Selective Dyna-style Planning Using Neural Network Models with Limited Capacity

by

Muhammad Zaheer

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

In model-based reinforcement learning, planning with an imperfect model of the environment has the potential to harm learning progress. But even when a model is imperfect, it may still contain information that is useful for planning. In this thesis, we investigate the idea of using an imperfect model *selectively*: the agent should plan in parts of the state space where the model would be helpful but refrain from using the model where it would be harmful. An effective selective planning mechanism needs to account for at least three sources of model errors: stochastic dynamics of the environment, insufficient coverage of the state space, and limited capacity to model the dynamics. Prior work has used *parameter uncertainty* for selective planning, where the estimated uncertainty signals the errors due to insufficient coverage. In this work, we emphasize the importance of *structural uncertainty* that signals the errors due to limited capacity; we show that the learned input-dependent variance, under the standard Gaussian assumption, can be interpreted as an estimate of structural uncertainty. We empirically evaluate the ability of the learned variance to help plan selectively under limited capacity. The results show that selective planning with the learned variance can be useful, even when planning with the model non-selectively would cause catastrophic failure.

"Having done nothing, I had nothing to lose. Having made a happy life without having achieved anything at all artistically, I found that any artistic achievement was a bonus. Having finally conceded that I wasn't a prodigy after all, I had the total artistic freedom that is afforded only to the beginner, the doofus, the aspirant."

– George Saunders, CivilWarLand in Bad Decline.

Acknowledgements

First and foremost, I would like to thank my supervisors, Martha White and Erin Talvitie, for their continued support and guidance. Martha has been a constant source of inspiration and reassurance; all what I have learned as a graduate student I owe, to a great extent, to Martha. Erin's expertise on model-based reinforcement learning, and her outlook on research in general, has time and again proven to be priceless.

I would like to thank Rich Sutton for his wisdom; amongst many things, he taught me how to separate a problem from its candidate solutions, and I have come to appreciate this invaluable thinking tool. I am also thankful to Adam White for teaching me how to be a useful member of the scientific community; he taught me the discipline for accomplishing meaningful empirical work, and the essence of writing thoughtful reviews.

I am thankful to my fellow graduate students from the RLAI group for the illuminating discussions and, more importantly, for their companionship. Finally, I would like to give a shout-out to Abhishek Naik, Khurram Javed, Yi Wan, Chenjun Xiao, Samuel Sokota, Sungsu Lim, Vincent Liu, Wesley Chung, Ehsan Imani, Taher Jaferjee, Raksha Kumaraswamy, Niko Yasui, Matthew Schlegel, and Roshan Shariff for making graduate school a fun experience.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reinforcement learning is a computational approach to learning via interaction. An algorithmic agent is tasked with determining a behavior policy that yields a large cumulative reward. Generally, the framework under which this agent learns its policy falls into one of two groups: model-free reinforcement learning or model-based reinforcement learning. In model-free reinforcement learning, the agent acts in ignorance of any explicit understanding of the dynamics of the environment, relying solely on its state to make decisions. In contrast, in model-based reinforcement learning, the agent possesses a *model* of how its actions affect the future. The agent uses this model to reason about the implications of its decisions and *plan* its behavior.

The model-based approach to reinforcement learning offers significant advantages in two regimes. The first is domains in which acquiring experience is expensive. Model-based methods can leverage planning to do policy improvement without requiring further samples from the environment. This is important both in the traditional Markov decision process setting, where sample efficiency is often used as a performance metric, and also in a more general pursuit of artificial intelligence, where an agent may need to quickly adapt to new goals. Second is the regime in which capacity for function approximation is limited and the optimal value function and policy cannot be represented. In such cases, agents that plan at decision-time can construct temporary local value estimates whose accuracy exceed the limits imposed by capacity restriction (Silver *et al.* 2008). These agents are thereby able to achieve superior

policies than similarly limited model-free agents.

Far from being special cases, sample-sensitive, limited-capacity settings are typical of difficult problems in reinforcement learning. It is therefore not surprising that many of the most prominent success stories of reinforcement learning are model-based. In the Arcade Learning Environment (ALE) (Bellemare *et al.* 2013), algorithms that distribute training across many copies of an exact model of the environment have been shown to massively outperform algorithms limited to a single instance of the environment (Kapturowski *et al.* 2019). And in Chess, Shogi, Go, and Poker, superhuman performance can be reached by means of decision-time planning on exact transition models (Brown and Sandholm 2017; Moravčík *et al.* 2017; Silver *et al.* 2018).

However, the premise of these successes is subtly different from the classical reinforcement learning problem. Rather than being asked to learn a model from interactions with a black box environment, these agents are provided an exact model (or many exact models) of the dynamics of the environment. While the latter is in itself an important problem setting, the former is more central to the pursuit of broadly intelligent agents.

Unfortunately, learning a useful model from interactions has proven difficult. While there are some examples of success in domains with smooth dynamics (Deisenroth and Rasmussen 2011; Hafner *et al.* 2019), learning an accurate model in more complex environments, such as the ALE, remains out of reach. In a pedagogical survey of the ALE, Machado *et al.* (2018) state "So far, there has been no clear demonstration of successful planning with a learned model in the ALE." Basic non-parametric models that replay observed experiences (Schaul *et al.* 2016) remain convincingly superior to state-of-the-art parametric models (Hasselt *et al.* 2019).

Some of the difficulty of increasing performance with a learned model arises from the twofold nature of the problem. First, learning a useful model is a difficult challenge. It is impractical to model transitions over observation space in complex environments but learning a more compact representation remains difficult. Second, it is not clear when and how an imperfect model is best used. Using an imperfect model can be catastrophic to progress if it is incorrectly

trusted by the agent.

