, D. J., & Wierstra, D. (2016). Variational intrinsic control. *arXiv preprint 1611.07507*.

Gu, S., Levine, S., Sutskever, I., & Mnih, A. (2018). MuProp: Unbiased backpropagation for stochastic neural networks. *Proceedings of the International Conference on Learning Representations*.

Ha, D., & Schmidhuber, J. (2018). World models. *Advances in Neural Information Processing Systems*, *31*, 2450–2462.

Hafner, D., Lee, K.-H., Fischer, I., & Abbeel, P. (2022). Deep hierarchical planning from pixels. *arXiv preprint 2206.04114*.

Hafner, D., Lillicrap, T., Ba, J., & Norouzi, M. (2020). Dream to control: Learning behaviors by latent imagination. *Proceedings of the International Conference on Learning Representations*.

Hafner, D., Lillicrap, T., Norouzi, M., & Ba, J. (2021). Mastering Atari with discrete world models. *Proceedings of the International Conference on Learning Representations*.

Harutyunyan, A., Dabney, W., Borsa, D., Heess, N., Munos, R., & Precup, D. (2019). The termination critic. *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2231–2240.

Harutyunyan, A., Dabney, W., Mesnard, T., Azar, M. G., Piot, B., Heess, N., van Hasselt, H. P., Wayne, G., Singh, S., Precup, D., et al. (2019). Hindsight credit assignment. *Advances in Neural Information Processing Systems*, *32*, 12488–12497.

Heess, N., Wayne, G., Silver, D., Lillicrap, T., Erez, T., & Tassa, Y. (2015). Learning continuous control policies by stochastic value gradients. *Advances in Neural Information Processing Systems*, *28*, 2944–2952.

Hessel, M., Danihelka, I., Viola, F., Guez, A., Schmitt, S., Sifre, L., Weber, T., Silver, D., & van Hasselt, H. (2021). Muesli: Combining improvements in policy optimization. *Proceedings of the International Conference on Machine Learning*, 4214–4226.

Holland, G. Z., Talvitie, E. J., & Bowling, M. (2018). The effect of planning shape on Dyna-style planning in high-dimensional state spaces. *arXiv preprint 1806.01825*.

Jang, E., Gu, S., & Poole, B. (2017). Categorical reparameterization with gumbel-softmax. *Proceedings of the International Conference on Learning Represenations*.

Janner, M., Fu, J., Zhang, M., & Levine, S. (2019). When to trust your model: Model-based policy optimization. *Advances in Neural Information Processing Systems*, *32*, 12519–12530.

Jinnai, Y., Abel, D., Hershkowitz, D., Littman, M., & Konidaris, G. (2019). Finding options that minimize planning time. *Proceedings of the International Conference on Machine Learning*, 3120–3129.

Jordan, M. I. (1988). *Supervised learning and systems with excess degrees of freedom* (technical report COINS TR 88-27). Massachusetts Institute of Technology.

Kaiser, L., Babaeizadeh, M., Milos, P., Osinski, B., Campbell, R. H., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S., et al. (2020). Model-based reinforcement learning for Atari. *Proceedings of the International Conference on Learning Representations*.

Karnin, Z., Koren, T., & Somekh, O. (2013). Almost optimal exploration in multi-armed bandits. *Proceedings of the International Conference on Machine Learning*, 1238–1246.

Katehakis, M. N., & Robbins, H. (1995). Sequential choice from several populations. *Proceedings of the National Academy of Sciences*, *92*(19), 8584–8585.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *Proceedings of the International Conference on Learning Representations*.

Kingma, D. P., & Welling, M. (2014). Auto-encoding variational bayes. *Proceedings of the International Conference on Learning Representations*.

Kipf, T., Li, Y., Dai, H., Zambaldi, V., Sanchez-Gonzalez, A., Grefenstette, E., Kohli, P., & Battaglia, P. (2019). CompILE: Compositional imitation learning and execution. *Proceedings of the International Conference on Machine Learning*, 3418–3428.

Klissarov, M., & Machado, M. C. (2023). Deep laplacian-based options for temporally-extended exploration. *Proceedings of the International Conference on Machine Learning*, 17198–17217.

Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. *Proceedings of the European conference on machine learning*, 282–293.

Kool, W., van Hoof, H., & Welling, M. (2019). Buy 4 reinforce samples, get a baseline for free! *ICLR Deep Reinforcement Learning Meets Structured Prediction Workshop*.

Kostas, J., Nota, C., & Thomas, P. (2020). Asynchronous coagent networks. *Proceedings of the International Conference on Machine Learning*, 5426–5435.

Krishnan, S., Fox, R., Stoica, I., & Goldberg, K. (2017). DDCO: Discovery of deep continuous options for robot learning from demonstrations. *Proceedings of the Conference on Robot Learning*, 418–437.

Kudashkina, K., Wan, Y., Naik, A., & Sutton, R. S. (2021). Planning with expectation models for control. *arXiv preprint 2104.08543*.

Laidlaw, C., Russell, S., & Dragan, A. (2023). Bridging RL theory and practice with the effective horizon. *Advances in Neural Information Processing Systems*.

LeCun, Y., Cortes, C., & Burges, C. (2010). Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*.

Lin, C.-C., Jaech, A., Li, X., Gormley, M. R., & Eisner, J. (2020). Limitations of autoregressive models and their alternatives. *arXiv preprint 2010.11939*.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, *8*(3), 293–321.

Lin, L.-J., & Mitchell, T. (1992). *Memory approaches to reinforcement learning in non-markovian domains* (technical report CMU-CS-92-138). Carnegie-Mellon University.

Machado, M. C., Bellemare, M. G., & Bowling, M. (2017). A laplacian framework for option discovery in reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 2295–2304.

Maddison, C. J., Mnih, A., & Teh, Y. W. (2017). The concrete distribution: A continuous relaxation of discrete random variables. *Proceedings of the International Conference on Learning Represenations*.

McGovern, A., & Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. *Proceedings of the International Conference on Machine Learning*, 361–368.

Mnih, A., & Gregor, K. (2014). Neural variational inference and learning in belief networks. *Proceedings of the International Conference on Machine Learning*, 1791–1799.

Mnih, A., & Rezende, D. J. (2016). Variational inference for monte carlo objectives. *Proceedings of the International Conference on Machine Learning*, 2188–2196.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Moerland, T. M., Broekens, J., & Jonker, C. M. (2020). Model-based reinforcement learning: A survey. *arXiv preprint 2006.16712*.

Munro, P. W. (1987). A dual back-propagation scheme for scalar reinforcement learning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, 165–176.

Naik, A., Shariff, R., Yasui, N., Yao, H., & Sutton, R. S. (2019). Discounted reinforcement learning is not an optimization problem. *NeurIPS Optimization Foundations for Reinforcement Learning Workshop*.

Ng, A., & Jordan, M. (2001). On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. *Advances in Neural Information Processing Systems*, *14*, 841–848.

Osband, I., Russo, D., & Van Roy, B. (2013). (More) efficient reinforcement learning via posterior sampling. *Advances in Neural Information Processing Systems*, *26*, 3003–3011.

Osband, I., & Van Roy, B. (2014). Near-optimal reinforcement learning in factored MDPs. *Advances in Neural Information Processing Systems*, *27*, 604–612.

Osband, I., Wen, Z., Asghari, S. M., Dwaracherla, V., Ibrahimi, M., Lu, X., & Van Roy, B. (2021). Epistemic neural networks. *arXiv preprint 2107.08924*.

Pan, Y., Zaheer, M., White, A., Patterson, A., & White, M. (2018). Organizing experience: A deeper look at replay mechanisms for sample-based planning in continuous state domains. *Proceedings of the International Joint Conference on Artificial Intelligence*, 4794–4800.

Parr, R., Li, L., Taylor, G., Painter-Wakefield, C., & Littman, M. L. (2008). An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 752–759.

Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2778–2787.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. Morgan Kaufmann.

Pignatelli, E., Ferret, J., Geist, M., Mesnard, T., van Hasselt, H., & Toni, L. (2023). A survey of temporal credit assignment in deep reinforcement learning. *arXiv preprint 2312.01072*.

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., & Sutskever, I. (2020). Zero-shot text-to-image generation. *Proceedings of the International Conference on Machine Learning*, 8821–8831.

Rezende, D. J., Mohamed, S., & Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. *Proceedings of the International Conference on Machine Learning*, 1278–1286.

Richalet, J., Rault, A., Testud, J., & Papon, J. (1978). Model predictive heuristic control: Applications to industrial processes. *Automatica*, *14*(5), 413–428.

Rosin, C. D. (2011). Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, *61*(3), 203–230.

Ross, S., & Bagnell, J. A. (2012). Agnostic system identification for model-based reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 1905–1912.

Sallans, B., & Hinton, G. E. (2004). Reinforcement learning with factored states and actions. *The Journal of Machine Learning Research*, *5*, 1063–1088.

Schmidhuber, J. (1990). *Making the world differentiable: On using fully recurrent self-supervised neural networks for dynamic reinforcement learning and planning in non-stationary environments* (technical report FKI-126-90). Institut für Informatik, Technische Universität München.

Schmidhuber, J. (1991a). Curious model-building control systems. *Proceedings of the International Joint Conference on Neural Networks*, *2*, 1458–1463.

Schmidhuber, J. (1991b). A possibility for implementing curiosity and boredom in model-building neural controllers. *Proceedings of the Conference on Simulation of Adaptive Behavior: From Animals to Animats*, 222–227.

Schmidhuber, J. (1997). *What's interesting?* (technical report IDSIA-35-97). IDSIA.

Schmidhuber, J. (2010). Formal theory of creativity, fun, and intrinsic motivation. *IEEE Transactions on Autonomous Mental Development*, *2*(3), 230–247.

Schmidhuber, J. (2015). On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *Preprint arXiv:1511.09249*.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. (2020). Mastering Atari, go, chess and shogi by planning with a learned model. *Nature*, *588*(7839), 604–609.

Shankar, T., & Gupta, A. (2020). Learning robot skills with temporal variational inference. *Proceedings of the International Conference on Machine Learning*, 8624–8633.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint 1712.01815*.

Silver, D., Sutton, R. S., & Müller, M. (2012). Temporal-difference search in computer Go. *Machine Learning*, *87*(2), 183–219.

Stolle, M., & Precup, D. (2002). Learning options in reinforcement learning. *Proceedings of the International Symposium on Abstraction, Reformulation, and Approximation*, 212–223.

Strehl, A. L., Diuk, C., & Littman, M. L. (2007). Efficient structure learning in factored-state MDPs. *Proceedings of the AAAI Conference on Artificial Intelligence*, *7*, 645–650.

Strens, M. (2000). A bayesian framework for reinforcement learning. *Proceedings of the International Conference on Machine Learning*, *2000*, 943–950.

Sun, W., Jiang, N., Krishnamurthy, A., Agarwal, A., & Langford, J. (2019). Model-based RL in contextual decision processes: PAC bounds and exponential improvements over model-free approaches. *Proceedings of the Conference on learning theory*, 2898–2933.

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

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the International Conference on Machine Learning*, 216–224.

Sutton, R. S. (2019). The bitter lesson. *Incomplete Ideas (blog)*.

Sutton, R. S., & Barto, A. G. (2020). *Reinforcement learning: An introduction* (Second Edition). MIT press.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, *112*(1-2), 181–211.

Sutton, R. S., Szepesvári, C., Geramifard, A., & Bowling, M. P. (2008). Dyna-style planning with linear function approximation and prioritized sweeping. *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 528–536.

Tang, C., & Salakhutdinov, R. R. (2013). Learning stochastic feedforward neural networks. *Advances in Neural Information Processing Systems*, *26*, 530–538.

Tesauro, G., & Galperin, G. (1996). On-line policy improvement using monte-carlo search. *Advances in Neural Information Processing Systems*, *9*, 1068–1074.

Thomas, P. S., & Barto, A. G. (2011). Conjugate markov decision processes. *Proceedings of the International Conference on Machine Learning*, 137–144.

Thrun, S. (1992). *Efficient exploration in reinforcement learning* (technical report CMU-CS-92-102). Carnegie-Mellon University.

Titsias, M. K., & Lázaro-Gredilla, M. (2015). Local expectation gradients for black box variational inference. *Advances in Neural Information Processing Systems*, *28*, 2638–2646.

Tucker, G., Mnih, A., Maddison, C. J., Lawson, J., & Sohl-Dickstein, J. (2017). REBAR: Low-variance, unbiased gradient estimates for discrete latent variable models. *Advances in Neural Information Processing Systems*, *30*, 2624–2633.

van Hasselt, H. P., Hessel, M., & Aslanides, J. (2019). When to use parametric models in reinforcement learning? *Advances in Neural Information Processing Systems*, *32*, 14322–14333.

van Seijen, H., & Sutton, R. S. (2015). A deeper look at planning as learning from replay. *Proceedings of the International Conference on Machine Learning*, 2314–2322.

Veeriah, V., Zahavy, T., Hessel, M., Xu, Z., Oh, J., Kemaev, I., van Hasselt, H. P., Silver, D., & Singh, S. (2021). Discovery of options via meta-learned subgoals. *Advances in Neural Information Processing Systems*, *34*, 29861–29873.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., & Kavukcuoglu, K. (2017). Feudal networks for hierarchical reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 3540–3549.

Wan, Y., Abbas, Z., White, A., White, M., & Sutton, R. S. (2019). Planning with expectation models. *Proceedings of the International Joint Conference on Artificial Intelligence*, 3649–3655.

Wan, Y., Naik, A., & Sutton, R. S. (2021). Learning and planning in average-reward markov decision processes. *Proceedings of the International Conference on Machine Learning*, 10653–1066.

Wan, Y., & Sutton, R. S. (2022). Toward discovering options that achieve faster planning. *arXiv preprint 2205.12515*.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

Watter, M., Springenberg, J., Boedecker, J., & Riedmiller, M. (2015). Embed to control: A locally linear latent dynamics model for control from raw images. *Advances in Neural Information Processing Systems*, *28*, 2746–2754.

Werbos, P. J. (1987). Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, *17*, 7–20.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*(3-4), 229–256.

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, *1*(1), 67–82.

Xu, Z., & Tewari, A. (2020). Reinforcement learning in factored MDPs: Oracle-efficient algorithms and tighter regret bounds for the non-episodic set-

ting. *Advances in Neural Information Processing Systems*, *33*, 18226–18236.

Ye, W., Liu, S., Kurutach, T., Abbeel, P., & Gao, Y. (2021). Mastering Atari games with limited data. *Advances in Neural Information Processing Systems*, *34*, 25476–25488.

Yin, M., & Zhou, M. (2019). Arm: Augment-reinforce-merge gradient for stochastic binary networks. *Proceedings of the International Conference on Learning Representations*.

Young, K. (2022). Hindsight network credit assignment: Efficient credit assignment in networks of discrete stochastic units. *Proceedings of the AAAI Conference on Artificial Intelligence*, 8919–8926.

Young, K., Ramesh, A., Kirsch, L., & Schmidhuber, J. (2023). The benefits of model-based generalization in reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 40254–40276.

Young, K., & Sutton, R. S. (2023). Iterative option discovery for planning, by planning. *arXiv preprint 2310.01569*.

Zahavy, T., Veeriah, V., Hou, S., Waugh, K., Lai, M., Leurent, E., Tomasev, N., Schut, L., Hassabis, D., & Singh, S. (2023). Diversifying ai: Towards creative chess with alphazero. *arXiv preprint 2308.09175*.

Zhu, Y., Stone, P., & Zhu, Y. (2022). Bottom-up skill discovery from unsegmented demonstrations for long-horizon robot manipulation. *IEEE Robotics and Automation Letters*, *7*(2), 4126–4133.

Appendix A

Appendices for the Benefits of Model-Based Generalization

A.1 Theorems Motivating the Benefit of Model Generalization

Here, I present and prove simple theorems motivating the benefit of learning a parametric model over learning a value function directly from ER. Intuitively speaking, the first theorem states that, when narrowing down the set of possible value functions based on observed data, we can rule out more if we first rule out models directly, and demand the value function be consistent with the reduced model class, than if we only demand the value function obeys the Bellman optimality equation with respect to observed transitions.

I state the theorems within the formalism of finite MDPs. That is MDPs with finite state space $\mathcal{S}$ and action space $\mathcal{A}$. Recall from Section 2.1 that, in general the transition distribution p maps state-actions pairs to probability distributions over possible next states. However, here, I consider deterministic MDPs, meaning each state-action pair maps to a distribution with probability one on a particular next state s' and zero for all other next states. In this case, it will be convenient to write $p : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ as a mapping from state-action pairs to the only possible next state.

I focus here on deterministic MDPs, but similar results likely hold for general MDPs, albeit significantly complicated by the fact that in the general case, models and value functions can only be ruled out with high probability

based on observed data, as opposed to with certainty.