In this work, we concern ourselves with the latter problem: how to effectively use imperfect models. We discuss planning methods that only use the model where it makes accurate predictions. Such techniques should allow the agent to plan in regions of the state space where the model is helpful but refrain from using the model when it would be damaging. We refer to this idea as *selective planning*.

There are two interrelated problems involved in selective planning: determining when the model is and is not accurate, and devising a planning algorithm which uses that information to plan selectively. In this thesis, we address these problems as follows:

- We formulate the first problem as that of uncertainty estimation. We discuss three types of uncertainty: parameter, aleatoric, and structural, and emphasize the relevance of structural uncertainty for selective planning under limited-capacity. We empirically investigate specific methods that let us represent uncertainty in the context of neural networks, and evaluate their effectiveness under the limited-capacity setting. We demonstrate that the learned input-dependent variance — under the standard Gaussian assumption (Nix and Weigend 1994), for instance — can reveal the presence of structural uncertainty.

- We address the second problem by empirically investigating selective planning in the context of Model-based Value Expansion (MVE), a planning algorithm which uses the learned model to construct multi-step TD targets for evaluating the greedy policy (Feinberg *et al.* 2018); we show that MVE can fail when then model is subject to capacity constraints. We study the performance of Selective MVE, an instance of selective planning which weights the multi-step TD targets according to the structural uncertainty in the model's predictions. Our findings show that the idea of selective planning is promising: selective planning can result in sample-efficient learning even with an imperfect model that otherwise leads to planning failures.

This thesis is organized into 7 chapters. Chapter 2 discusses relevant background concepts. Chapter 3 discusses the three types of uncertainty. Chapter 4 uses a simple regression problem to highlight the importance of structural uncertainty for expressing model errors due to limited-capacity. Chapter 5 contains the results of our empirical investigations into selective planning with the proposed structural uncertainty method. Chapter 6 emphasizes the benefits of structural uncertainty over parameter uncertainty for selective planning under the limited-capacity setting. Finally, Chapter 7 concludes the thesis with a discussion of possible future work.

Chapter 2

Background

In this chapter, we will briefly review some of the key reinforcement-learning concepts. We will also introduce the relevant notation.

2.1 Learning as Reward Maximization

Reinforcement learning (RL) is a computational approach to learning from interaction (Sutton and Barto 2018). An *agent* interacts with its *environment* to determine a behavior that maximizes a special signal, called the *reward*. Formulating goals as maximization of a reward signal is one of the central ideas of reinforcement learning.

2.2 Markov Decision Processes

A reinforcement-learning problem is typically formulated as a finite Markov Decision Process (MDP) (Puterman 2014). An MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$: $\mathcal{S}$ is the set of states; $\mathcal{A}$ is the set of actions; $\mathcal{R}$ is the set of rewards; p denotes the dynamics of the environment such that $p(s', r|s, a) \doteq Pr(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$, for all $s', s \in \mathcal{S}, a \in \mathcal{A}$, and $r \in \mathcal{R}$; $\gamma : (\mathcal{S}, \mathcal{A}, \mathcal{S}) \rightarrow \mathbb{R}^+$ is the discount function.

At each time-step t, the environment is in some *state* $S_t \in \mathcal{S}$; the agent executes an *action* $A_t \in \mathcal{A}$; the environment responds with some reward $R_{t+1} \in \mathcal{R}$ and transitions to a new state $S_{t+1} \in \mathcal{S}$. This agent-environment interaction gives rise to a stream of experience: $... S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, ...$

The agent chooses actions according to a *policy* $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, which maps states to probabilities of selecting each possible action [1]. The goal of reinforcement learning is to use this experience to learn a *policy* π_* that maximizes the future reward.

2.3 Value Functions

Value functions underlie most RL solution methods. A value function summarizes the reward consequences of future behavior into a single number. The value of a state s under a policy π, denoted as $v_\pi(s)$, is the discounted cumulative reward the agent is expected to receive when it starts in s and follows π thereafter:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right]$$

The action-value function of a state s and action a under a policy π, denoted as $q_\pi(s, a)$, is the discounted cumulative reward the agent is expected to receive if it starts in s, takes the action a, and follows π thereafter:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]$$

There exists a unqiue value function, referred to as the *optimal value function* v_*, which maximizes the value over all states:

$$v_*(s) \doteq \max_\pi v_\pi(s)$$

for all $s \in \mathcal{S}$. All policies that share the optimal value function are optimal; an *optimal policy* can be written as:

$$\pi_* \doteq \arg\max_\pi v_\pi(s)$$

Optimal policies also share the same action-value function q_*, which can be written as:

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

[1] If the policy is deterministic, we can alternatively write it as a function $\pi : \mathcal{S} \rightarrow \mathcal{A}$

Value-based reinforcement methods typically estimate the optimal action-value function as a means to an optimal policy; for instance, a determinstic policy $\pi : \mathcal{S} \to \mathcal{A}$ that is greedy with respect to q_* is an optimal policy:

$$\pi_*(s) = \arg\max_a q_*(s, a)$$

2.4 Bellman Equations and Dynamic Programming

Bellman equations are at the heart of reinforcement learning solution methods. One of the key properties of Bellman equations is that they relate the value of a state (or a state-action pair) to that of its successor states (or successor state-action pairs), paving a way for learning to be online and incremental. Consider, for instance, the Bellman equation of the action-value function q_π of a policy π:

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)\Big[r + \gamma \sum_{a'} \pi(a'|s')q_\pi(s', a')\Big]$$

If the environment's dynamics p are known, then the above Bellman equation can be mechanized into a *policy evaluation* algorithm for estimating the action-value function of an arbitrary policy π — 1) initialize an action-value function estimate $Q(s, a)$ for all (s, a) pairs, 2) loop over all (s, a) pairs and update $Q(s, a)$ towards the quantity $\sum_{s',r} p(s', r|s, a)[r+\gamma \sum_{a'} \pi(a'|s')Q(s', a')$, 3) halt when estimates change only negligibly. The essence of this algorithm is in the following update equation:

$$Q(s, a) \leftarrow \sum_{s',r} p(s', r|s, a)\Big[r + \gamma \sum_{a'} \pi(a'|s')\ Q(s', a')\Big]$$

An application of the above update equation is also referred to as a *full backup* — it uses the dynamics model to perform a full-width lookahead over all possible state transitions. Now consider the Bellman equation for the optimal action-value function q_*:

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big]$$

The above Bellman equation leads to a *value iteration* algorithm for estimating the optimal action-value function, which has the following update equation:

$$Q(s,a) \leftarrow \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \max_{a'} Q(s',a')\Big]$$

If the above update equation is used for the action values of all state-action pairs infinitely many times, the estimated Q converges to q_*.

Value iteration combines policy evaluation with *policy improvement* — the policy to be evaluated is always greedy with respect to the currently estimated Q. Value iteration can be interpreted as an instance of generalized policy iteration (GPI): a general idea which refers to policy evaluation and policy improvement working in tandem to find the optimal value function and an optimal policy (Sutton and Barto 2018).

Policy evaluation and value iteration are instances of *dynamic programming* (DP), a class of methods which can compute quantities such as q_π and q_* if given a perfect model of the environment's dynamics. DP methods are *model-based*: they primarily rely on planning with a model to estimate value functions; while interactions with the environment may be needed as means to a model, no further interaction is needed once the model has been obtained. DP methods also *bootstrap*: they update the value estimates of states on the basis of the value estimates of successor states. Bootstrapping is central to many RL algorithms as it enables the learning to be online: the estimates can be improved without waiting for the final outcome to be known — the sum of future rewards, for example. For the learning to progress, the successor states and the rewards along the way are all that are needed.

2.5 Q-learning

While DP methods require a perfect model of the environment's dynamics, methods based on *temporal-difference* (TD) learning (Sutton 1988) can estimate value functions *model-free*; that is, TD methods learn directly from raw experience without an explicit model of the environment's dynamics. Consider, for instance, Q-learning (Watkins 1989) — a prototypical temporal-

difference learning algorithm for estimating q_*. Given a sample transition, (S_t, A_t, R_t, S_{t+1}), Q-learning updates the action values according to the update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \qquad (2.1)$$

where α is the step-size parameter.

2.6 Dyna

Dyna is an approach to model-based RL which unfies the essential ideas of dynamic programming and temporal difference learning. A planning agent uses real experience, acquired by interacting with the environment, to update the value function with TD learning; in addition, the real experience is used to learn a model of the environment, which is then used to simulate additional experience; the value function is updated with the simulated experience in the same way as with the real experience — with TD learning.

DynaQ (Sutton 1991) is a prototypical Dyna algorithm. DynaQ uses the Q-learning update rule (Equation 2.1) for both learning from real experience and planning from simualted experience. For every real transition, DynaQ updates the value function using the Q-learning update rule; the same transition is also used to update the model, to make it more consistent with the real dynamics of the environment. Then, planning proceeds until some computational budget runs out. During planning, a state, S, is sampled from the set of previously seen states; an action, A, is sampled from the set of actions previously taken in S; the model is used to predict the next state, S', and reward, R'; the resulting simulated transition, (S, A, R, S'), is used to update the value function using the same Q-learning update (Equation 2.1).

Dyna allows flexible planning with its *search control*: a process that selects the starting states and actions for the simulated experiences generated by the model. Planning efficiency can be significantly improved if the planning budget is focused on particular state-action pairs. For instance, *prioritized sweeping* (Moore and Atkeson 1993; Peng and Williams 1993) improves

planning efficiency by prioritizing the predecessor states of state-action pairs whose estimated values have changed. Search control makes Dyna an attractive framework for selective planning: planning computations can be focused on states and actions for which the model makes accurate predictions.

2.7 Value-Function Approximation

Our discussion so far has focused on the *tabular solution methods*: that in which the state and action spaces are represented as a *table*. In many interesting problems, the size of the state space, $\mathcal{S}$, or the state-action space, $\mathcal{S} \times \mathcal{A}$, it too large for the tabular methods to be applicable. *Approximate solution methods* extend the key ideas of reinforcement learning to large state spaces by means of function approximation.

Consider, for instance, the optimal action-value function. The optimal action-value function can be approximated using a function parametrized by a weight vector $\mathbf{w} \in \mathbb{R}^d$: $\hat{q}_{\mathbf{w}}(s,a) \approx q_*(s,a)$. Learning, in this case, involves the use of real experience to adjust the parameters so as to reduce the approximation error. Semi-gradient Q-learning, a variant of Q-learning for function approximation, is an example of a learning algorithm which, given a transition, $(S_t, A_t, R_{t+1}, S_{t+1})$, adjusts the parameters according to the update rule:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \Big[R_{t+1} + \gamma \max_a \hat{q}_{\mathbf{w}_t}(S_{t+1}, a) - \hat{q}_{\mathbf{w}_t}(S_t, A_t) \Big] \nabla_{\mathbf{w_t}} \hat{q}(S_t, A_t) \qquad (2.2)$$

where $\nabla_{\mathbf{w_t}} \hat{q}(S_t, A_t)$ is the gradient of $\hat{q}(S_t, A_t)$ with respect to the weight vector $\mathbf{w}$ at time-step t.

A simple approach to function approximation is the use of *linear functions*. Under linear function approximation, the approximate function is linear in the features — $\hat{q}_{\mathbf{w}}(s,a) \doteq \mathbf{w}^\top \mathbf{x}(s,a)$, where $\mathbf{x} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^m$ is the feature extractor. The gradient $\nabla_{\mathbf{w}} \hat{q}(s,a)$, in this case, is simply the feature vector $\mathbf{x}(s,a)$.