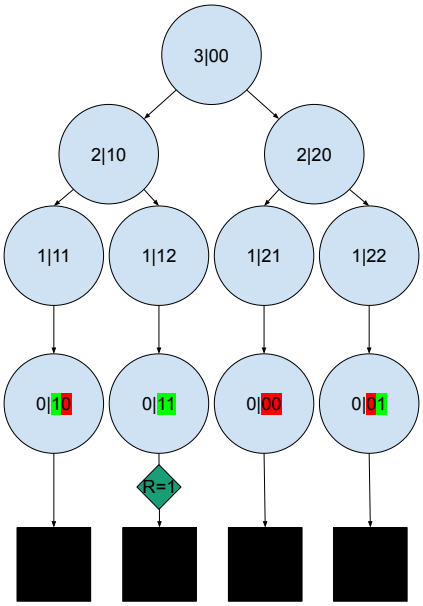

Figure A.1: Illustration of the model class used as an example where selecting a hypothesis based on model consistency is arbitrarily more selective than Bellman consistency. The action space consists of 2 actions, left and right. The states consist of 1 component which acts as a countdown to termination along with M *passcode* components (with $M = 2$ in the figure) which act to record the action sequence executed so far. The digits take values in $\{0, 1, 2\}$ with 1 indicating the left action, 2 indicating the right action and 0 padding the future actions. When the countdown reaches 0 the correct passcode digits will switch to 1 and the incorrect digits to 0. In the following step, the episode will terminate. Note that termination always occurs at $t = M + 2$. A reward of 1 will be given on termination if and only if all passcode digits are 1, and the reward is otherwise 0. Different models in the class vary only in the initially unknown passcode.

Theorem 3.1. *Consider a class of episodic MDPs, $\mathcal{M}$, with fixed reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and deterministic transition function belonging to a hypothesis class $H \subseteq \{p : \mathcal{S} \times \mathcal{A} \to S\}$. Assume, for all $p \in H$, all policies lead to eventual termination.*

Define the class of optimal action-value functions associated with H:

$$H_Q = \{q : s, a \to \mathbb{R} \,|\exists p \in H : \forall s, a \; q(s, a) = r(s, a) + \max_{a'} q(p(s, a), a') \wedge q(\perp, a) = 0\}. \quad \text{(A.1)}$$

Consider a dataset $D = \{(s_n, a_n, s'_n) | n \in \{0, 1, .., N\}\}$ *of transitions such that* $p(s_n, a_n) = s'_n$ *for some* $p \in H$*. Now, define two different notions of hypothesis classes over action-value functions which are consistent with* D*:*

$$H_B(D) = \{q \in H_Q | \forall n \; q(s_n, a_n) = r(s_n, a_n) + \max_{a'} q(s'_n, a')\}$$
$$H_M(D) = \{q \in H_Q | \exists p \in H : (\forall n \; p(s_n, a_n) = s'_n) \wedge (\forall s, a \; q(s, a) = r(s, a) + \max_{a'} q(p(s, a), a'))\}$$

Where B *stands for Bellman consistency and* M *stands for model consistency. In words, these are the hypothesis classes consisting of value functions which obey the Bellman optimality equation with respect to the observed transitions, and the hypothesis class consists of true optimal value functions for transition dynamics which are consistent with the observed transitions respectively. Then the following are true:*

1. $H_M(D) \subseteq H_B(D)$*.*
2. *For any* $N \in \mathbb{N}$ *there exists some choices of* $\mathcal{M}$ *and* D *such that* $\frac{|H_B(D)|}{|H_M(D)|} > N$*.*
3. *For a tabular transition function class, that is one that includes every possible mapping from state-action pairs to next states,* $H_M(D)$=$H_B(D)$*.*

Proof. I begin by proving **part 1**. Towards this, assume $q \in H_M(D)$. Then by definition, we have the following:

$$\begin{aligned}
&\exists p \in H : (\forall n \; p(s_n, a_n) = s'_n) \wedge (\forall s, a \; q(s, a) = r(s, a) + \max_{a'} q(p(s, a), a')) \\
\implies &\exists p \in H : (\forall n \; p(s_n, a_n) = s'_n) \wedge (\forall n \; q(s_n, a_n) = r(s_n, a_n) + \max_{a'} q(p(s_n, a_n), a')) \\
\implies &\forall n \; q(s_n, a_n) = r(s_n, a_n) + \max_{a'} q(s'_n, a') \\
\implies &q \in H_B(D),
\end{aligned}$$

which proves part 1.

To prove **part 2**, it suffices to construct a specific $\mathcal{M}$ and D for which $\frac{|H_M(D)|}{|H_B(D)|} > N$. Towards this, consider the model class illustrated in Figure A.1 with dynamics described in the caption. Now consider a dataset D consisting of a single episodic trajectory which terminates with the incorrect passcode, and thus 0 reward. As a concrete example, consider the following possible

sequence of transitions which may be observed for the problem instance in Figure 3.2:

$$(s_0 = 3|00, a_0 = 1, s'_0 = 2|10)$$
$$(s_1 = 2|10, a_1 = 2, s'_1 = 1|12)$$
$$(s_2 = 1|12, a_2 = 1, s'_2 = 0|00)$$
$$(s_3 = 0|00, a_3 = 2, s'_3 = \perp)$$

Note that the transition $(s_2 = 1|12, a_2 = 1, s'_2 = 0|00)$ indicates that both of the previous actions were incorrect. This transition can only ever be observed within the model class for the MDP for which the true passcode is 21. Thus observing this transition alone narrows $H_M(D)$ down to a singleton containing only the transition function associated with the passcode 21. More generally, for any M and any instance of the model class illustrated in Figure 2 with passcode of length M, it is true that for a dataset D consisting of any single episodic trajectory, the transition at time $t = M$ is sufficient to uniquely specify the passcode and thus narrow $H_M(D)$ down to a single element.

Next, consider $H_B(D)$ in the case where D consists of a single episodic trajectory which terminates with the incorrect passcode, and thus 0 reward. We can easily show that the optimal action-value function $\hat{q}$ for every MDP in the problem class *except* the one with passcode matching the final transition (12 in the above example) remains in $H_B(D)$. To see this, consider the optimal action-value function $\hat{q}$ for some arbitrary passcode different from the one that was entered in the trajectory contained in D. By assumption, we have that $\hat{q} \in H_Q$, hence to show that $\hat{q} \in H_B(D)$ we just need to show that $\hat{q}(s_n, a_n) = r(s_n, a_n) + \max_{a'} q(s'_n, a')$ for each n indexing the elements of D. Since, by assumption, D does not contain the single rewarding transition, $r(s_n, a_n) = 0$ for all n. Moreover, since $\hat{q}$ is assumed to be the value function for a passcode distinct from the one that was entered, we can let $\tilde{n}$ be the first integer such that $a_{\tilde{n}}$ differs from the passcode associated with $\hat{q}$, that is the first integer such that $a_{\tilde{n}} \notin \operatorname{argmax}_{a'} \hat{q}(s'_n, a')$. For $n < \tilde{n}$ we have $\hat{q}(s_n, a_n) = \max_{a'} q(s'_n, a') =$

1 since the actions up to that point are optimal. For $n \geq \tilde{n}$, $\hat{q}(s_n, a_n) = \max_{a'} q(s'_n, a') = 0$ since after the first suboptimal action, all paths lead to zero reward. Thus indeed $\hat{q} \in H_B(D)$ for all $\hat{q}$ associated with passcodes besides the one that was selected for the trajectory in D. This is a set of size $2^M - 1$, hence recalling that $|H_M(D)| = 1$, $|H_B(D)| = 2^M - 1 = (2^M - 1)|H_M(D)|$. Selecting passcode length $M \geq \frac{\log(N+1)}{\log(2)}$ satisfies the requirement of part 2, thus part 2 is proven.

Finally, I prove **part 3**. Since we already know from part 1 that $q \in H_M(D) \implies q \in H_B(D)$ in the general case, it suffices to show that, with the additional restriction of a tabular class H of transition matrices, we have $q \in H_B(D) \implies q \in H_M(D)$. Recall that by a tabular H we mean one which includes every possible mapping from state-action pairs to next states. Thus, in this case, we are free to choose $p(s, a)$ to be an independently selected s' for every (s, a) pair, and know that the resulting $p \in H$. Assume $q \in H_B(D)$, then by definition we know $q \in H_Q$, meaning

$$\exists p \in H : \forall s, a \; q(s, a) = r(s, a) + \max_{a'} q(p(s, a), a'), \tag{A.2}$$

and, by the definition of $q \in H_B(D)$, we know

$$\forall n \; q(s_n, a_n) = r(s_n, a_n) + \max_{a'} q(s'_n, a'). \tag{A.3}$$

Now, given tabular H we can set, for all n, $p(s_n, a_n) = s'_n$, which we know from Equation A.3 gives us:

$$q(s_n, a_n) = r(s_n, a_n) + \max_{a'} q(p(s_n, a_n), a').$$

Now for s, a where $\nexists n : (s, a) = (s_n, a_n)$, given Equation A.2, we know that for some choice of $p(s, a)$ it holds that:

$$q(s, a) = r(s, a) + \max_{a'} q(p(s, a), a').$$

Thus, given the choice of tabular H, we can choose p to simultaneously satisfy $p(s_n, a_n) = s'_n$ for all n and the Bellman optimality equation for all (s, a), and

indeed:

$$
\begin{aligned}
& q \in H_B(D) \\
& \implies \exists p \in H : (\forall n \; p(s_n, a_n) = s'_n) \wedge (\forall s, a \; q(s,a) = r(s,a) + \max_{a'} q(p(s,a), a')) \\
& \implies q \in H_M(D),
\end{aligned}
$$

which completes the proof of part 3. □

Practical learning algorithms like stochastic gradient descent applied with neural network function approximation don't work by directly ruling out hypotheses based on the data. Rather than considering models within an explicitly defined class, a neural network would have some inductive bias towards particular models and away from others depending on the architecture and hyperparameters. Nonetheless, the idea behind Theorem 3.1 helps to give insight into why we should expect learning a parametric model to provide a sample efficiency benefit.

Intuitively, we can think of the process of performing many updates to a sufficiently high-capacity neural network, with data from a dataset, as incrementally constraining the possible functions represented by the network. The specific function converged to will depend on the initialization and random batches selected for updates. Theorem 3.1 suggests that learning a model as an intermediate step can impose more constraints on the possible value functions. This in turn should increase the chance of converging to a value function that generalizes well from limited data.

I next provide the proof of Theorem 3.2. This theorem highlights that learning a model doesn't only allow us to narrow down the possible optimal action-value functions faster; in addition, forcing an agent to make decisions based only on the value information that can be determined using Bellman consistency can mean it takes an arbitrary factor more samples to achieve reasonable performance.

Theorem 3.2. *Consider a class of episodic MDPs, $\mathcal{M}$, with fixed reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and deterministic transition function belonging to a*

hypothesis class $H \subseteq \{p : \mathcal{S} \times \mathcal{A} \to S\}$. *Assume, for all* $p \in H$, *all policies lead to eventual termination.*

Define the class of optimal action-value functions associated with H*:*

$$H_Q = \{q : s, a \to \mathbb{R} \,|\exists p \in H : \forall s, a \; q(s,a) = r(s,a) + \max_{a'} q(p(s,a), a')\}.$$

Assume an agent is able to interact with the MDP in an online fashion selecting actions at each time step. Let $D_t = \{(s_n, a_n, s'_n)|n \in \{0, 1, .., t\}\}$ *be the dataset of all transitions observed up to time* t *during this interaction.*

Next, define two different notions of hypothesis classes over action-value functions which are consistent with D_t*:*

$$\begin{aligned} H_B(D_t) &= \{q \in H_Q | \forall n \; q(s_n, a_n) = r(s_n, a_n) + \max_{a'} q(s'_n, a')\} \\ H_M(D_t) &= \{q \in H_Q | \exists p \in H : (\forall n \; p(s_n, a_n) = s'_n) \wedge \\ &\qquad (\forall s, a \; q(s,a) = r(s,a) + \max_{a'} q(p(s,a), a'))\} \end{aligned}$$

Where B *stands for Bellman consistency and* M *stands for model consistency. For any* $\delta \in (0, 1]$ *and* $N \in \mathbb{N}$*, there exists* $\mathcal{M}$ *such that the following are true:*

1. *For any agent which selects actions at each time* t *based only on* $H_B(D_t)$ *and the current state* S_t*, there is some MDP in* $\mathcal{M}$ *such that with probability at least* $1 - \delta$ *the return for all episodes up to episode* N *will be* 0*.*
2. *There exists an agent which selects actions at each time* t *based on* $H_M(D_t)$ *and* S_t *such that the return for all episodes after the first is guaranteed to be* 1*, which is optimal.*

Proof. The proof of both part 1 and part 2 once again uses the MDP class illustrated in Figure 3.2. Note that at the beginning of the agent-environment interaction $D_t = \emptyset$ and $H_B(D_t) = H_M(D_t) = H_Q$. H_Q has one element for each passcode and thus $|H_Q| = 2^M$.

To prove **part 2**, simply note that after a single episode of experience, with any action sequence, the transition at time $t = M$ suffices to uniquely determine the passcode. Thus after a single episode of experience $|H_M(D_t)| = 1$. Likewise, by observing the singleton value function in $H_M(D_t)$ after one

episode an agent can uniquely determine the passcode associated with the environment instance and guarantee the optimal return of 1 by following the associated policy from that point forward.

To prove part **part 1**, begin by noting that except when the agent executes the action sequence associated with the passcode, $|H_B(D_t)|$ is reduced by at most 1 for each additional episode of experience. Specifically, if the agent executes an action sequence corresponding to a passcode that hasn't been previously tired, the optimal action-value function associated with that passcode will be removed from the set. Note that the resulting change will be identical for all MDPs in $\mathcal{M}$ except the one for which the action sequence matched the passcode. Thus, until the correct passcode has been tried, the information contained in $H_B(D_t)$ is the same as a list of passcodes which have not yet been tried. As soon as the correct passcode is tried once, $H_B(D_t)$ will be reduced to a single item corresponding to the correct passcode.

Whatever (potentially stochastic) strategy the model-free agent employs for action selection induces a distribution over sequences of passcodes to attempt assuming all attempts up to that point were failures. Regardless of the strategy employed, there will be some passcode such that after N episodes the probability of having ever sampled it will be no more than $\frac{N}{2^M}$. This is a simple consequence of the fact that otherwise, the expected number of passcodes tried after N episodes would have to be more than N, but it is only possible to attempt one passcode per episode.

Choose any passcode with a probability of being sampled by the agent in the first N episodes of at most $\frac{N}{2^M}$ and consider the MDP for which that passcode is correct. Clearly, in the event that it is not sampled the return for the first N episodes will be 0. Hence with probability at least $1 - \frac{N}{2^M}$ the return for the first N episodes will be zero. Now, simple choose the passcode length $M \geq \frac{\log(N/\delta)}{\log(2)}$ to get the desired result. □

A.2 Further Environment Details

The environments I investigate are all Markov and use flat binary observation vectors to represent the environment state. To allow the model-based agent to learn about the reward or termination function, the environments include occasional random transitions to rewarding or terminal states. The probability of these transitions is chosen to result in much lower return than is achievable with optimal performance in each environment. Except for these rare random transitions to rewarding or terminal states, all environments are deterministic, so simple models can be expected to work well, though I also investigate more sophisticated latent-variable models. Here, I will describe each of the environments in some detail.

The first environment, ProcMaze, is illustrated in Figure 3.5(left). ProcMaze is an episodic environment. ProcMaze consists of procedurally generated grid world mazes, where an agent has to navigate from a start state to a goal state. The maze itself, along with the start state and goal state are randomized at the start of each episode. A reward of -1 is given for each step until the goal is reached, at which point the episode terminates. Also, the agent is rarely randomly teleported to the goal (probability $0.1/T$ where T is the time required to complete the worst-case problem instance for the grid size under the optimal policy) such that it can obtain knowledge of the reward function even with a poor behaviour policy. Difficulty could be scaled by increasing the grid size. The observations consist of a flat binary vector including: one hot vectors for the goal location and agent location, a vector which is one if and only if a cell contains a wall, and a vector which is one if and only if a cell does not contain a wall. The action space includes attempting to move in each cardinal direction, and no-op. An attempted move will fail if it would lead the agent into a wall or the edge of the grid. In each episode, a new maze is generated using randomized depth-first search, which produces reasonable mazes and guarantees the goal is reachable.

The second environment, ButtonGrid, is illustrated in Figure 3.5(middle). ButtonGrid is a continuing environment, with no termination. ButtonGrid

consists of a grid world with a set of randomly placed buttons. An agent (orange in the figure) can move around on the grid and if it hits a button it will toggle it either on (black in the figure) or off (white in the figure). Reward is given whenever all the buttons are set to on, at which point the button locations are randomized, but the number of buttons is held fixed, and all buttons are set to off. Occasionally all the buttons will spontaneously switch to on (probability $0.1/(\text{grid size})^2$), meaning the agent can receive examples of the reward function even while behaving suboptimally. Importantly, it does not suffice to touch each button once to solve this environment, as they are toggled on and off by repeated contact, an agent must also carefully avoid hitting them again after the first time they are pressed. Difficulty can be scaled by increasing the number of buttons on the grid, as well as the grid size, but I focus on the former. The observations consist of a flat binary vector including: one hot vectors for the agent location, a vector which is one if and only if a cell contains a button which is turned on, and a vector which is one if and only if a cell contains a button which is turned off. The action space includes attempting to move in each cardinal direction, and no-op. An attempted move will fail only if it would lead the agent into the edge of the grid.

The third environment, PanFlute is illustrated in Figure 3.5(right). PanFlute is a continuing environment with no termination. PanFlute is intended as a minimal instantiation of an environment with combinatorial complexity in terms of optimal behaviour, but a simple factored transition structure. The observations consist of a binary value for each square in the figure. An agent has n actions available to it, (a,b,c,d,e) in the figure. Each action will activate the associated cell at the bottom of a specific *pipe*. The pipe associated with the last action (alphabetically) has a length of one cell, every other pipe is one cell longer than its (alphabetical) successor. If a cell is activated at a given time step, it will deactivate and activate the cell above it in the same pipe at the next time step. A reward of 1 is received if the cells at the end of each pipe are simultaneously active, otherwise, the reward is always zero. An active pipe-end will always deactivate at the next step regardless of whether

reward is obtained. Occasionally, the cells at the end of all pipes will activate spontaneously (probability $1/n^2$), thus allowing the agent to observe a rewarding situation without having to create it through its own actions. Otherwise, due to the arrangement of pipe lengths, the only way for the agent to obtain reward is to choose each of the n actions in sequence. The difficulty of the environment can be scaled by changing the number of actions n. The probability of a random sequence of n actions reaching the rewarding state (aside from spontaneous activation) is $1/n^n$. Observations consist of a flat binary vector which includes the active/inactive state of each cell in each pipe.

A.3 Experiments in an Environment Without Structured Transitions

Chapter 3 primarily focuses on highlighting environments in which model-based learning is expected to be beneficial. Nevertheless, it is worthwhile to contrast this with what happens in environments which do not have such favourable characteristics. To that end, I ran an additional experiment on an environment without factored structure, which I will present in more detail here.

The environment for this experiment, which I refer to as OpenGrid, was simply an open grid with a goal in the bottom right corner and a reward of -1 for every step until the goal is reached at which point termination occurs. The agent starts in a random location in each episode. As in my other experiments, this environment includes occasional spontaneous transitions to the goal (probability 0.1/(grid size)). The agent location is simply represented by a one-hot vector (effectively tabular) so there is no real structure to exploit. The learned model must essentially memorize every individual transition to learn the dynamics.

The experimental design was the same as in Section 4. I tuned the action-value-function step-size and softmax-exploration temperature from the same set of values on a grid size of size 12 and then used the best hyperparameters for each agent on the other grid sizes. In this experiment, I focused on the

low-data regime where the simple model tended to have the biggest advantage over ER in the previous experiments.

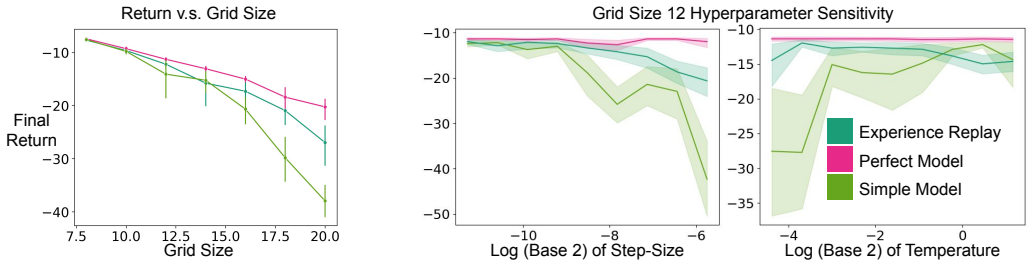

Figure A.2: **Left**: Final performance of greedy policy v.s grid size for OpenGrid in the low data regimes, that is 100 thousand interactions with 10 updates per step. **Right**: Softmax temperature and step-size sensitivity curves for each approach resulting from the grid search on size 12 OpenGrid. In these plots, the other hyperparameter is fixed to its best value from the grid search while varying the temperature or step size.

The results are shown in Figure A.2, with performance v.s. grid size on the left and hyperparameter sensitivity curves from the initial tuning on the right. In contrast to my other experiments, here we see that the simple model becomes worse relative to ER as the environment complexity increases. This is reasonable as the model has no ability to extrapolate beyond the data. The best it can do is memorize what is already in the ER buffer and the limitations of finite model capacity and imperfect optimization prevent it from doing so perfectly. This result helps to contextualize my main results for environments with factored structure by showing how the performance of the model-based approach suffers in a simple environment without such structure.

A.4 Model Details

The latent-variable model consists of the following components:

- Representation Model: $Z_t \sim \hat{p}(Z_t|S_t;\theta)$
- State Reconstructor: $\hat{S}_t \sim \hat{p}(\hat{S}_t|Z_t;\theta)$
- Transition Predictor: $\hat{Z}_t \sim \hat{p}(\hat{Z}_t|Z_{t-1},A_{t-1};\theta)$
- Reward Predictor: $\hat{R}_t \sim \hat{p}(\hat{R}_t|Z_{t-1},A_{t-1};\theta)$
- Termination Predictor: $\hat{\gamma}_t \sim \hat{p}(\hat{\gamma}_t|Z_{t-1},A_{t-1};\theta)$.

All components are implemented as neural networks with θ representing the combined parameter vector. S_t is the state from the environment at time t, A_t is the action, R_t is the reward, γ_t is the continuation probability.[1] Z_t is the latent state constructed by the model from the observation from which transitions, rewards, and terminations are all predicted. The associated versions of each variable with hats are predictions made by the model. The action-value functions associated with the latent-variable models are always trained to predict action values directly from Z_t as opposed to first reconstructing the observation.

For training the model, I use a loss very similar to that employed by Hafner et al. (2021):

$$\mathcal{L}_t(\theta) = -\log(\hat{p}(S_t|Z_t;\theta)) - \log(\hat{p}(R_t|Z_{t-1},A_{t-1};\theta)) - \log(\hat{p}(\gamma_t|Z_{t-1},A_{t-1};\theta)) + KL(\hat{p}(Z_t|S_t;\theta)|\hat{p}(Z_t|Z_{t-1},A_{t-1};\theta)),$$

where $Z_t \sim \hat{p}(Z_t|S_t;\theta)$. I also employ KL-balancing, as described by Hafner et al. (2021), with $\alpha = 0.8$. I experiment with both Gaussian-latent and categorical-latent variables for Z_t. For the categorical case, I use the straight-through estimator to propagate gradients through the discrete latent variables where necessary. In all cases, I train the model on randomly sampled transitions from a replay buffer. Note that it is not necessary to train on sequences in this case, as the lack of recurrence means that the loss at each time step can be independently evaluated. The state reconstructor $\hat{p}(\hat{S}_t|Z_t;\theta)$ uses a sigmoid activation to output the means of a vector of Bernoulli distributions since all tested environments use binary state representations. The reward predictor $\hat{p}(\hat{R}_t|Z_{t-1},A_{t-1};\theta)$ uses a linear activation and outputs the mean of a univariate Gaussian, in which case the above loss is effectively mean-squared error. The termination predictor $\hat{p}(\hat{\gamma}_t|Z_{t-1},A_{t-1};\theta)$ uses a sigmoid activation to output a single Bernoulli termination probability.

For the simple model, the reward and termination predictors are the same except that they take raw state representations as input instead of latent

[1]Two out of three of the environments in the main experiment are continuing, thus termination will never occur and this prediction could be omitted, however, $\gamma_t = 1$ should be learned easily and thus it should make little difference whether it is included or not.

variables. Likewise the transition predictor $\hat{p}(\hat{S}_t|S_{t-1}, A_{t-1}; \theta)$ works directly with the state representation provided as input to the model, and is trained with a negative log-likelihood loss relative to the true states.

A.5 Illustrative Experiment Details

Here I give some additional detail on the setup of the illustrative experiment in Section 3.3. For the most part, I used the same hyperparameters as my main experiments, as detailed in Appendix A.6. The only exception is that for this simple experiment I did not tune any hyperparameters, but rather fixed the Q-learning step size to $2e-4$ and softmax-exploration temperature to 0.1. The softmax-exploration temperature only applies to the model in this case, since the model-free agent was trained on fixed data and thus never selects actions during learning.

I trained all agents for 1,000,000 training steps, to ensure convergence, which was excessive given the small fixed dataset used for training. To control for the total number of value function updates, and the total number of (real or imagined) transitions used in each update, I made the following choices: for model-free DQN and the 1-step model-based agent the value function is updated on batches of 320 transitions, from the dataset or the learned model; the 10-step model-based agent uses a batch of 32 sequences of length 10 generated by the model to equate the number of transitions per update. The models are always trained on 32 transitions from the dataset in each update.

A.6 Hyperparameters

Shared Hyperparameters	**Value**
Number of Hidden Layers	3
Number of Hidden Units	200
Hidden Activation	ELU
Optimizer	AdamW
Adam β_1	0.9
Adam β_2	0.99
Adam ϵ	1e-5
Adam weight decay	1e-6
Q-learning Step-Size	**Tuned** (See Appendix A.7)
Discount Factor	0.9
Batch Size	32 for model-based, 320 for model-free
Exploration Strategy	Softmax
Softmax Temperature	**Tuned** (See Appendix A.7)
Target Network Update Frequency	100
Buffer Size	100,000
Training Start Time	1000
Model Hyperparameters	
Model Learning Step-Size	2e-4
Rollout Length	10
Categorical-Latent Hyperparameters	
Number of Features	32
Width of Features	32
KL Balancing	0.8
KL Loss Scale	1.0
Gaussian Latent Hyperparameters	
Number of Features	32
KL Balancing	0.8
KL Loss Scale	1.0
Minimum std	0.1
std activation	$2 \cdot \sigma(x/2)$

Table A.1: Table of hyperparameters used in experiments in Section 3.5.

A.7 Hyperparameter Tuning and Sensitivity Experiments

Here, I present hyperparameter sensitivity plots for step-size and softmax exploration temperature which resulted from the initial grid search to select hyperparameters for the main experiments in Section 3.5. The initial grid search tried the set $\{0.0125, 0.025, 0.05, 0.1, 0.2, 0.4, 0.8, 1.6, 3.2\}$ for the softmax temperature and the set $\{1.25e-05, 2.50e-05, 5.00e-05, 1.00e-04, 2.00e-$

$04, 4.00e-04, 8.00e-04, 1.60e-03, 3.20e-03\}$ for the step-size. This range was extended in some cases where there was a significant positive trend at the boundary of the range for either parameter, this only affected results for ER and the Gaussian-latent model and the impact was negligible compared to using the best parameters in the initial range. In each case, the mean performance of 30 random seeds was used to evaluate each hyperparameter setting in terms of final performances of the greedy policy. The hyperparameters with the best final performance were selected in each case for use in the main experiments. Sensitivity curves for step size and temperature are shown in Figure A.3 and Figure A.4 respectively.

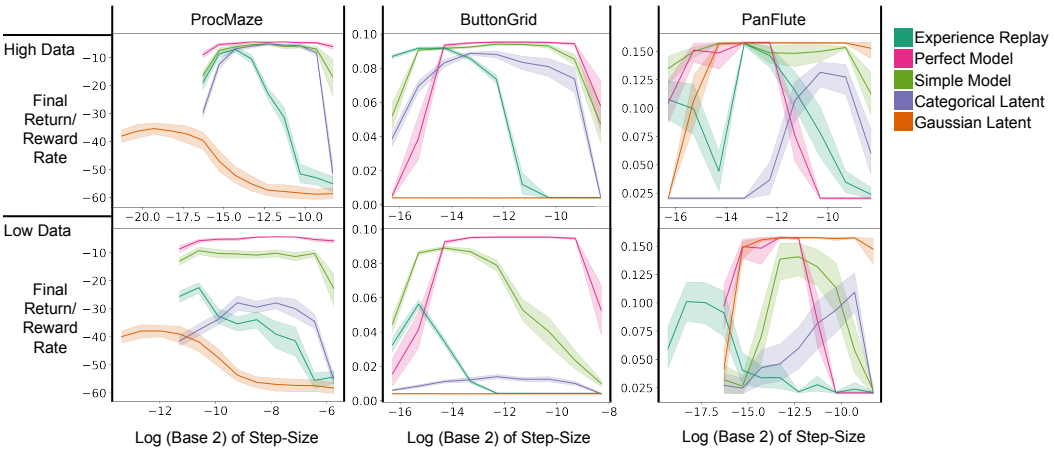

Figure A.3: Step-size sensitivity curves for each approach resulting from the grid-search on an intermediate level of difficulty for each environment (size 4 ProcMaze, 4 button ButtonGrid, 7 pipe PanFlute). In these plots, the softmax temperature is fixed to its best value from the grid-search while varying step size. The search was extended in a few cases when there was a significant positive trend at the boundary of the initial search grid.

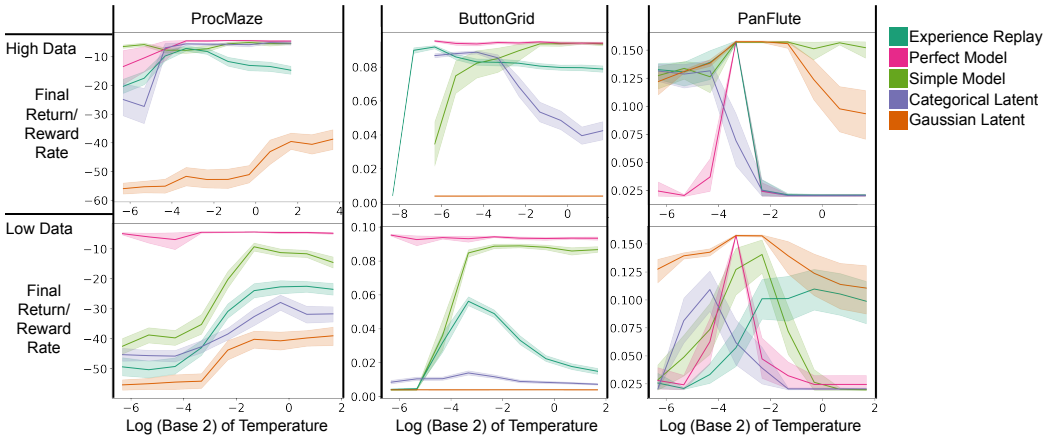

Figure A.4: Softmax temperature sensitivity curves for each approach resulting from the grid-search on an intermediate level of difficulty for each environment (size 4 ProcMaze, 4 button ButtonGrid, 7 pipe PanFlute). In these plots, the step size is fixed to its best value from the grid search while varying softmax temperature. The search was extended in a few cases when there was a significant positive trend at the boundary of the search grid.