While the class of linear functions is well-understood, its effectiveness greatly depends on the feature mapping $\mathbf{x}$. An alternative is to use a non-linear function approximator, such as a neural network, for value function

approximation. On the other hand, while neural networks offer a rich class of functions, they can be difficult to optimize due to nonstationarity, bootstrapping, and delayed targets — a combination of challenges unique to reinforcement learning. In the next section, we discuss Deep Q-Networks (DQN) (Mnih *et al.* 2015), an algorithm which uses a neural network for value-function approximation.

2.8 Deep Q-Networks

Deep Q-Network (DQN) has achieved impressive performance on Arcade Learning Environment (ALE) (Bellemare *et al.* 2013), a suite of Atari 2600 games designed to study reinforcement learning algorithms. It is a semi-gradient Q-learning algorithm which approximates the value function using a deep (multi-layer) neural network trained via backpropagation (Rumelhart *et al.* 1986). In order to stabilize the learning process under nonstationarity and bootstrapping, DQN relies on two cruical pieces: *experience replay* and *target networks* — which effectively bring Q-learning closer to the simpler supervised-learning setting in which neural networks tend to work reliably.

Instead of updating the value function online with every transition, as done in Q-learning, DQN places the transitions into an *experience replay buffer* (Lin 1992). A batch of transitions is sampled from the replay buffer and the value function is updated using the semi-gradient Q-learning update (Equation 2.2). Experience replay offers several cruicial benefits: it temporally decorrelates the updates as the transitions are randomly sampled from the buffer, it enables smooth sample gradients by means of a batch of randomly sampled transitions, and, finally, it allows for more efficient use of real experience — a particular transition is used to update the value function several times before it is removed from the replay buffer.

DQN addresses the optimization challenges caused by bootstrapping by employing a target network, parameterized by the weight vector $\mathbf{w}^- \in \mathbb{R}^d$. Every K steps, the parameters, $\mathbf{w}$, of the approximate value function, $\hat{q}_{\mathbf{w}}$, are copied to the target network. For the next K steps, the target network is

used to provide Q-learning update targets: $R_{t+1} + \gamma \max_a \hat{q}_{\mathbf{w}^-}(S_{t+1}, a)$. Target network ensures that the update targets are stationary during the interval in which the target network parameters remain fixed.

2.9 Experience Replay as Dyna-style Planning

RL algorithms with experience replay mechanisms can be well thought of as instantiations of the Dyna framework: planning involves improving the approximate value function by replaying transitions from the experience replay buffer — a non-parametric model of the environment's dynamics (Hasselt *et al.* 2019).

However, there are limited benefits to simply replaying stored experiences. For instance, a replay buffer cannot be used to simulate novel experiences as it only stores what the agent has already experienced. On the other hand, a parametric model that generalizes meaningfully can be used to generate new experiences, which may involve actions that have not yet been taken, or states that have not yet been visited. Planning with a parametric model, which can simulate novel experiences, can be significantly more sample-efficient than simply replaying stored experiences (Holland *et al.* 2018).

2.10 Model-based Value Expansion

Model-based Value Expansion (MVE) (Feinberg *et al.* 2018) can be seen as an extension of DQN in which the model-simulated experience is used to evaluate the greedy policy. DQN samples a batch of transitions from the replay buffer to update the value estimates of state-action pairs towards the TD target constructed with the rewards and the values of the next states. In contrast, MVE samples a batch of transitions from the replay buffer, and simulates — for each state-action pair in the sampled batch — an H-step trajectory using the learned model and the greedy policy. The simulated trajectories are then used to construct multi-step TD-targets towards which the value estimates of the state-action pairs are updated. If the model is accurate, model-based *value expansion* can help reduce the bias of the updates, potentially improving

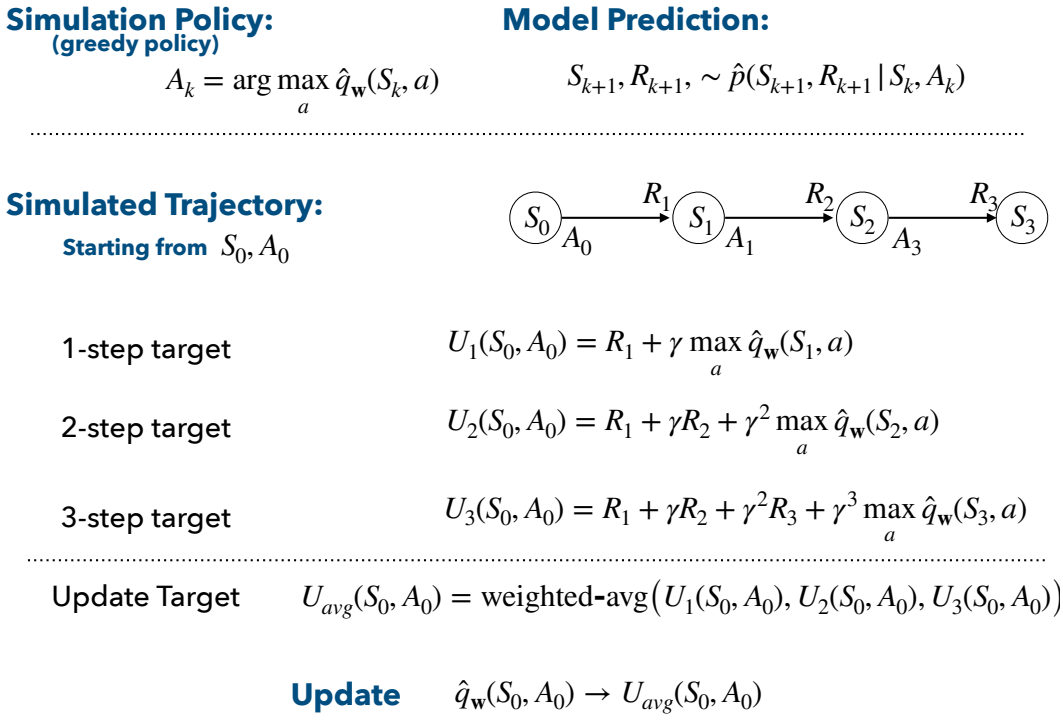

Figure 2.1: A pictorial representation of MVE: a model is used to simulate a trajectory. The simulated trajectory is used to construct multi-step TD targets for evaluating the greedy policy. The TD-targets are combined using a weighted average — the weight can be distributed uniformly on all targets, for example. The approximate action-value is updated towards the weighted average.

sample-efficiency.