A.8 Learning Curves

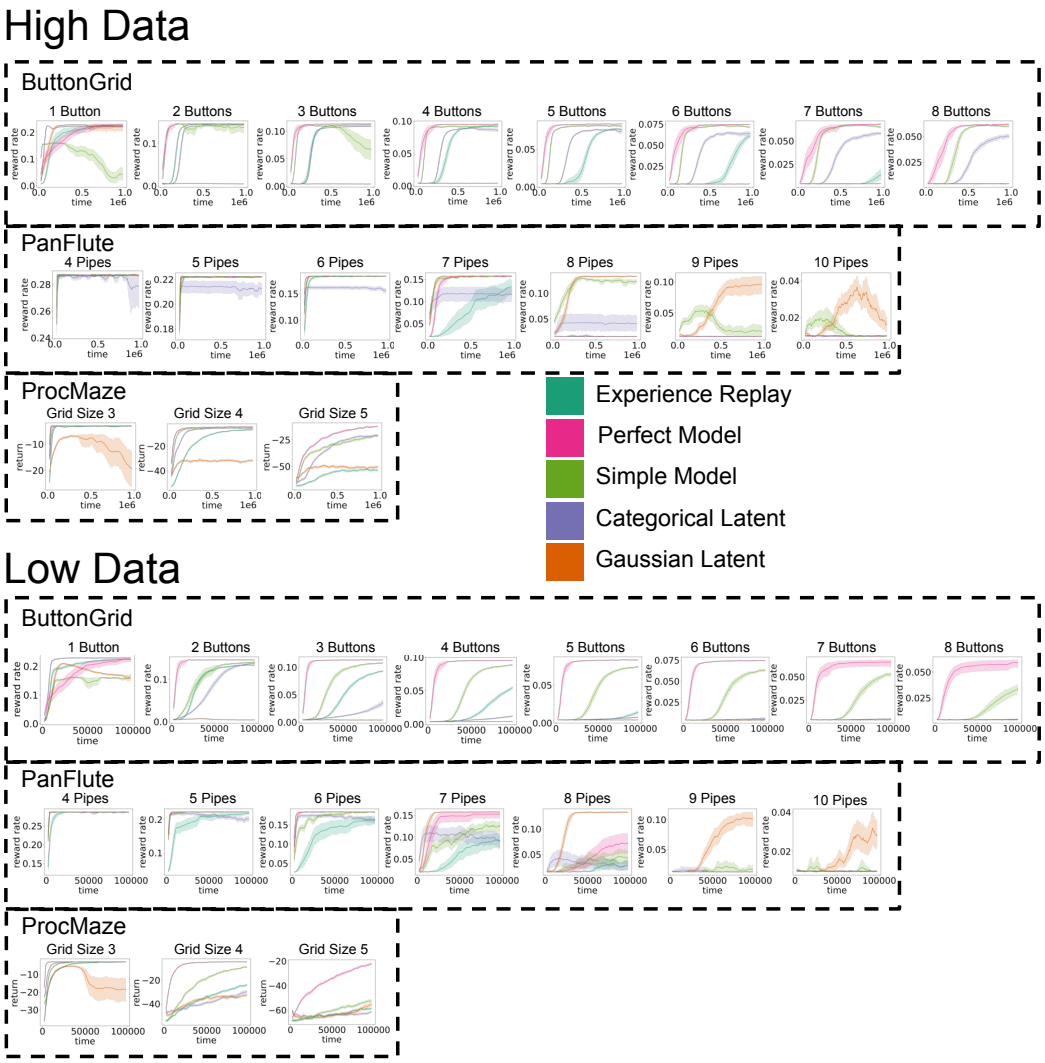

Figure A.5: Full learning curves for experiments in Section 3.5. The performance of the greedy policy is plotted every 5000 updates and smoothed with a moving average over the last 10 values.

A.9 Ablation Experiments

In this section, I present some additional ablation studies to better understand the impact of some of my experimental design decisions made in the main paper. Figure A.6 highlights the impact of removing spontaneous transitions to rewarding states in each of the environments. Figure A.7 compares the

performance of the simple model with 1-step and 10-step model rollouts, where the total number of simulated transitions used in each batch used for DQN updates is controlled.

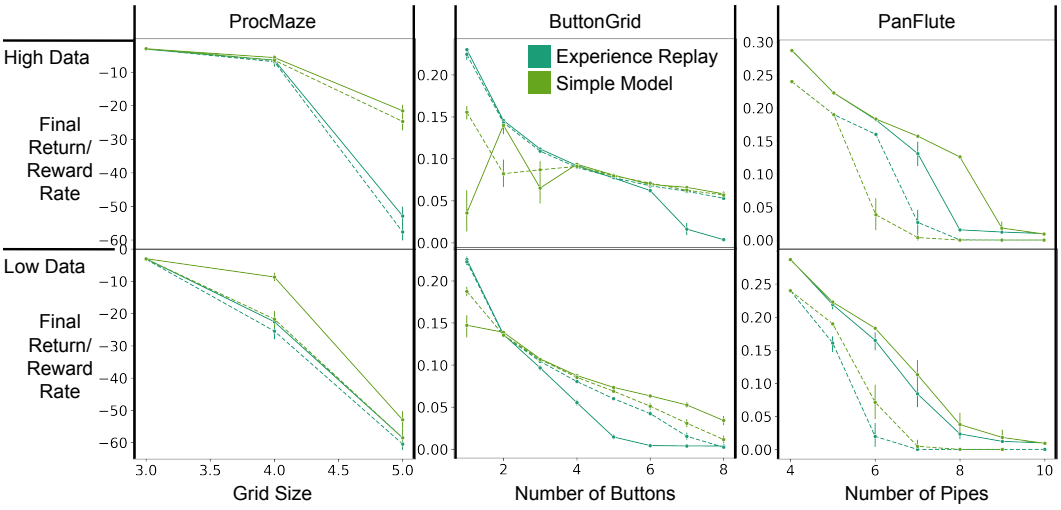

Figure A.6: Comparing the performance of ER and the simple model in variants of each environment class with spontaneous rewards disabled to the original environment. Dotted lines show the curve with spontaneous rewards disabled. The effect of this is highly variable, but is generally detrimental to the model-based approach. Perhaps surprisingly, ER appears to perform better without the random transitions to rewarding states in ButtonGrid, perhaps due to elimination of the resulting noise from the learning signal. The impact was largest in PanFlute, where the absence of spontaneous rewards negatively affected both model-free and model-based approaches, but in the high-data regime led model-free to outperform model-based.

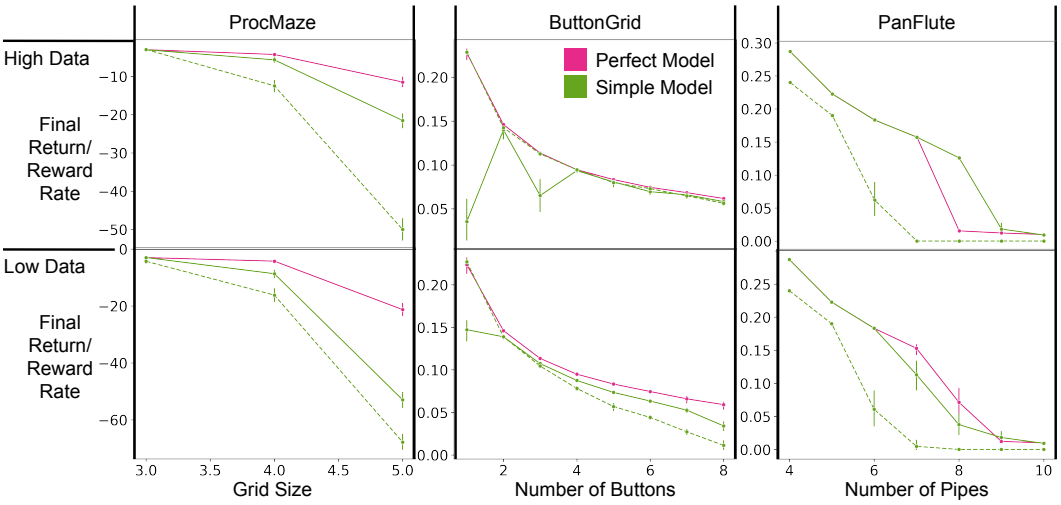

Figure A.7: Comparing simple model performance with 1-step and 10-step rollouts, with perfect model included for reference. Dotted line is 1-step rollouts, solid line is 10-step. In ProcMaze and PanFlute, 10-step rollouts consistently perform much better. However, in ButtonGrid 1-step rollouts perform better for lower button counts in the high data regime, while 10-step rollouts generally perform better in the low data regime.

Appendix B

Appendices for Hindsight Network Credit Assignment

B.1 The Local REINFORCE Estimator is Unbiased

Here, I show that the local REINFORCE estimator $\hat{G}_\Phi^{\mathrm{RE}} = \frac{\partial \log(\pi_\Phi(\Phi|\,\mathrm{pa}(\Phi)))}{\partial \theta_\Phi} R$ is an unbiased estimator of the gradient of the expected reward with respect to θ_Φ.

$$
\begin{aligned}
\mathbb{E}[\hat{G}_\Phi^{\mathrm{RE}}] &= \mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\,\mathrm{pa}(\Phi)))}{\partial \theta_\Phi} R\right] \\
&\overset{(a)}{=} \sum_b \mathbb{P}(\mathrm{pa}(\Phi) = b) \sum_\phi \pi_\Phi(\phi|b) \frac{\partial \log(\pi_\Phi(\phi|b))}{\partial \theta_\Phi} \mathbb{E}\left[R|\mathrm{pa}(\Phi) = b, \Phi = \phi\right] \\
&\overset{(b)}{=} \sum_b \mathbb{P}(\mathrm{pa}(\Phi) = b) \sum_\phi \frac{\partial \pi_\Phi(\phi|b)}{\partial \theta_\Phi} \mathbb{E}\left[R|\mathrm{pa}(\Phi) = b, \Phi = \phi\right] \\
&\overset{(c)}{=} \frac{\partial}{\partial \theta_\Phi} \sum_b \mathbb{P}(\mathrm{pa}(\Phi) = b) \sum_\phi \pi_\Phi(\phi|b) \mathbb{E}\left[R|\mathrm{pa}(\Phi) = b, \Phi = \phi\right] \\
&= \frac{\partial \mathbb{E}[R]}{\partial \theta_\Phi},
\end{aligned}
$$

where (a) expands the expectation over $\mathrm{pa}(\Phi)$ and Φ, (b) rewrites the log gradient, and (c) follows from the fact that the probability of the parents of Φ, $\mathbb{P}(\mathrm{pa}(\Phi) = b))$, does not depend on the parameters θ_Φ controlling Φ itself, nor does the expected reward conditioned on Φ and $\mathrm{pa}(\Phi)$.

B.2 Derivation of Conditional Probability of Output Conditioned on a Markov Blanket

Here, I prove Equation 4.2, that is

$$\mathbb{P}(\Phi = \phi | \operatorname{mb}(\Phi)) = \frac{\pi_\Phi(\phi | \operatorname{pa}(\Phi)) \prod_{C \in \operatorname{ch}(\Phi)} \pi_C(C | \operatorname{pa}(C) \setminus \Phi, \Phi = \phi)}{\sum_{\phi'} \pi_\Phi(\phi' | \operatorname{pa}(\Phi)) \prod_{C \in \operatorname{ch}(\Phi)} \pi_C(C | \operatorname{pa}(C) \setminus \Phi, \Phi = \phi')},$$

which is used in deriving the HNCA gradient estimator. In doing so, I will use Theorem 1 from Section 4 of Pearl (1988), restated here in my notation for convenience:

Theorem B.1 (Theorem 1 (Pearl, 1988))**.** *Let X be a random variable in a Bayesian network. Let $\neg X$ represent the set of all random variables in the network besides X. Then:*

$$\mathbb{P}(X = x | \neg X) = \alpha \, \mathbb{P}(X = x | \operatorname{pa}(X)) \prod_{C \in \operatorname{ch}(X)} \mathbb{P}(C | \operatorname{pa}(C) \setminus X, X = x),$$

where α is a normalizing factor which does not depend on x.

Using this theorem, we can compute $\mathbb{P}(\Phi = \phi | \operatorname{mb}(\Phi))$ as follows:

$$\begin{aligned}
\mathbb{P}(\Phi = \phi | \operatorname{mb}(\Phi)) &\overset{(a)}{=} \mathbb{P}(\Phi = \phi | \neg \Phi) \\
&\overset{(b)}{=} \alpha \, \mathbb{P}(\Phi = \phi | \operatorname{pa}(\Phi)) \prod_{C \in \operatorname{ch}(\Phi)} \mathbb{P}(C | \operatorname{pa}(C) \setminus \Phi, \Phi = \phi) \\
&\overset{(c)}{=} \frac{\mathbb{P}(\Phi = \phi | \operatorname{pa}(\Phi)) \prod_{C \in \operatorname{ch}(\Phi)} \mathbb{P}(C | \operatorname{pa}(C) \setminus \Phi, \Phi = \phi)}{\sum_{\phi'} \mathbb{P}(\Phi = \phi' | \operatorname{pa}(\Phi)) \prod_{C \in \operatorname{ch}(\Phi)} \mathbb{P}(C | \operatorname{pa}(C) \setminus \Phi, \Phi = \phi')} \\
&\overset{(d)}{=} \frac{\pi_\Phi(\phi | \operatorname{pa}(\Phi)) \prod_{C \in \operatorname{ch}(\Phi)} \pi_C(C | \operatorname{pa}(C) \setminus \Phi, \Phi = \phi)}{\sum_{\phi'} \pi_\Phi(\phi' | \operatorname{pa}(\Phi)) \prod_{C \in \operatorname{ch}(\Phi)} \pi_C(C | \operatorname{pa}(C) \setminus \Phi, \Phi = \phi')},
\end{aligned}$$

where (a) follows from the fact that $\operatorname{mb}(\Phi) = \{\operatorname{ch}(\Phi), \operatorname{pa}(\Phi), \operatorname{pa}(\operatorname{ch}(\Phi)) \setminus \Phi\}$ is a *minimal* Markov blanket for Φ and hence Φ is independent of all other variables in the network given $\operatorname{mb}(\Phi)$, (b) follows from Theorem B.1, (c) simply makes the normalizing factor α explicit and (d) uses the fact that $\mathbb{P}(\Phi = \phi | \operatorname{pa}(\Phi)) = \pi_\Phi(\phi | \operatorname{pa}(\Phi))$.

B.3 The HNCA Gradient Estimator has Lower Variance than REINFORCE

Here, I provide the proof of Theorem 4.1.

Theorem 4.1. *Recall that*

$$\hat{G}_{\Phi}^{RE} \doteq \frac{\partial \log(\pi_{\Phi}(\Phi|\operatorname{pa}(\Phi)))}{\partial \theta_{\Phi}} R$$

and

$$\hat{G}_{\Phi}^{HNCA} = \sum_{\phi} \frac{\prod\limits_{C \in \operatorname{ch}(\Phi)} \pi_C(C|\operatorname{pa}(C) \setminus \Phi, \Phi = \phi)}{\sum\limits_{\phi'} \pi_{\Phi}(\phi'|\operatorname{pa}(\Phi)) \prod\limits_{C \in \operatorname{ch}(\Phi)} \pi_C(C|\operatorname{pa}(C) \setminus \Phi, \Phi = \phi')} \frac{\partial \pi_{\Phi}(\phi|\operatorname{pa}(\Phi))}{\partial \theta_{\Phi}} R,$$

then $\mathbb{V}(\hat{G}_{\Phi}^{HNCA}) \leq \mathbb{V}(\hat{G}_{\Phi}^{RE})$*, where* $\mathbb{V}(\vec{X})$ *stand for the elementwise variance of random vector* $\vec{X}$*, and the inequality holds elementwise.*

Proof. The proof follows from applying the law of total variance elementwise. From the derivation in Section 4.2 we know that

$$\hat{G}_{\Phi}^{\mathrm{HNCA}} = \mathbb{E}\left[\hat{G}_{\Phi}^{\mathrm{RE}} \middle| \operatorname{mb}(\Phi), R\right].$$

Now apply the law of total variance to rewrite the variance of the REINFORCE estimator as follows:

$$\begin{aligned}
\mathbb{V}(\hat{G}_{\Phi}^{\mathrm{RE}}) &= \mathbb{E}\left[\mathbb{V}\left(\hat{G}_{\Phi}^{\mathrm{RE}} \middle| \operatorname{mb}(\Phi), R\right)\right] + \mathbb{V}\left(\mathbb{E}\left[\hat{G}_{\Phi}^{\mathrm{RE}} \middle| \operatorname{mb}(\Phi), R\right]\right) \\
&\geq \mathbb{V}\left(\mathbb{E}\left[\hat{G}_{\Phi}^{\mathrm{RE}} \middle| \operatorname{mb}(\Phi), R\right]\right) \\
&= \mathbb{V}(\hat{G}^{\mathrm{HNCA}}(\Phi)).
\end{aligned}$$

□

B.4 HNCA for Softmax Output layer of Contextual Bandit Experiments

For $\Phi = A$, corresponding to the softmax output layer, computing a counterfactual probability $\pi_{\Phi}(\Phi|\operatorname{pa}(\Phi) \setminus B, B = b)$, will require $\mathcal{O}(N_A)$ time (where N_A is the number of possible actions), instead of constant time. This can

be seen by noting that we can easily compute the counterfactual logit corresponding to each action in constant time, but to compute the probability of any given action we must compute counterfactual logits for all actions. Hence, to compute counterfactual probabilities for each parent of the output unit will require $\mathcal{O}(NN_A|\operatorname{pa}(A)|)$, where again N is the number of possible outputs for each parent, assumed the same across parents. Note that this is again $N = 2$ times the complexity of the forward pass if all the parents are Bernoulli units. Again, this can be reduced to $N - 1 = 1$ by reusing the value computed in the forward pass.

Algorithm 3 provides an efficient pseudocode implementation for the softmax output unit used in my contextual bandit experiments. Note that the output unit itself uses the REINFORCE estimator in its update, as it has no children, which precludes the use of HNCA. Nonetheless, the output unit still needs to provide information to its parents, which do use HNCA.

If the entire network consisted of softmax units, each with N output choices, we can see from the above discussion that computing all counterfactual probabilities for each parent would require $\mathcal{O}(N^2 \sum_\Phi |\operatorname{pa}(\Phi)|)$. On the other hand, the forward pass in this case only requires $\mathcal{O}(N \sum_\Phi |\operatorname{pa}(\Phi)|)$. Hence, HNCA would add a factor of N overhead in this case compared to the forward pass. However, it's worth noting that applying the biased straight-through estimator in the softmax case, as is done for example by Hafner et al. (2021), in principle suffers the same N overhead for the backward pass. This is because while the forward pass simply needs to pass a single output for each node, the backward pass operates as if a size N vector of probabilities had been passed, which blows up the input size by a factor of N.

HNCA (Softmax output unit)

1: Receive $\vec{x}$ from parents
2: $\vec{l} = \Theta\vec{x} + \vec{b}$
3: $\vec{p} = \frac{\exp \vec{l}}{\sum_i \exp \vec{l}[i]}$
4: Output $\phi \sim \vec{p}$
5: Receive R from environment
6: **for** all i **do**
7: $\quad L_1[i] = \vec{l} + \Theta[i] \odot (1 - \vec{x})$
8: $\quad L_0[i] = \vec{l} - \Theta[i] \odot \vec{x}$
9: **end for**
10: $\vec{p}_1 = \frac{\exp L_1[\phi]}{\sum_i \exp L_1[i]}$
11: $\vec{p}_0 = \frac{\exp L_0[\phi]}{\sum_i \exp L_0[i]}$
12: Pass $\vec{p}_1, \vec{p}_0, R$ to parents
13: **for** all i **do**
14: $\quad \Theta[i] = \Theta[i] + \alpha\vec{x}(\mathbb{1}(\phi = i) - \vec{p}[i])R$
15: $\quad b[i] = b[i] + \alpha(\mathbb{1}(\phi = i) - \vec{p}[i])R$
16: **end for**

Algorithm 3: Efficient implementation of HNCA message passing for a softmax output unit in a contextual bandit setting. Lines 1-4 implement the forward pass, in this case producing an integer ϕ corresponding to the possible actions. Lines 6-11 compute counterfactual probabilities of the given output class conditional on fixing the value of each parent. Note that $\Theta[i]$ refers to the i_{th} row of the matrix Θ. In this case, computing these counterfactual probabilities requires computation on the order of the number of parents, times the number of possible actions. Line 12 passes the necessary information back to the parents. Lines 13-16 update the parameters according to $\hat{G}^{\text{RE}}_{\Phi}$.

B.5 HNCA to Train a Final Bernoulli Hidden Layer in a Nonlinear Network

Here, I provide a simple demonstration of using HNCA to train a Bernoulli layer as the last hidden layer of a nonlinear network. The task is the same contextual bandit version of MNIST outlined in Section 4.2. The architecture consists of two convolutional layers with 16 channels each, followed by ReLU activation which then feeds into a layer of 200 Bernoulli units, and finally a softmax output. To compute the HNCA estimator in this case I again use Equation 4.3, but now the gradients $\frac{\partial \pi_{\Phi}(\phi|X)}{\partial \theta_{\Phi}}$ are computed by backprop and

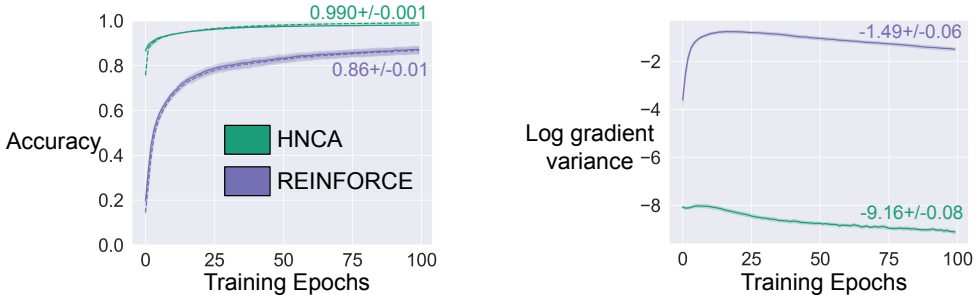

Figure B.1: Training stochastic networks on a contextual bandit version of MNIST with two deterministic convolutional layer forming the input to a single Bernoulli hidden layer. Each line represents the average of 20 random seeds with error bars showing 95% confidence interval. Final values at the end of training (train accuracy for the left plot) are written near each line in matching colour. The top row shows the online training accuracy (or equivalently the average reward) as a dotted line, and the test accuracy as a solid line. The bottom row shows the natural logarithm of the mean gradient variance. Mean gradient variance is computed as the mean of the per-parameter empirical variance over examples in a training batch of 50. HNCA significantly outperforms REINFORCE in this setting.

summed over units when parameters are shared between them. More precisely, define q_0^j, q_1^j and $\bar{q}^j$ as in Algorithm 1 but with an additional index j indicating the specific unit in the Bernoulli layer. The HNCA estimator can then be efficiently implemented in an automatic differentiation framework by defining the following loss:

$$\mathcal{L} = -R \sum_j \text{SG}\left(\frac{q_1^j - q_0^j}{\bar{q}^j}\right) \pi_{\Phi^j}(\phi|X),$$

where in this case $\pi_{\Phi^j}(\phi|X)$ is the policy of unit j, and has a differentiable, nonlinear dependence on the context with arbitrary parameter sharing between units. SG stands for stop gradient, indicating that gradients are not propagated through $\frac{q_1^j - q_0^j}{\bar{q}^j}$. Computing the gradient of this loss function gives us the HNCA gradient estimator for this case, that is $\hat{G}^{HNCA} = \frac{\partial \mathcal{L}}{\partial \theta}$. The softmax output unit still implements Algorithm 3.

I again compare against REINFORCE. As in Section 4.2 I map the output of the Bernoulli units to -1 or 1. The results are shown in Figure B.1 where we see that HNCA again provides a significant benefit over REINFORCE in this setting.

B.6 Derivation of f-HNCA Estimator

In this section, I elaborate on how to derive the f-HNCA gradient estimator $\hat{G}_\Phi^{f\text{-HNCA},i}(\phi)$. Recall that I defined $f_\Phi^i(\phi)$ as the random variable defined by taking the function $f^i(\widetilde{\text{pa}}(f^i);\theta^i)$ and substituting the specific value ϕ instead of the random variable Φ into the arguments while keeping all other $\widetilde{\text{pa}}(f^i)$ equal to the associated random variables. With this definition, we can express $\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\,\text{pa}(\Phi)))}{\partial\theta_\Phi} f^i\right]$ as follows:

$$
\begin{aligned}
&\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\,\text{pa}(\Phi)))}{\partial\theta_\Phi} f^i\right] \\
&\overset{(a)}{=} \mathbb{E}\left[\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\,\text{pa}(\Phi)))}{\partial\theta_\Phi} f^i \middle| \text{mb}(\Phi), \widetilde{\text{pa}}(f^i)\setminus\Phi\right]\right] \\
&\overset{(b)}{=} \mathbb{E}\left[\mathbb{E}\left[\sum_\phi \mathbb{1}(\Phi=\phi)\frac{\partial \log(\pi_\Phi(\phi|\,\text{pa}(\Phi)))}{\partial\theta_\Phi} f_\Phi^i(\phi) \middle| \text{mb}(\Phi), \widetilde{\text{pa}}(f^i)\setminus\Phi\right]\right] \\
&\overset{(c)}{=} \mathbb{E}\left[\sum_\phi \mathbb{E}\left[\mathbb{1}(\Phi=\phi)\middle|\text{mb}(\Phi), \widetilde{\text{pa}}(f^i)\setminus\Phi\right] \frac{\partial \log(\pi_\Phi(\phi|\,\text{pa}(\Phi)))}{\partial\theta_\Phi} f_\Phi^i(\phi)\right] \\
&\overset{(d)}{=} \mathbb{E}\left[\sum_\phi \mathbb{E}\left[\mathbb{1}(\Phi=\phi)|\text{mb}(\Phi)\right] \frac{\partial \log(\pi_\Phi(\phi|\,\text{pa}(\Phi)))}{\partial\theta_\Phi} f_\Phi^i(\phi)\right] \\
&= \mathbb{E}\left[\sum_\phi \mathbb{P}(\Phi=\phi|\,\text{mb}(\Phi)) \frac{\partial \log(\pi_\Phi(\phi|\,\text{pa}(\Phi)))}{\partial\theta_\Phi} f_\Phi^i(\phi)\right], \qquad \text{(B.1)}
\end{aligned}
$$

where (a) applies the law of total expectation, (b) follows because the indicator function is zero except where the summand equals the expression from the previous line, (c) moves deterministic quantities out of the inner expectation, and (d) exploits the fact that Φ is independent of $\widetilde{\text{pa}}(f^i)\setminus\Phi$ given $\text{mb}(\Phi)$. From here, as in Section 4.2, we substitute Equation 4.2 into the expression within the expectation to get the following unbiased estimator for $\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\,\text{pa}(\Phi)))}{\partial\theta_\Phi} f^i\right]$:

$$
\hat{G}_\Phi^{f\text{-HNCA},i}(\phi) \doteq \sum_\phi \rho_\Phi(\phi)\frac{\partial \pi_\Phi(\phi|\,\text{pa}(\Phi))}{\partial\theta_\Phi} f_\Phi^i(\phi),
$$

where $\rho_\Phi(\phi) = \frac{\prod_{C\in\text{ch}(\Phi)} \pi_C(C|\,\text{pa}(C)\setminus\Phi,\Phi=\phi)}{\sum_{\phi'}\pi_\Phi(\phi'|\,\text{pa}(\Phi))\prod_{C\in\text{ch}(\Phi)}\pi_C(C|\,\text{pa}(C)\setminus\Phi,\Phi=\phi')}$. In the case where $\text{ch}(\Phi)\cap\widetilde{\text{an}}(f^i)=\emptyset$ it's not necessary to propagate credit from the children, $\text{ch}(\Phi)$,

as they cannot influence the reward. In this case, I instead use a simpler estimator derived as follows:

$$
\begin{aligned}
&\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\operatorname{pa}(\Phi)))}{\partial\theta_\Phi} f^i\right] \\
&\quad\overset{(a)}{=} \mathbb{E}\left[\mathbb{E}\left[\frac{\partial \log(\pi_\Phi(\Phi|\operatorname{pa}(\Phi)))}{\partial\theta_\Phi} f^i \middle| \operatorname{pa}(\Phi), \widetilde{\operatorname{pa}}(f^i)\setminus\Phi\right]\right] \\
&\quad\overset{(b)}{=} \mathbb{E}\left[\mathbb{E}\left[\sum_\phi \mathbb{1}(\Phi=\phi)\frac{\partial \log(\pi_\Phi(\phi|\operatorname{pa}(\Phi)))}{\partial\theta_\Phi} f^i_\Phi(\phi) \middle| \operatorname{pa}(\Phi), \widetilde{\operatorname{pa}}(f^i)\setminus\Phi\right]\right] \\
&\quad\overset{(c)}{=} \mathbb{E}\left[\sum_\phi \mathbb{E}\left[\mathbb{1}(\Phi=\phi)|\operatorname{pa}(\Phi)\right]\frac{\partial \log(\pi_\Phi(\phi|\operatorname{pa}(\Phi)))}{\partial\theta_\Phi} f^i_\Phi(\phi)\right] \\
&\quad= \mathbb{E}\left[\sum_\phi \pi_\Phi(\phi|\operatorname{pa}(\Phi))\frac{\partial \log(\pi_\Phi(\phi|\operatorname{pa}(\Phi)))}{\partial\theta_\Phi} f^i_\Phi(\phi)\right] \\
&\quad= \mathbb{E}\left[\sum_\phi \frac{\partial \pi_\Phi(\phi|\operatorname{pa}(\Phi))}{\partial\theta_\Phi} f^i_\Phi(\phi)\right],
\end{aligned}
\tag{B.2}
$$

where (a) applies the law of total expectation, (b) follows because the indicator function is zero except where the summand equals the expression from the previous line, (c) exploits the fact that Φ is independent of $\widetilde{\operatorname{pa}}(f^i)\setminus\Phi$ given $\operatorname{pa}(\Phi)$ due to the assumption $\operatorname{ch}(\Phi)\cap\widetilde{\operatorname{an}}(f^i)=\emptyset$. The final expression within the expectation gives us the unbiased estimator

$$
\hat{G}^{f\text{-HNCA},i}_\Phi(\phi) \doteq \sum_\phi \frac{\partial \pi_\Phi(\phi|\operatorname{pa}(\Phi))}{\partial\theta_\Phi} f^i_\Phi(\phi).
$$

In my experiments, I only distinguish the cases where $\operatorname{ch}(\Phi)\cap\widetilde{\operatorname{an}}(f^i)=\emptyset$ and $\operatorname{ch}(\Phi)\cap\widetilde{\operatorname{an}}(f^i)\neq\emptyset$. However, as alluded to in Section 4.3, if only a subset of $\operatorname{ch}(\Phi)$ lies in $\widetilde{\operatorname{an}}(f^i)$ we can replace $\operatorname{ch}(\Phi)$ in $\rho_\Phi(\phi)$ with $\operatorname{ch}^i(\Phi) = (\operatorname{ch}(\Phi)\cap\widetilde{\operatorname{an}}(f^i))$. To see that this is the case, it suffices to note that if a particular child C has no downstream connections to a particular function component f^i, then $\frac{\partial\,\mathbb{E}[f^i]}{\partial\theta_\Phi}$ must be the same in a new network with the connection from Φ to C severed as in the original network.

B.7 Efficient Implementation of f-HNCA

f-HNCA algorithm for Linear Function Components
1: Receive $\vec{x}$ from parents
2: $l = \vec{\theta} \cdot \vec{x} + b$
3: $f = \eta(l)$
4: $\vec{l_1} = l + \vec{\theta} \odot (1 - \vec{x})$
5: $\vec{l_0} = l - \vec{\theta} \odot \vec{x}$
6: $\vec{f_1} = \eta(\vec{l_1})$
7: $\vec{f_0} = \eta(\vec{l_0})$
8: Pass $\vec{f_1}, \vec{f_0}, f$ to parents

Algorithm 4: Efficient implementation of f-HNCA for a function component which consists of a linear function of its inputs followed by an arbitrary activation η. Inputs are assumed to be Bernoulli. The forward pass in lines 1-3 takes input from the parents and uses it to compute the function component R. Lines 4-7 use the logit l to efficiently compute a vector of counterfactual function components $\vec{f_1}$ and $\vec{f_0}$ where each element corresponds to a counterfactual function component obtained if all else was the same but a given parent's value was fixed to 1 or 0. Here $\odot$ represents the elementwise product. Line 8 passes the necessary information to the unit's children.

In addition to the efficiency of computing counterfactual probabilities, for f-HNCA, we have to consider the efficiency of computing counterfactual function components $f^i_\Phi(\phi)$. For function components with no direct connection to a unit Φ, this is trivial as $f^i_\Phi(\phi) = f^i$. If f^i is directly connected, then implementing f-HNCA with efficiency similar to HNCA will require that we are able to compute $f^i_\Phi(\phi)$ from f^i in constant time. This is the case, for example, if f^i is a linear function followed by some activation. For example functions of the form $f^i = \log(\sigma(\vec{\theta} \cdot \vec{x} + b))$ which appear in the ELBO function used in my VAE experiments. Algorithm 4 presents pseudocode for efficiently computing counterfactual values for such function components, and passing them to connected units.