The essence of MVE is depicted in Figure 2.1 using a single state-action pair as an example. Given a state-action pair, (S_0, A_0)[2], selected for a planning update by search control, a 3-step trajectory is simulated by the model using the greedy policy implied by the approximate action value function $\hat{q}_{\mathbf{w}}$. The simulated trajectory is used to construct TD-targets: a 1-step target, $U_1(S_0, A_0)$, a 2-step target, $U_2(S_0, A_0)$, and a 3-step target, $U_3(S_0, A_0)$. More generally, an h-step target U_h can be written as:

[2]In this case, the subscript zero of (S_0, A_0) should not be confused with the real time-step of the agent-environment interaction. We use zero to indicate the state-action pair sampled using *search control* — random sampling from the replay buffer, for example. The value of the sampled state-action pair are updated using the simulated experience.

$$U_h(S_0, A_0) = \sum_{k=1}^{h} \gamma^{k-1} R_k + \quad \gamma^h \max_{a \in \mathcal{A}} \hat{q}_{\mathbf{w}}(S_h, a) \tag{2.3}$$

The TD-targets are combined into $U_{avg}(S_0, A_0)$ by computing the weighted average; for instance, the weight can be distributed uniformly on all targets, or all weight can be put exclusively on the target obtained by bootstrapping from the last simulated state. Finally, the estimated value of the state-action pair, $\hat{q}_{\mathbf{w}}(S_0, A_0)$, is updated towards $U_{avg}(S_0, A_0)$ — by the semi-gradient update, for example.

2.11 Model Learning as Supervised Learning

A model of the environment can be classified into one of the three types: distribution models, sample models, or expectation models. Given a state feature vector and an action: a distribution model produces a distribution over the next-state feature vectors and rewards, a sample model generates a sample of the next-state feature vector and reward, and an expectation model produces the *expectation* of the next-state feature vector and reward. Expectation models have been used to extend Dyna to linear function approximation (Asadi 2015; Sutton *et al.* 2008). While expectation models are relatively easier to learn, they introduce bias in TD updates if the environment is stochastic and the value function is non-linear (Wan *et al.* 2019). For the special case of deterministic environments, expectation models have been used with non-linear function approximation (Oh *et al.* 2015). Feinberg *et al.* (2018) also use expectation models to evaluate MVE, a flavor of Dyna-style planning, in deterministic environments. In this thesis, we develop selective planning in the context of expectation models and use non-linear value functions; as a result, we simulate experience only for the state-action pairs for which the dynamics are deterministic.

Learning an expectation model of next-state features can be well thought of as a supervised learning problem. Consider $\mathcal{D} = \{(\mathbf{x}(S_i), A_i, \mathbf{x}(S'_i))\}_{i=1}^{N}$, a *training-set*, generated by interacting with an environment with dynamics

p, using some policy π. Our goal is to find a predictor $f : \mathbb{R}^d \times \mathcal{A} \to \mathbb{R}^d$, from some function class $\mathcal{F}$, which, given a state feature vector, in $\mathbb{R}^d$, and an action, in $\mathcal{A}$, predicts the expected next-state feature vector, in $\mathbb{R}^d$.

In particular, we are interested in learning a parametric model using a neural network — we would like to find a predictor in a neural network's function class $\mathcal{F}$ that consists of all functions representable by the network with its allowable parameter configurations. The model parameters can be chosen by minimizing the *empirical risk* (Vapnik 1992): find a weight vector $\theta \in \mathbb{R}^m$ that minimizes the loss on the training set.

$$\theta^* = \min_{\theta \in \mathbb{R}^m} \mathbb{E}_{\mathcal{D}} \Big[\ell \Big(f_\theta \big(\mathbf{x}(S), A \big), \mathbf{x}(S') \Big) \Big] \tag{2.4}$$

where ℓ is a loss function. For squared loss, the optimal model is the expectation model (Geman *et al.* 1992): the deterministic function of $\mathbf{x}(S)$ and A which predicts the expected $\mathbf{x}(S')$.

2.12 Stochastic Ensemble Value Expansion

The idea of selective planning is not novel. A particularly prominent approach is Stochastic Ensemble Value Expansion (STEVE) (Buckman *et al.* 2018). STEVE is an extension of MVE that uses an ensemble of neural networks to estimate parameter uncertainty. Intuitively, individual networks in an ensemble should make similar predictions in the regions of the state space where sufficient samples have been observed while making dissimilar predictions elsewhere. The degree of agreement can be used as a proxy for the trustworthiness of the model in a particular area of the state space. STEVE uses the estimated parameter uncertainty to weight the multi-step TD targets: it uses a small weight for a target with high parameter uncertainty, and vice versa.

However, we argue that paramater uncertainty may not offer a sufficient measure of whether the model is trustworthy, and that it needs to be used in tandem with structural uncertainty. In case of a neural network ensemble, for instance, a set of networks with insufficient capacity may reach an agreement

on a solution that efficiently allocates capacity but does not match the observed data. Being aware of this type of error is critical for selective planning, especially due to the fact that it is unreasonable to assume that the model will have sufficient capacity in a complex environment. In this thesis, we highlight the importance of structural uncertainty for selective planning under limited capacity.