If only a subset of $\mathrm{ch}(\Phi)$ lies in $\widetilde{\mathrm{an}}(f^i)$ we could use $\mathrm{ch}^i(\Phi) = (\mathrm{ch}(\Phi) \cap \widetilde{\mathrm{an}}(f^i))$, or any superset, in place of $\mathrm{ch}(\Phi)$ in $\rho_\Phi(\phi)$. In this case, we would also have to consider the complexity of computing the HNCA estimator for

each such $\mathrm{ch}^i(\Phi)$. In the worst case $\mathrm{ch}^i(\Phi)$ may be different for each i, meaning that $\rho_\Phi(\phi)$ may have to be separately computed for each i, requiring a product of up to $|\mathrm{ch}(\Phi)|$ numbers for each function component f^i. I leave open the question of how efficiently this can be done in general. For now, I focus on the case where either $\mathrm{ch}^i(\Phi) = \emptyset$ or $\mathrm{ch}^i(\Phi) = \mathrm{ch}(\phi)$. Focusing on this case allows us to rewrite the f-HNCA gradient estimator as follows:

$$\hat{G}_\Phi^{f\text{-HNCA}} = \sum_\phi \frac{\partial \pi_\Phi(\phi|\,\mathrm{pa}(\Phi))}{\partial \theta_\Phi} \Bigg(\rho_\Phi(\phi) \Bigg(\sum_{i:\mathrm{ch}^i(\Phi)\neq\emptyset, \Phi\in\widetilde{\mathrm{pa}}(f^i)} f^i_\Phi(\phi) + \sum_{i:\mathrm{ch}^i(\Phi)\neq\emptyset, \Phi\notin\widetilde{\mathrm{pa}}(f^i)} f^i \Bigg) + \sum_{i:\mathrm{ch}^i(\Phi)=\emptyset, \Phi\in\widetilde{\mathrm{pa}}(f^i)} f^i_\Phi(\phi) \Bigg) + \sum_i \frac{\partial f^i}{\partial \theta_\Phi},$$

where $\rho_\Phi(\phi) = \frac{\prod_{C\in\mathrm{ch}(\Phi)} \pi_C(C|\,\mathrm{pa}(C)\setminus\Phi, \Phi=\phi)}{\sum_{\phi'} \pi_\Phi(\phi'|\,\mathrm{pa}(\Phi)) \prod_{C\in\mathrm{ch}(\Phi)} \pi_C(C|\,\mathrm{pa}(C)\setminus\Phi, \Phi=\phi')}$. Notice that we do not need to compute a different value of $\rho_\Phi(\phi)$ for each f^i, as we treat the dependence on children as either all or none. The three sums over function components from first to last handle: function components with both mediated and direct connection to Φ, function components with only mediated connections to Φ, and function components with only direct connections to Φ.

Furthermore, if during the backward pass there are function components which we know have no direct connection to units further upstream, we can accumulate these in a sum and credit upstream units with the sum rather than separately computing the sum in each unit. This is analogous to accumulating the future return in RL.

Algorithm 5 presents pseudocode for an efficient implementation of f-HNCA for a Bernoulli unit within a feedforward architecture where each function component is credited as being either downstream of every unit in the following layer or none.

B.8 The f-HNCA Gradient Estimator has Lower Variance than REINFORCE

Here, I verify that the components of the f-HNCA estimator with $\hat{G}_{\Phi}^{f\text{-HNCA},i}(\phi)$ have lower variance than the associated components of the analogous REINFORCE estimator. This is formalized in the following theorem:

Theorem B.2. *Let*

$$\hat{G}_{\Phi}^{f\text{-}HNCA,i}(\phi) \doteq \begin{cases} \sum_{\phi} \rho_{\Phi}(\phi) \frac{\partial \pi_{\Phi}(\phi|\operatorname{pa}(\Phi))}{\partial \theta_{\Phi}} f_{\Phi}^{i}(\phi) & \text{if } \operatorname{ch}(\Phi) \cap \widetilde{\operatorname{an}}(f^i) \neq \emptyset \\ \sum_{\phi} \frac{\partial \pi_{\Phi}(\phi|\operatorname{pa}(\Phi))}{\partial \theta_{\Phi}} f_{\Phi}^{i}(\phi) & \text{if } \operatorname{ch}(\Phi) \cap \widetilde{\operatorname{an}}(f^i) = \emptyset \end{cases}$$

where $\rho_{\Phi}(\phi) = \frac{\prod_{C \in \operatorname{ch}(\Phi)} \pi_C(C|\operatorname{pa}(C)\setminus\Phi, \Phi=\phi)}{\sum_{\phi'} \pi_{\Phi}(\phi'|\operatorname{pa}(\Phi)) \prod_{C \in \operatorname{ch}(\Phi)} \pi_C(C|\operatorname{pa}(C)\setminus\Phi, \Phi=\phi')}$. *Let*

$$\hat{G}_{\Phi}^{RE,i} = \frac{\partial \log(\pi_{\Phi}(\Phi|\operatorname{pa}(\Phi)))}{\partial \theta_{\Phi}} f^i,$$

that is, the obvious generalization of REINFORCE to a specific function component. Then

$$\mathbb{V}(\hat{G}_{\Phi}^{f\text{-}HNCA,i}) \leq \mathbb{V}(\hat{G}_{\Phi}^{RE,i}).$$

Proof. We will separately consider the case where $\operatorname{ch}(\Phi) \cap \widetilde{\operatorname{an}}(f^i) \neq \emptyset$ and $\operatorname{ch}(\Phi) \cap \widetilde{\operatorname{an}}(f^i) = \emptyset$. First, when $\operatorname{ch}(\Phi) \cap \widetilde{\operatorname{an}}(f^i) \neq \emptyset$ we know from Equation B.1 that can write $\hat{G}_{\Phi}^{f\text{-HNCA},i}$ as follows:

$$\begin{aligned} \hat{G}_{\Phi}^{f\text{-HNCA},i} &= \mathbb{E}\left[\frac{\partial \log(\pi_{\Phi}(\Phi|\operatorname{pa}(\Phi)))}{\partial \theta_{\Phi}} f^i \middle| \operatorname{mb}(\Phi), f_{\Phi}^{i}(\phi) \right] \\ &= \mathbb{E}\left[\hat{G}_{\Phi}^{\mathrm{RE},i} \middle| \operatorname{mb}(\Phi), \widetilde{\operatorname{pa}}(f^i) \setminus \Phi \right]. \end{aligned}$$

Now apply the law of total variance to rewrite the variance of the REINFORCE estimator as follows:

$$\begin{aligned} \mathbb{V}(\hat{G}_{\Phi}^{\mathrm{RE},i}) &= \mathbb{E}\left[\mathbb{V}\left(\hat{G}_{\Phi}^{\mathrm{RE},i} \middle| \operatorname{mb}(\Phi), \widetilde{\operatorname{pa}}(f^i) \setminus \Phi \right) \right] + \mathbb{V}\left(\mathbb{E}\left[\hat{G}_{\Phi}^{\mathrm{RE},i} \middle| \operatorname{mb}(\Phi), \widetilde{\operatorname{pa}}(f^i) \setminus \Phi \right] \right) \\ &\geq \mathbb{V}\left(\mathbb{E}\left[\hat{G}_{\Phi}^{\mathrm{RE},i} \middle| \operatorname{mb}(\Phi), \widetilde{\operatorname{pa}}(f^i) \setminus \Phi \right] \right) \\ &= \mathbb{V}(\hat{G}^{f\text{-HNCA}}(\Phi)). \end{aligned}$$

For the case where $\operatorname{ch}^{i}(\Phi) = \emptyset$, we know from Equation B.2 that

$$\begin{aligned} \hat{G}_{\Phi}^{f\text{-HNCA},i} &= \mathbb{E}\left[\frac{\partial \log(\pi_{\Phi}(\Phi|\operatorname{pa}(\Phi)))}{\partial \theta_{\Phi}} f^i \middle| \operatorname{pa}(\Phi), \widetilde{\operatorname{pa}}(f^i) \setminus \Phi \right] \\ &= \mathbb{E}\left[\hat{G}_{\Phi}^{\mathrm{RE},i} \middle| \operatorname{pa}(\Phi), \widetilde{\operatorname{pa}}(f^i) \setminus \Phi \right]. \end{aligned}$$

Now, again, apply the law of total variance to rewrite the variance in the REINFORCE estimator:

$$\begin{aligned}\mathbb{V}(\hat{G}_{\Phi}^{\mathrm{RE},i}) &= \mathbb{E}\left[\mathbb{V}\left(\hat{G}_{\Phi}^{\mathrm{RE},i}\middle|\mathrm{pa}(\Phi), \widetilde{\mathrm{pa}}(f^i) \setminus \Phi\right)\right] + \mathbb{V}\left(\mathbb{E}\left[\hat{G}_{\Phi}^{\mathrm{RE},i}\middle|\mathrm{pa}(\Phi), \widetilde{\mathrm{pa}}(f^i) \setminus \Phi\right]\right) \\ &\geq \mathbb{V}\left(\mathbb{E}\left[\hat{G}_{\Phi}^{\mathrm{RE},i}\middle|\mathrm{pa}(\Phi), \widetilde{\mathrm{pa}}(f^i) \setminus \Phi\right]\right) \\ &= \mathbb{V}(\hat{G}^{f\text{-HNCA,i}}(\Phi)).\end{aligned}$$

□

B.9 Further Details of Discrete VAE Experiments

Here, I provide some additional detail on the methods used in my discrete VAE experiments.

I compare f-HNCA with REINFORCE and two stronger, unbiased, baselines for optimizing an ELBO of a VAE trained to generate MNIST digits. The other baselines are DisARM (Dong, Mnih, et al., 2020), and REINFORCE leave one out (REINFORCE LOO; Kool et al. (2019)).

REINFORCE LOO, based on the version used by Dong, Mnih, et al. (2020), samples two partial forward passes starting at each layer to compute its baseline. In other words, we first run a single forward pass to generate one sample form each $\vec{\Phi}_i = \vec{\phi}_i(1)$. All the function components that lie downstream of $\vec{\Phi}_i$ are summed up to produce one sample of the forward function components $\tilde{f}_i(1)$. This serves as the first of 2 samples used to construct the REINFORCE LOO gradient estimator in each layer. Then, in each layer, i we also draw a second sample $\vec{\Phi}_i = \vec{\phi}_i(2)$ conditioned on $\vec{\phi}_{i-1}(1)$ (or $\vec{X}$ for $i = 1$) all $\vec{\Phi}_j$ for $j > i$ are then resampled sequentially and the new sampled values used as input to the forward function components. This produces, for each layer, another sample of the forward function components which we'll call $\tilde{f}_i(2)$. This results in the following gradient estimator:

$$\begin{aligned}\hat{G}^{\mathrm{RLOO}}(\Phi) = \frac{1}{2}\Bigg(&\frac{\partial \log(\pi_\Phi(\phi(1)|\,\mathrm{pa}(\Phi)))}{\partial \theta_\Phi}(\tilde{f}(1) - \tilde{f}(2)) + \\ &\frac{\partial \log(\pi_\Phi(\phi(2)|\,\mathrm{pa}(\Phi)))}{\partial \theta_\Phi}(\tilde{f}(2) - \tilde{f}(1))\Bigg),\end{aligned} \tag{B.3}$$

where I have suppressed the specific layer and written the estimator for a specific unit Φ in the vector $\vec{\Phi}_i$. Note that the computational cost of this procedure is quadratic in the number of layers, as we need to resample a partial forward pass to generate $\tilde{f}_i(2)$ for each layer i. DisARM has a similar computational requirement, requiring forward resampling to generate an antithetic sample in each layer.

I also experimented with another version of REINFORCE LOO that avoided this quadratic scaling of computational cost with number of layers. This second version of REINFORCE LOO used 2 independent forward passes for each input to construct a baseline, I call this REINFORCE LOO IS, for independent sample. Since REINFORCE LOO IS doesn't require sampling partial forward passes for each layer, it avoids a quadratic scaling of compute time with number of network layers which occurs for both DisARM and REINFORCE LOO. More precisely, rather than resampling in each layer, REINFORCE LOO IS simply generates 2 full forward passes, using the downstream function components of the first sample in each layer i to define $\tilde{f}_i(1)$ and $\pi_\Phi(\phi(1)|\,\mathrm{pa}(\Phi)))$ and the downstream components of the second to define $\tilde{f}_i(2)$ and $\pi_\Phi(\phi(2)|\,\mathrm{pa}(\Phi)))$. The form of the resulting estimator is otherwise the same as Equation B.3. The drawback is that the baselines used for REINFORCE LOO IS will be less correlated, since unlike REINFORCE LOO its baseline uses a different sample for nodes upstream of the layer for which the baseline is being computed. Empirically, I found this version to perform just slightly worse than the first version, hence I chose to omit the results to avoid clutter.

In f-HNCA with baseline, for each layer, the baseline consists of a scalar moving average of the sum of those components of f with mediated connections (those highlighted in pink and orange in Figure 4.2). This baseline is subtracted from the leftmost sum over i in Equation 4.6 to produce a centred learning signal. I use a discount rate 0.99 for the moving average. For REINFORCE with baseline I use a similar moving average baseline, but in this case constructed as the sum of all downstream function components.

As in my contextual bandit experiments, I use dynamic binarization. Following Dong, Mnih, et al. (2020), the decoder and encoder each consist of

a fully connected, stochastic feedforward neural network with 1, 2 or 3 layers, each hidden layer has 200 Bernoulli units. As in Section 4.2, I train using ADAM optimizer with a step size 10^{-4} and batch size of 50. Training proceeds for 840 epochs, approximately equivalent to the 10^6 updates used by Dong, Mnih, et al. (2020). For consistency with prior work, I use Bernoulli units with a zero-one output. Unlike Dong, Mnih, et al. (2020) I use ADAM to train the parameters of the prior as well, rather than using SGD.

For all methods, each unit is trained based only on downstream function components as opposed to using the full function f. Also, for all methods, direct gradients (i.e. the right expectation in Equation 4.4) are trained with only a single sample per training example. In practice, it may be natural to use multiple samples in methods like REINFORCE LOO given that multiple samples are drawn to construct the estimator of the left expectation anyway. This choice was made to reduce confounding, given I am mainly interested in how well different methods estimate the left expectation.

B.10 Multisample Test-set Bounds

In this section, I report 100 sample ELBOs on the MNIST test set for networks trained with each of the algorithms evaluated in the experiments of Section 4.3. Multi-sample bounds, as introduced by Burda et al. (2015) provide a tighter bound on the data likelihood under the generative model. Note that these results simply compute a multi-sample bound using the final trained encoder and decoder and, unlike Burda et al. (2015), still use the single-sample ELBO as a training objective. These results are presented in Table B.1. These results show the same trend as the training ELBOs in Figure 4.3.

	1 Layer	2 Layer	3 Layer
HNCA	**-107.5±0.1**	-103.7±0.1	-102.1±0.2
HNCA with Baseline	NA	**-97.3±0.1**	**-94.6±0.2**
DisARM	-108.2±0.2	-99.27±0.06	-96.7±0.1
REINFORCE LOO	-108.3±0.1	-99.5±0.1	-96.9±0.1
REINFORCE	-120.1±0.2	-115.1±0.1	-114.7±0.1
REINFORCE with Baseline	-110.6±0.1	-102.8±0.2	-100.2±0.1

Table B.1: 100 sample test-set likelihood bounds for networks trained with each of the algorithms evaluated in Section 4.3. Each cell provides the mean and 95% confidence interval from 5 random seeds. The best result for each Layer count is written in bold.

B.11 HNCA Ablation Results

In this section, I assess the impact of avoiding propagating credit through children in f-HNCA when a particular function component has only direct connections (those highlighted in green in Figure 4.2). In particular, instead of using

$$\hat{G}_{\Phi}^{f\text{-HNCA},i}(\phi) \doteq \begin{cases} \sum_{\phi} \rho_{\Phi}(\phi) \frac{\partial \pi_{\Phi}(\Phi|\,\mathrm{pa}(\Phi))}{\partial \theta_{\Phi}} f_{\Phi}^{i}(\phi) & \text{if } \mathrm{ch}(\Phi) \cap \widetilde{\mathrm{an}}(f^i) \neq \emptyset \\ \sum_{\phi} \frac{\partial \pi_{\Phi}(\Phi|\,\mathrm{pa}(\Phi))}{\partial \theta_{\Phi}} f_{\Phi}^{i}(\phi) & \text{if } \mathrm{ch}(\Phi) \cap \widetilde{\mathrm{an}}(f^i) = \emptyset \end{cases}, \tag{B.4}$$

I simply use

$$\hat{G}_{\Phi}^{f\text{-HNCA},i}(\phi) \doteq \begin{cases} \sum_{\phi} \rho_{\Phi}(\phi) \frac{\partial \pi_{\Phi}(\Phi|\,\mathrm{pa}(\Phi))}{\partial \theta_{\Phi}} f_{\Phi}^{i}(\phi) & \text{if } \mathrm{ch}(\Phi) \neq \emptyset \\ \sum_{\phi} \frac{\partial \pi_{\Phi}(\Phi|\,\mathrm{pa}(\Phi))}{\partial \theta_{\Phi}} f_{\Phi}^{i}(\phi) & \text{if } \mathrm{ch}(\Phi) = \emptyset \end{cases}, \tag{B.5}$$

multiplying by $\rho_{\Phi}(\phi)$ as long as the unit has children, even if no children have downstream connections to the function component, that is even if $\mathrm{ch}(\Phi) \cap \widetilde{\mathrm{an}}(f^i) = \emptyset$. In this case, I also include these function components in the subtracted baseline. I additionally investigate the impact of including redundant upstream function components in the HNCA gradient estimator. The results for the hierarchical VAE task are shown in Figure B.2. Propagating credit through all children resulted in significantly worse performance for f-HNCA . The additional impact of including upstream function components is minimal. Presumably, the subtracted baseline is able to mitigate the majority of increased variance resulting from including these function components.