Chapter 3

Types of Uncertainty

There are many reasons why the model may be inaccurate in certain situations: the agent may as yet not seen sufficient data, or the outcomes are highly stochastic, or there is too much complexity to model.

Consider an agent learning to navigate. The agent can take an action for each of the four cardinal directions: up, down, left, and right. An action moves the agent in the intended direction unless the agent is in front of an obstacle, in which case the agent stays where it is. The agent begins to roam around the world, and decides to build a model with its stream of experience. Crucially, the agent has not yet had a face-off with any of the obstacles; as a result, the data used to build the model does not include any information about what the agent should expect when it is facing an obstacle. What should the model predict when the agent faces an obstacle and plans to move towards it?

In other cases, the outcome is stochastic. Imagine a world with slippery surfaces. When the agent attempts to move in one direction while it is on a slippery surface, it has a nonzero chance of slipping in the opposite direction. What should happen when the agent uses the model to make predictions about its movement on slippery surfaces?

The world is also likely to be more complex than what the agent can model. Consider a navigation problem in which the agent senses the x, y coordinates representing its position, and the obstacles are arranged in a way that both coordinates jointly determine the next position. Imagine that the agent can only afford a factored model that cannot take both dimensions into account

jointly. To get the majority of the predictions correct, the model completely ignores the presence of the obstacles. What should happen when the agent faces an obstacle in this case?

In all of these cases, the model is likely to make arbitrary predictions which can cause planning failures. It is therefore desirable that the model conveys some quantity which lets the agent distinguish such cases from the ones in which the model predictions can be used safely. That is, the model should not only predict the future but also convey its level of *uncertainty* in the prediction it makes. In this chapter, we discuss different sources of uncertainty and the methods to estimate them.

3.1 Sources of Uncertainty

While the three example scenarios outlined earlier are similar in that they all bring attention to the need of mechanisms which account for uncertainty in the predictions, they are exemplars of the distinct types of uncertainty which may require separate treatment in terms of formalism and solution methods. In the following, we discuss the sources of uncertainty, and label the situations in which it is desirable for the model to express low confidence in its predictions.

3.1.1 Stochasticity

The inherent stochasticity of the data generating process can contribute an irreducible component to a model's uncertainty in its predictions. Consider, for instance, the roll of a die: the probability of each face may be nonzero, making the outcome truly random.

This type of uncertainty is referred to as *aleatoric uncertainty*. Aleatoric uncertainty is irreducible in that it cannot be resolved by collecting more samples, or by increasing the complexity of the model. In the context of reinforcement learning, MDPs with stochastic dynamics — which can capture our slippery surface navigation problem, for example — lead to aleatoric uncertainty.

3.1.2 Insufficient Coverage

The model is typically learned using a training set $\mathcal{D}$ constructed from a finite number of samples $\{(S_i, A_i, R_i, S'_i)\}_{i=1}^{N}$ generated randomly with respect to a combination of the envrionment dynamics, p, and a policy, π (Section 2.11). Depending on the policy, the training set may or may not be representative of the environment dynamics. A training set with insufficient coverage of the state space, in that it does not account for certain regions of the state space, may lead to an inaccurate model. While the resulting model may still be useful for the states that are sufficiently represented in the training set, it may make arbitrary predictions elsewhere. As such, the model should express uncertainty when it makes a prediction for a state for which it has not yet observed sufficient training samples.

In one of our example navigation problems, the training set was not sufficiently representative of the dynamics: it did not include transitions which had obstacles. If a model learned from this training set expresses uncertainty for input states in which the agent faces an obstacle, the agent can avoid using the model in those states.

In the case of parametric models, the uncertainty due to insufficient coverage is referred to as *parameter uncertainty*; there can be a large number of parameter configurations that are consistent with the observed data — there is uncertainty about which parameter configuration should be chosen for making the prediction. Unlike aleatoric uncertainty, parameter uncertainty can be reduced by gathering more data and making the training set more representative.

3.1.3 Limited Capacity

The model may have insufficient capacity to express the environment dynamics. In other words, the model is unable to fit the training set. In case of limited capacity, the model may trade off accuracy on certain regions of the input space to make accurate predictions elsewhere. While the exact nature of this trade-off may depend on the specific model-learning approach, we would

like the model to express high uncertainty on the compromised regions of the state space.

In our factored model example, the model had limited capacity: it was unable to express the dynamics perfectly as it could not consider the interactions between the two coordinates. In this case, the model should express uncertainty when any of the coordinates has a value that sometimes precedes an obstacle.

The uncertainty due to limited capacity is also referred to as *structural uncertainty*. Structural uncertainty due to limited capacity can only be resolved by increasing the model complexity.

3.2 Uncertainty in the Context of the Bias-Variance Trade-off

Ultimately, we want the uncertainty estimates to reflect how the learned model differs from the true dynamics of the environment: the uncertainty estimates should provide a sense of the *approximation error* of the learned model. One can gain a deeper insight into how various types of uncertainty relate to the approximation error by using the lens of the bias-variance trade-off (Geman *et al.* 1992) — one of the central tenets of the field of machine learning.

Consider the problem of regression: given a training set $\mathcal{D} = \{(\mathbf{x_i},\ y_i)_{i=1}^N\}$ from $\mathbb{R}^d \times \mathbb{R}$, find a predictor $f : \mathbb{R}^d \to \mathbb{R}$ which predicts the target y given the input vector $\mathbf{x}$. The samples $(\mathbf{x}, y)$ in the training set $\mathcal{D}$ are drawn from a probability distribution P over $\mathbb{R}^d \times \mathbb{R}$.

The predictor f is typically chosen from some *function class* or *hypothesis space* $\mathcal{F}$. For instance, a neural network's function class $\mathcal{F}$ consists of all functions f that are representable by the network with its allowable parameter configurations. The predictor f can be chosen using the principle of *empirical risk minimization* (Vapnik 1992): find a predictor $f_\mathcal{D} \in \mathcal{F}$ that minimizes the *empirical risk* — or, in other words, the training loss

$$f_\mathcal{D} = \min_{f \in \mathcal{F}} \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\ell(f(\mathbf{x}), y)] \tag{3.1}$$

where ℓ is a loss function (squared loss, for instance).

While the *true risk* of a model $\mathbb{E}_{(\mathbf{x},y)\sim P}[\ell(f(\mathbf{x}), y)]$ — or the approximation error, in other words — is the objective evaluation criteria of the model's efficacy, we cannot compute it since we do not have access to the true distribution P. The hope is that the empirical risk can be a good estimate of the true risk; that is, a predictor that minimizes the loss on the training samples should *generalize* to the *test samples* that are drawn independently from P. Generalization to unseen test samples is in fact the goal of supervised machine learning.

For squared loss, the optimal predictor f^* that minimizes the true risk is the mean predictor (Geman *et al.* 1992): the deterministic function of $\mathbf{x}$ which predicts the mean value of y conditioned on $\mathbf{x}$. As such, the risk of a predictor $f_{\mathcal{D}}$ depends on the variance of y given $\mathbf{x}$ (irreducible stochasticity), as well as how well the predictor approximates f^*. Geman *et al.* (1992) showed that the expected squared error, for a given $\mathbf{x}$, can be decomposed as follows:

$$\begin{aligned}\mathbb{E}_{y\sim P(y|\mathbf{x}),\mathcal{D}}[(f_{\mathcal{D}}(\mathbf{x}) - y)^2] &= \mathbb{E}_{y\sim P(y|\mathbf{x})}[(y - \mathbb{E}[y|\mathbf{x}])^2] && \text{(Irreducible Error)}\\ &+ \mathbb{E}_{\mathcal{D}}[(f_{\mathcal{D}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[f_{\mathcal{D}}(\mathbf{x})])^2] && \text{(Variance)}\\ &+ (\mathbb{E}_{\mathcal{D}}[f_{\mathcal{D}}(\mathbf{x})] - \mathbb{E}[y|\mathbf{x}])^2 && \text{(Bias}^2\text{)}\end{aligned}$$

This decomposition is suggestive of the prevalent regularization techniques as it emphasizes the need to balance bias and variance. If the function class $\mathcal{F}$ is too simple, a predictor $f_{\mathcal{D}}$ will have bias: $f_{\mathcal{D}}$ fails to fit the training samples (large empirical risk), and predicts poorly on the test samples (large true risk). On the other hand, a complex function class may lead to high variance: $f_{\mathcal{D}}$ is susceptible to the nonrepresentativeness of the training samples, and while $f_{\mathcal{D}}$ may successfully capture the training samples (small empirical risk), it generalizes poorly to the test samples (large true risk).

The bias-variance decomposition of squared-error is reminiscent of the types of uncertainty discussed in the previous section. A predictor contributes to the approximation error in three ways: with the limitation of its function class (bias), with its susceptibility to the spurious patterns manifested by

the insufficient coverage of the training set (variance), and with the inherent stochasticity of the data generating process (irreducible error).

In the context of selective planning, we need uncertainty mechanisms that identify the regions of the state space where the learned model is not accurate, enabling the planning algorithm to be robust to the inaccuracies of the learned model. The difficulty is that it is difficult to express uncertainty for all three sources of error. Bayesian approaches, though, do allow us to express uncertainty in case of insufficient coverage, as we describe in the next section.

3.3 Estimating Parameter Uncertainty using Bayesian Inference

Bayesian inference offers an elegant approach for estimating parameter uncertainty. The key idea is to maintain a (posterior) distribution over plausible hypotheses (parameter configurations). When there are several plausible hypotheses that are consistent with the observed data, the degree to which they disagree in their predictions for an input not covered by the training set reflects the level of uncertainty: there is uncertainty about which hypothesis should be chosen for making the prediction.

3.3.1 Maintaining a Posterior

Consider a hypothesis space $\mathcal{F}$ with N hypotheses: $f_1, f_2, ..., f_N$. Our *prior* belief in the plausibility of hypothesis f_i is $p(f_i)$. The *likelihood* of observing data $\mathcal{D}$ if hypothesis f_i is true is $p(\mathcal{D}|f_i)$. Once we observe data $\mathcal{D}$, we can update our beliefs in the plausibility of alternative hypotheses by computing the *posterior* with *Bayes' rule* (Bayes 1763):

$$p(f_i|\mathcal{D}) = \frac{p(f_i)p(\mathcal{D}|f_i)}{p(\mathcal{D})} \tag{3.2}$$

$p(\mathcal{D})$ is the marginal distribution of the data: $p(\mathcal{D}) = \sum_{i=1}^{N} p(\mathcal{D}|f_i)p(f_i)$. It is also referred to as *model evidence* or *marginal likelihood*. We can write Bayes' rule in words as:

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{model evidence}} \tag{3.3}$$

Under the Bayesian framework, *learning* refers to Bayesian inference: the process of transforming the prior over the alternative hypotheses in $\mathcal{F}$ into posterior by incorporating the observed data. The posterior distribution is then used for making predictions and estimating uncertainty in the hypotheses. Let us now make these ideas concrete by looking at the problem of Bayesian regression.