f-HNCA algorithm for Bernoulli unit

1: Receive $\vec{x}$ from parents
2: $l = \vec{\theta} \cdot \vec{x} + b$
3: $p = \sigma(l)$
4: $\phi \sim Bernoulli(p)$
5: Pass ϕ to children
6: Receive $\vec{q}_1, \vec{q}_0$ from child units
7: Receive $\vec{f}_0^d, \vec{f}_1^d$ from child function components with only direct connections
8: Receive $\vec{f}_0^c, \vec{f}_1^c$ from child function components which are also connected through children
9: Receive G, sum of downstream non-child function components
10: $f_0^c = \sum_i \vec{f}_0^c[I]$; $f_1^c = \sum_i \vec{f}_1^c[i]$; $f_0^d = \sum_i \vec{f}_0^d[i]$; $f_1^d = \sum_i \vec{f}_1^d[i]$
11: $q_1 = \prod_i \vec{q}_1[i]$; $q_0 = \prod_i \vec{q}_0[i]$
12: $\bar{q} = pq_1 + (1-p)q_o$
13: $\vec{l}_1 = l + \vec{\theta} \odot (1 - \vec{x})$; $\vec{l}_0 = l - \vec{\theta} \odot \vec{x}$
14: $\vec{p}_1 = (1-\phi)(1-\sigma(\vec{l}_1)) + \phi\sigma(\vec{l}_1)$; $\vec{p}_0 = (1-\phi)(1-\sigma(\vec{l}_0)) + \phi\sigma(\vec{l}_0)$
15: Pass $\vec{p}_1, \vec{p}_0$ to parents
16: $\vec{\theta} = \vec{\theta} + \alpha\sigma'(l)\vec{x}\left(\frac{q_1 f_1^c - q_0 f_0^c}{\bar{q}} + \frac{q_1 - q_0}{\bar{q}}G + f_1^d - f_0^d\right)$
17: $b = b + \alpha\sigma'(l)\left(\frac{q_1 f_1^c - q_0 f_0^c}{\bar{q}} + \frac{q_1 - q_0}{\bar{q}}G + f_1^d - f_0^d\right)$

Algorithm 5: Efficient implementation of f-HNCA for a Bernoulli unit, where function components are credited through all children, or none. I omit any direct dependence of function components on network parameters for conciseness. Lines 1-5 implement the forward pass, which takes input from the parents, computes the fire probability p and samples $\phi \in \{0, 1\}$. In the backward pass, the unit receives two vectors $\vec{q}_1$ and $\vec{q}_0$, each with one element for each child unit of the current unit, as in Algorithm 1. The unit also receives vectors $\vec{f}_1^d$ and $\vec{f}_0^d$ containing counterfactual function components from function components with only direct connections. Likewise, $\vec{f}_1^c$ and $\vec{f}_0^c$ contain counterfactual function components from child function components with direct connections as well as additional connections mediated through children. Finally, G contains the cumulative sum of all function components which are downstream of the current unit but not directly connected. Line 10 sums up the counterfactual function components. Line 11 takes the product of all child unit probabilities to compute $\prod_{C \in \mathrm{ch}(\Phi)} \pi_C(C|\,\mathrm{pa}(C) \setminus \Phi, \Phi = 0/1)$. Line 12 computes the associated normalizing factor. Lines 13 and 14 use the already computed logit l to efficiently compute a vector of probabilities $\vec{p}_1$ and $\vec{p}_0$ where each element corresponds to a counterfactual probability of ϕ if all else was the same but a given parent's value was fixed to 1 or 0. Here $\odot$ represents the elementwise product. Line 15 passes the necessary information to the unit's children. Lines 16 and 17 finally update the parameter using $\hat{G}_\Phi^{f\text{-HNCA}}$ with learning-rate hyperparameter α.

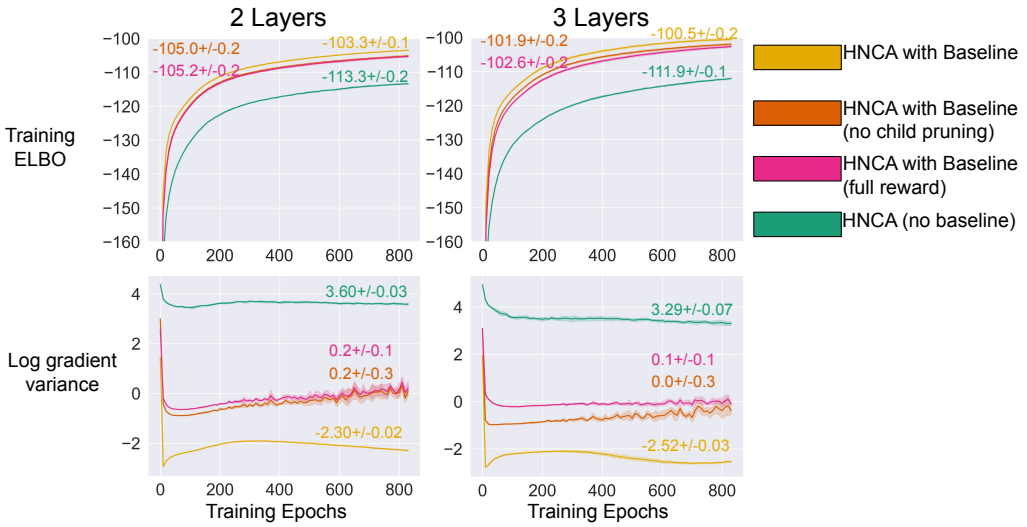

Figure B.2: Training stochastic VAEs to generate MNIST digits with f-HNCA with different aspects ablated. I omit the single-layer VAE as the ablations are not meaningful in this case. No child pruning refers to unnecessarily multiplying by $\rho_\Phi(\phi)$ even when no children have downstream connections to a function component, that is Equation B.5. Full reward, does the same as no child pruning, in addition to unnecessarily including upstream function components in the estimator. For full reward, these additional function components are also included in the moving average baseline. Each line represents the average of 5 random seeds with error bars showing 95% confidence interval. Final values at the end of training are written near each line in matching colour. The top row shows the online training ELBO. The bottom row shows the natural logarithm of the mean gradient variance. Mean gradient variance is computed as the mean over parameters and batches of the per-parameter empirical variance over examples in a training batch of 50. It appears that unnecessarily including children has a significant negative impact on f-HNCA with baseline, while the impact of including upstream function components is negligible.

Appendix C

Appendices for Option Iteration

C.1 Motivation for Introducing ElectricProcMaze

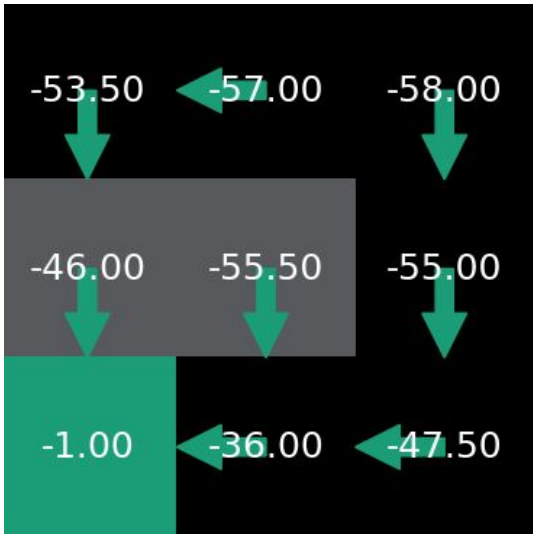

Figure C.1: A simple ElectricProcMaze instance showing the greedy policy (green arrows) and associated action values under the uniform random policy. The values printed on each cell are the action values associated with moving into that cell from any other cell which works in this case since the reward is determined by the cell being entered regardless of the origin. The action values were computed by explicitly solving the Bellman equation with a uniform random behaviour policy. The reward for entering a wall cell is -7 in this case which is one less than the return achievable by following an optimal policy from the furthest cell to the goal. Hence the optimal policy will never enter a wall cell, but the greedy policy with respect to the action values of the uniform random policy does.

ElectricProcMaze is a variant of ProcMaze where instead of remaining in

place upon transitioning into a wall, the agent is allowed to move into the wall cell but with a large negative reward (analogous to an electric shock in animal experiments) set to be equal to one more than the largest possible number of steps required to reach the goal across all possible maze configurations.

My motivation for using ElectricProcMaze rather than ProcMaze stems from observations I made in preliminary experiments on ProcMaze. In particular, I observed that using a random rollout policy (and thus relying on only value function learning to guide the search) was almost as effective as learning a single rollout policy in this case. This can be explained by noticing that due to there being only one path to the goal, ProcMaze has the characteristic that the greedy policy with respect to the action-value function of the random policy is optimal. Laidlaw et al. (2023) have demonstrated that this characteristic is surprisingly common, particularly in environments where standard deep RL algorithms perform well. This does not necessarily mean that policy learning cannot be beneficial, as the number of samples required to evaluate the random policy to sufficient accuracy to reliably identify the greedy action may be much larger than the number of samples required under an improved policy. It does however mean that policy learning is relegated to a more minor role than in the general case where it is necessary for convergence to the optimal policy when bootstrapping from multistep rollouts, regardless of the number of simulations used. This is undesirable when we are interested in investigating the benefits of learning multiple options in order to improve the quality of rollouts.

ElectricProcMaze breaks the condition that the greedy policy with respect to the action-value function of the random policy is optimal. The best action under the uniform random policy will often follow a shorter path to the goal which passes through walls, as doing otherwise will often mean hitting more walls in expectation over a random walk. This is demonstrated on a simple problem instance in Figure C.1. On the other hand, an optimal policy will always avoid walls by design since I set the penalty to be large enough that it is never better to move through a wall than to follow the open path.

C.2 Expert Iteration Loss Ablation

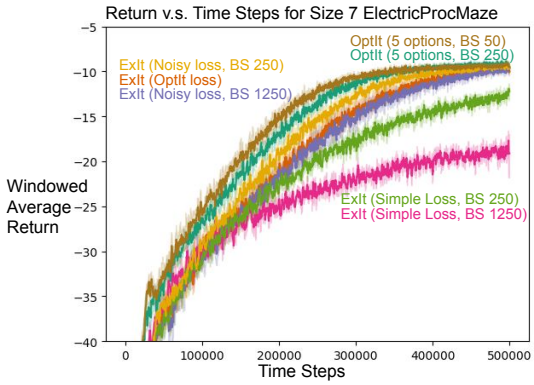

Figure C.2: Windowed average return over training time for various loss function choices for ExIt and OptIt on size 7 ElectricProcMaze. OptIt loss refers to the same loss used to train OptIt which uses samples from the search policy and trains on length 5 sequences. Noisy Loss still uses samples from the search policy but trains on independent samples. Simple Loss trains on the full cross entropy with the search policy on independent samples. BS refers to the batch size used in the gradient update. Error bars show 95% confidence interval over 5 random seeds. The y-axis is thresholded at -40 to omit the rapid period of initial improvement.

In this section, I explore the impact of the choice to optimize the single policy learned by ExIt using the same loss function as OptIt. In particular, this means I optimize ExIt on sequences rather than independent samples, though the latter would be more natural when learning a single policy. It also means I fit the learned policy to samples from the search policy in each update rather than directly minimizing cross-entropy loss. This was necessary for tractability when optimizing over joint cross-entropy for a set of options over a sequence, but not when optimizing a single policy. Note that these choices should not impact the expectation of the loss but do change the noise in the updates. To better understand the impact of these decisions, I present an ablation study evaluating ExIt trained with alternative loss functions. The results are displayed in Figure C.2.

The first variant of ExIt I test directly minimizes cross-entropy on independent samples. Surprisingly, I found that this version was significantly worse than simply using the OptIt loss, to better understand why, I also tried a

version that trains on individual samples instead of sequences but still fits to sampled actions from the search policy rather than directly minimizing the cross-entropy with the search policy in each update. This version recovers similar performance, indicating that the benefit was likely due to sampling the search policy actions as opposed to training on sequences. For each loss, I test ExIt using a batch size of 250 and a batch size of 1250, the latter matches the total number of samples[1] per update to that of the sequence-trained variant with batch size 250 and option rollouts of length 5. I found that the smaller batch size learned faster in all cases. For completeness, I also tested OptIt with 5 options and a batch size of 50 (thus a total of 250 samples per batch when accounting for sequence length), this also learned faster than the original setting of OptIt suggesting that the original batch size was simply suboptimally large for both OptIt and ExIt.

The fact that fitting to samples from the search policy, rather than the expectation, improves performance is curious and may be of independent interest. Further exploration of this may be fruitful, but is beyond the scope of the present work. Throughout, I report results for ExIt and OptIt trained with the same loss, since using the sampled cross-entropy appears significantly beneficial and optimizing on sequences not detrimental for ExIt.

C.3 Investigating the Learned Options for HierarchicalElectricProcMaze

Compared to the learned options for the Compass environment presented in Figure 5.2, I did not find the options learned for HierarchicalElectricProcMaze to facilitate easy interpretation. Nevertheless, here, I make an effort to highlight some of the features of the learned options that may contribute to their improved search performance compared to using a single policy. I focus on the options learned at 500,000 training steps, in the middle of training before the performance has plateaued.

I first plot the behaviour of each of the 5 options across 5 random base-

[1]Modulo truncation due to episode ends.

environment states while varying the controller environment state across the entire grid. The results are displayed in Figure C.3. The option in the far right column shows a strong tendency to move left, but apart from that, as already mentioned, the learned options are not very interpretable. Inspection reveals the policies do display a fair amount of diversity across options.

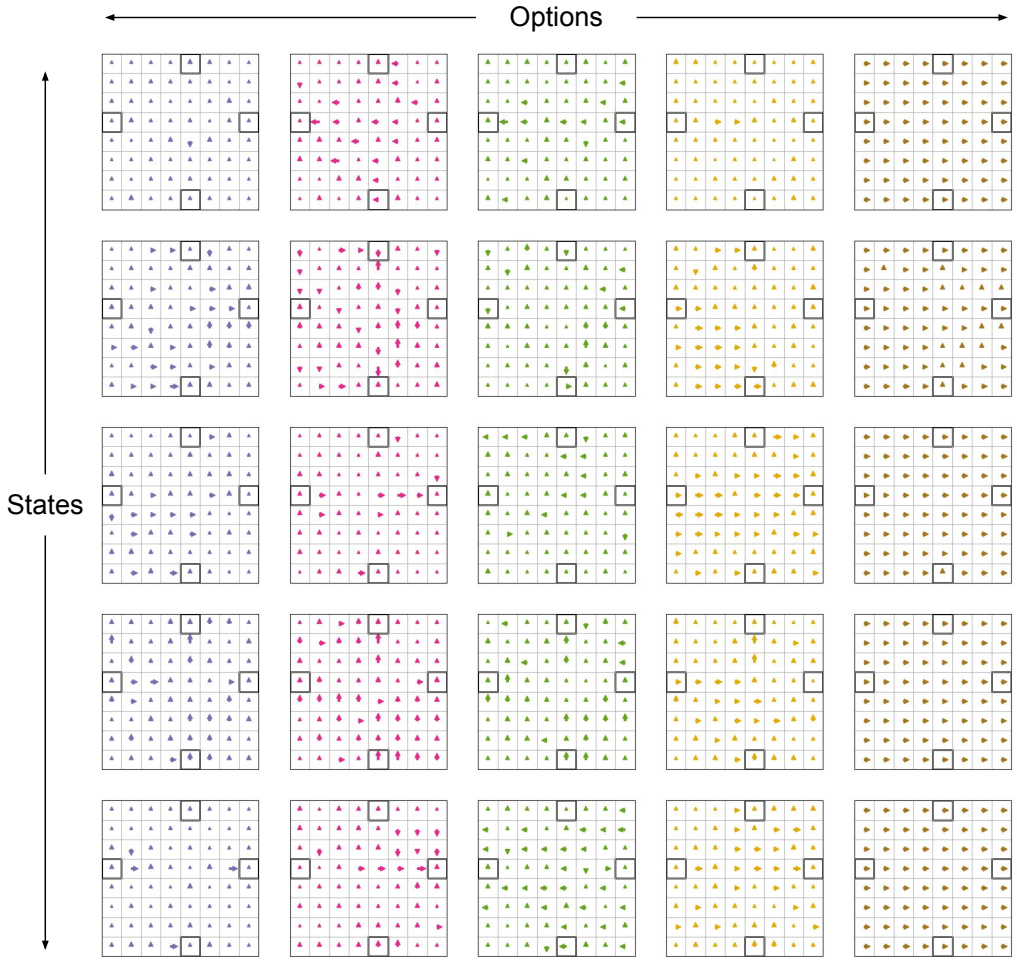

Figure C.3: The 5 options learned by OptIt displayed for each state in the controller-environment grid across 5 different base-environment states in HierarchicalElectricProcMaze. The direction of each arrow shows the highest probability action in that cell, while the length of the arrow indicates the probability of that action with probability 1 corresponding to the arrow touching the edge of the grid cell. Button locations are highlighted with bold squares. Results are displayed for a single random seed corresponding to Seed 1 in Figure C.4.

In order to gauge the degree of behavioural diversity among the learned

options, I randomly initialize 1000 random environment states each in a random maze with the agent in the center of the controller environment. I then perform 1000 length 20 rollouts of each option in each of these random states and note the first button reached (Up, Down, Left or Right). I simply discard rollouts in which no button is reached within 20 steps. A visualization comparing the distribution of buttons reached by different options, as well as the single learned policy of ExIt is show in Figure C.4.

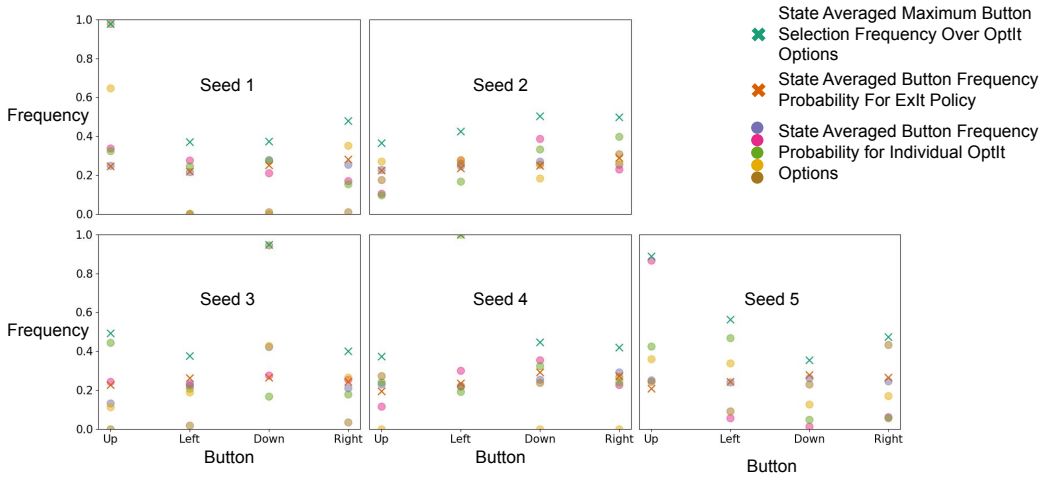

Figure C.4: A visual comparison of the selection frequencies for each of the 4 buttons in the HierarchicalElectricProcMaze controller environment. The values were obtained by initializing the environment in 1000 random positions, each in a random maze with the agent in the center of the controller environment. The green X shows the maximum over options of the frequency with which each button was reached, averaged over states. The orange X similarly shows the frequency with which each button was reached by the single learned policy of ExIt. The coloured dots show the state-averaged button selection frequency for individual OptIt options. Each of 5 random seeds used for training are displayed separately.

Note that in most cases (with the exception of Seed 2), we observe the emergence of at least one option that sharply favours the selection of a particular button. However, unlike in the comparatively simple Compass environment, we do not observe an option specializing to each button across all states. On the other hand, the selection frequencies for the single ExIt policy are generally close to uniform as may be expected. Furthermore, looking at the state-averaged maximum button selection frequency over options (green

X) we see that when fixing the state there appears to be significantly more specialization among options than when we average over states. Note that, the green X is necessarily higher than all of the coloured dots simply because the expectation of a maximum is always greater than or equal to the maximum of an expectation.

More quantitatively, I looked at an empirical estimate of the mutual information[2] between the option selected and the first button reached using the same setup outlined previously. The overall mutual information, without conditioning on state, was 0.19 ± 0.04 compared to the total button selection entropy of 1.31 ± 0.02. Hence, without conditioning on the state, the option selected is a poor predictor of the button reached. However, if we instead look at the mean state conditional mutual information we get 0.36 ± 0.04 compared to the mean state conditional button entropy of 0.87 ± 0.03. Hence, within a particular state, the selected option accounts for a little under half the entropy in the distribution of selected buttons.

Overall, while the learned options do not seem to facilitate straightforward interpretability in this case, they do present significant diversity and appear to be strongly associated with the button selected when conditioning on a particular starting state. This is likely a factor in the improved performance observed for 5-option OptIt compared to ExIt in HierarchicalElectricProcMaze.

[2]To compute this I perform 1000 rollouts for each state-option combination for 1000 states, throwing out all rollouts which did not hit a button within 20 steps. Within the remaining rollouts, I found the overall frequency f_i with which each button was reached as well as $f_{i|n}$, the frequency of each button within those rollouts in which a particular option was used. I estimated the button entropy $\hat{H}(i) \approx \sum_i f_i \log(f_i)$ and option conditional button entropy $\hat{H}(i|n) \approx \sum_i f_{i|n} \log(f_{i|n})$. Finally I estimated the mutual information as $\hat{H}(i) - \frac{1}{N} \sum_n \hat{H}(i|n)$. I estimated the mean state conditional entropy and mutual information similarly except that the frequencies were computed separately for each state, and I computed a final average over the results across all 1000 sampled states.

C.4 Jointly Learning Termination, Option Policies, and the Policy Over Options

In this work, I focus on the case where options are executed and learned for a fixed horizon. However, one could straightforwardly extend this method to the case where termination conditions are learned. Here, I outline an efficient algorithm for this. The approach presented here is functionally equivalent to the algorithm of Fox et al. (2017). However, they propose an explicit forward-backward algorithm which is rather involved. Here, I note that one can achieve the same thing by simply using the forward recursion relationship discussed by Fox et al. (2017) to compute the quantity we need to optimize and then computing the associated gradient using any automatic differentiation framework. I outline this approach explicitly since it may be useful to anyone trying to implement a similar idea.

For the purposes of this section, I assume we have a dataset of length K trajectories $\tau = s_0, a_0, ..., s_K$ which I assume are generated by using a search procedure to select actions at each step. We wish to maximize the log probability of the data under the distribution induced by a learned policy over options, option policies and a termination function. More precisely, I model the trajectories as generated by first using the policy over options $\rho(n|s_0;\theta)$ to choose an initial option, then selecting an action from $\pi_n(\cdot|s_k;\theta)$ for each step until the option terminates. At each step $k \geq 1$, the current option is terminated with probability given by the option termination function $\psi(s_k;\theta)$. If option termination does occur at step k, then a new option is sampled from $\rho(n|s_k;\theta)$ actions are selected for $j \geq k$ from $\pi_n(\cdot|s_j;\theta)$ until the next termination or the end of the trajectory. We seek to optimize the probability $\mathbb{P}(s_0, a_0, ..., s_K;\theta)$ of trajectories in the dataset being generated by this process. Towards this goal, we begin with a recursion relation for $\phi_k(n;\theta) \doteq \mathbb{P}(s_0, a_0, ..., s_k, n_k = n;\theta)$ outlined by Fox et al. (2017), where n_t denotes the option whose policy is used to

select the action at time t. The recursion relation is as follows:

$$\begin{aligned}
\phi_0(n;\theta) &= p_0(s_0)\rho(n|s_0;\theta) \\
\phi_{k+1}(n;\theta) &= \left(\sum_{n'} \phi_k(n';\theta)\pi_{n'}(a_k|s_k;\theta)p(s_{k+1}|s_k,a_k;\theta)\psi_{n'}(s_{k+1};\theta)\rho(n|s_{k+1};\theta)\right) \\
&\quad + \phi_k(n;\theta)\pi_n(a_k|s_k;\theta)p(s_{k+1}|s_k,a_k)(1-\psi_n(s_{k+1};\theta)).
\end{aligned}$$

Intuitively, the first term of the second equation accounts for the probability that some option terminated at time $k+1$ and option n was selected from ρ thereafter. The second term accounts for the probability that option n was already being executed at time k and did not terminate at $k+1$. Now note that we can factor out the transition probabilities, which are not under the agent's control, such that $\phi_k(n;\theta) = \tilde{\phi}_k(n;\theta)p_0(s_0)\prod_{t=0}^{k-1} p(s_{t+1}|s_t,a_t)$ with

$$\begin{aligned}
\tilde{\phi}_0(n;\theta) &= \rho(n|s_0;\theta) \\
\tilde{\phi}_{k+1}(n;\theta) &= \left(\sum_{n'} \tilde{\phi}_k(n';\theta)\pi_{n'}(a_k|s_k;\theta)\psi_{n'}(s_{k+1};\theta)\right)\rho(n|s_{k+1};\theta) \\
&\quad + \tilde{\phi}_k(n;\theta)\pi_n(a_k|s_k;\theta)(1-\psi_n(s_{k+1};\theta)).
\end{aligned} \tag{C.2}$$

We can then find the probability $\mathbb{P}(s_0,a_0,....,s_K;\theta)$ by simply marginalizing out the final option

$$\begin{aligned}
\mathbb{P}(s_0,a_0,....,s_K;\theta) &= \sum_n \phi_K(n;\theta) \\
&= p_0(s_0)\prod_{t=0}^{K-1} p(s_{t+1}|s_t,a_t)\sum_n \tilde{\phi}_K(n;\theta).
\end{aligned}$$

Taking the log of both sides we get

$$\log(\mathbb{P}(s_0,a_0,....,s_K;\theta)) = \log\left(p_0(s_0)\prod_{t=0}^{K-1} p(s_{t+1}|s_t,a_t)\right) + \log\left(\sum_n \tilde{\phi}_K(n;\theta)\right)$$

Note that the left term is independent of θ hence we get

$$\frac{\partial}{\partial\theta}\log(\mathbb{P}(s_0,a_0,....,s_K;\theta)) = \frac{\partial}{\partial\theta}\log\left(\sum_n \tilde{\phi}_K(n;\theta)\right).$$

Having computed $\sum_n \tilde{\phi}_K(n;\theta)$ via the forward recursion in Equation C.2, we can simply backpropagate to compute its gradient using any standard automatic differentiation framework.[3] The computation required for each step of

[3]In practice, one may want to implement the recursion in Equation C.2 in log-space, using a numerically stable implementation of LogSumExp to compute $\log(\tilde{\phi}_{k+1}(n;\theta))$ at each step.

the forward recursion is dominated by a sum over N options, which can be computed once and shared across Equation C.2 for all n, and it's unrolled for K steps hence the total complexity is on the order of $\mathcal{O}(KN)$, the order of the backward pass is the same. Note that the order of complexity is the same as computing the likelihood under a set of options and associated option policy even without termination conditions, so including termination conditions is no harder in that regard. However, without termination, we can parallelize over time steps which is no longer possible with termination conditions as we have to compute Equation C.2 sequentially.

The learned termination conditions, in combination with the learned option policies, could be utilized for planning in various ways. One simple approach would be to incorporate option termination into the rollouts. The MCS planner would select the initial option and action for each rollout, but at each subsequent step the option termination function would be queried and the current option terminated with probability $\psi_{n'}(s_{k+1}; \theta)$. When termination occurs in a rollout a new option would be selected according to ρ as normal. This would allow for greater flexibility, as the learned options wouldn't necessarily need to be useful for the entirety of an arbitrary rollout horizon but could be switched mid-rollout based on the environment observations. For example, in HierarchicalElectricProcMaze it might be desirable to terminate the current option upon reaching one of the buttons in the controller environment.

C.5 Hyperparameters

Table C.1 shows the hyperparameters used in the Compass and ElectricProcMaze experiments. For ElectricProcMaze, the step-size parameter α and entropy-regularization factor β were tuned for the ExIt baseline from $\alpha \in \{0.0000625, 0.000125, 0.00025, 0.0005, 0.001\}$ and $\beta \in \{0.01, 0.1, 1.0\}$. I fix the option step-size (policy step-size for ExIt) to α and set the value step-size to 2α. I evaluate each hyperparameter combination over 5 random seeds and choose the one with the best average return over the last 100,000 of 500,000 time steps. I display sensitivity curves resulting from this grid search in Fig-

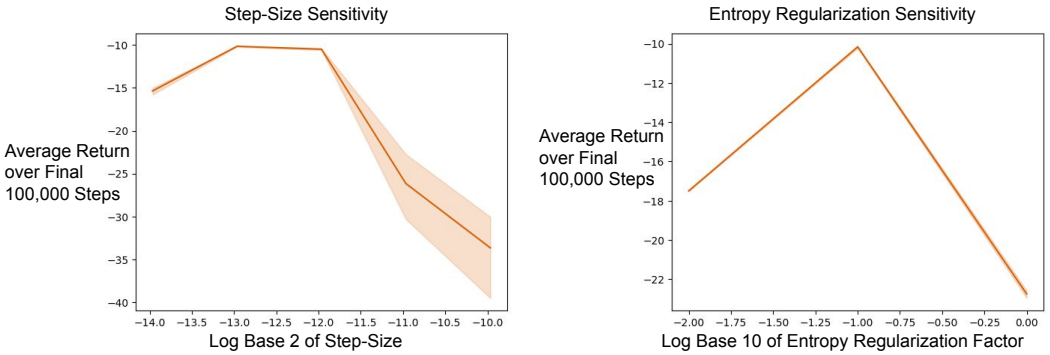

Figure C.5: Sensitivity curves for ExIt resulting from grid sweep over step size α and entropy-regularization factor β in size 7 ElectricProcMaze. In each plot, the other hyperparameter is fixed to its best value from the grid search while varying the hyperparameter of interest. Error bars show 95% confidence interval over 5 random seeds.

ure C.5. Other hyperparameters were fixed to reasonable defaults.

For Compass, I mainly kept the same values from ElectricProcMaze with a few deliberate exceptions. I drastically reduced the simulation budget such there would be insufficient rollouts under a random policy to be likely to sample the optimal action sequence. I increased the option rollout length such that it was possible for rollouts to reach any edge of the grid from any starting location. I reduced the number of options to 4 to emphasize that 4 options were sufficient in this case. I also reduced β to 0.01 to facilitate convergence to near-optimal performance and cleaner learned options. At $\beta = 0.1$, OptIt still performed significantly better than ExIt with a final average return of around 0.9, and the learned options still showed a tendency to prefer one of the 4 cardinal directions each but became significantly more chaotic.

For HierarchicalElecticProcMaze, I also used most of the same hyperparameters from ElectricProcMaze. I increase the option rollout length to 8 to match the width of the control grid. I also doubled the number of hidden units as I found in preliminary experiments that this significantly improved the performance of ExIt indicating that the 400 hidden unit network was significantly underparameterized for the more challenging problem. Finally, I found it was necessary to reduce the entropy regularization to observe a performance benefit for OptIt. To facilitate a fair comparison, and a more

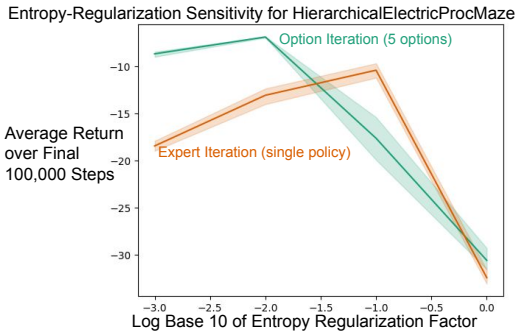

Figure C.6: Sensitivity curve for ExIt and OptIt resulting from grid sweep entropy-regularization factor β in HierarchicalElectricProcMaze. OptIt achieves optimal performance at a lower entropy regularization and remains robust when lowering the regularization further. Error bars show 95% confidence interval over 5 random seeds.

complete picture of the behaviour of each approach, I performed a sweep over $\beta \in \{0.001, 0.01, 0.1, 1.0\}$. The results of the sweep are displayed in Figure C.6.

Hyperparameter	**ElectricProcMaze**	**HierarchicalElectricProcMaze**	**Compass**
Number of Hidden Layers	3	—	—
Number of Hidden Units	400	800	400
Hidden Activation	ELU	—	—
Optimizer	AdamW	—	—
Adam β_1	0.9	—	—
Adam β_2	0.99	—	—
Adam ϵ	1e-5	—	—
Adam Weight Decay	1e-6	—	—
Option Step-Size	1.25e-4 **(Tuned for ExIt)**	—	—
Value Function Step-Size	2.5e-4 (Twice Above)	—	—
Running Average Variance Decay	0.99	—	—
Discount Factor	0.99	—	—
Batch Size	250	—	—
Number of Parallel Workers	16	—	—
Gradient Updates per Environment Step	16	—	—
Entropy-Regularization Factor (β)	0.1**(Tuned for ExIt)**	0.1 (ExIt)/0.01 (OptIt)	0.01
Simulation Budget	1000	—	50
Option Rollout Length (K)	5	8	20
Number of Options	5	—	4
Buffer Size	100,000	—	—
Training Start Time	100	—	—

Table C.1: Table of hyperparameters used in Compass and ElectricProcMaze experiments. Dashes denote that the same value is used as ElectricProcMaze.