3.3.2 Bayesian Regression: An Example

Assume we are given a data set $\mathcal{D} = \{(\mathbf{x_i},\ y_i)_{i=1}^N\}$. Our goal is to determine the extent to which a function $f_{\mathbf{w}} : \mathbb{R}^d \rightarrow \mathbb{R}$, parametrized by the weight vector $\mathbf{w}$, is likely to have generated the targets y given the input vectors $\mathbf{x}$. Let's assume that our hypothesis space consists of the linear functions, $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$, corresponding to all plausible configurations of the weight vector, $\mathbf{w} \in \mathbb{R}^d$.

We begin by assuming a likelihood distribution $p(y|\mathbf{x}, \mathbf{w})$. Given a weight vector $\mathbf{w}$, the likelihood determines how the target Y is generated for a particular input $\mathbf{x}$. Consider, for example, the Gaussian likelihood: $p(y|\mathbf{x}, \mathbf{w}) = \mathcal{N}(\mathbf{w}^\top \mathbf{x}, \sigma^2)$. This likelihood assumes that the target variable Y is Gaussian distributed, with mean $\mathbf{w}^\top \mathbf{x}$ and input-independent variance σ^2. Once the data $\mathcal{D}$ has been observed, the posterior can be written using the Bayes' rule (Bayes 1763) as follows:

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{w})p(\mathbf{y}|\mathbf{X}, \mathbf{w})}{p(\mathbf{y}|\mathbf{X})} \tag{3.4}$$

where $\mathbf{y} = [y_1\ y_2\ ...\ y_N]^T$ is an $n \times 1$ vector, $\mathbf{X} = [\mathbf{x_1}\ \mathbf{x_2}\ ...\ \mathbf{x_N}]^T$ is an $n \times d$ matrix, and n is the number of training samples in $\mathcal{D}$.

If the prior, $p(\mathbf{w})$, is a zero-mean Gaussian with covariance Σ_d, then the corresponding posterior (Bishop 2006) is:

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \mathcal{N}(\frac{1}{\sigma^2} A^{-1} \mathbf{X}^\top \mathbf{y}, A^{-1}) \tag{3.5}$$

where

$$A = \frac{1}{\sigma^2} \mathbf{X}^\top \mathbf{X} + \Sigma_d^{-1} \tag{3.6}$$

The application of Bayes rule increases our belief in the weight configurations under which the likelihood of observed data is large, while reducing our belief in the weight configurations under which the likelihood of observed data is small. The updated beliefs can be used to predict the target for a test input $\mathbf{x}$ as follows:

$$p(y|\mathbf{x}, \mathbf{X}, \mathbf{y}) = \int p(y|\mathbf{x}, \mathbf{w}) p(\mathbf{w}|\mathbf{X}, \mathbf{y}) d\mathbf{w} \tag{3.7}$$

This distribution is referred to as the *predictive distribution*. The expected value of Y under the predictive distribution, $\mathbb{E}[Y]$, can be used as a prediction of the target variable Y for a given $\mathbf{x}$, whereas the variance of Y, $\text{Var}[Y]$, can be used as an estimate of parameter uncertainty. In our example. the predictive distribution can be written as:

$$p(y|\mathbf{x}, \mathbf{X}, \mathbf{y}) = \mathcal{N}(\frac{1}{\sigma^2}\mathbf{x}^\top A^{-1}\mathbf{X}^\top \mathbf{y}, \mathbf{x}^\top A^{-1}\mathbf{x}) \tag{3.8}$$

where $\mathbb{E}[Y] = \frac{1}{\sigma^2}\mathbf{x}^\top A^{-1}\mathbf{X}^\top \mathbf{y}$; $\text{Var}[Y] = \mathbf{x}^\top A^{-1}\mathbf{x}$ — parameter uncertainty. Parameter uncertainty can be used to identify situations in which we have insufficient coverage. In the next section, we discuss methods which extend Bayesian regression, beyond the simple case of linear models, to neural networks.

3.4 Parameter Uncertainty Methods for Neural Nets

We now turn towards specific techniques for expressing parameter uncertainty in the context of neural networks. There is a rich history of research on neural-network uncertainty under the umbrella of *Bayesian neural neworks* (Barber and Bishop 1998; Bishop *et al.* 1998; Hinton and Van Camp 1993; MacKay 1992; Neal 1995). Under the Bayesian framework, predictions require integration over the posterior distribution; in the case of neural networks, the integral is analytically intractable. A significant body of research on Bayesian neural networks is concerned with the approximation of these intergrals — by modeling the posterior distribution using a Gaussian (MacKay 1992), or by generating samples from the posterior distribution with Markov Chain Monte

Carlo (MCMC) simulations, or by formulating the inference problem as an optimization problem: variational inference (Barber and Bishop 1998; Graves 2011; Hinton and Van Camp 1993). Some of the more recent approaches approximate the posterior distribution using dropout sampling (Gal and Ghahramani 2016; Gal *et al.* 2017; Li and Gal 2017).

There is an alternative line of research inspired by the statistical bootstrap (Efron 1982). The corresponding methods train an *ensemble* of neural networks, possibly on independent *bootstrap samples* of the original training samples, and use the predictive distribution of the ensemble to estimate parameter uncertainty (Lakshminarayanan *et al.* 2017; Osband *et al.* 2016; Osband *et al.* 2018; Pearce *et al.* 2018). The essential idea is that the individual members in the ensemble would make dissimilar predictions for the regions of the input space that are not sufficiently represented in the training samples. While ensemble-based methods can be interpreted as Bayesian approximations only under restricted settings (Fushiki 2005a; Fushiki *et al.* 2005b; Osband *et al.* 2018), they do share do share one of the goals of the Bayesian methods — express the uncertainty due to insufficient coverage. In the following, we review a subset of the methods that can be used to estimate parameter uncertainty.

3.4.1 Monte-Carlo Dropout

Gal and Ghahramani (2016) proposed to use *dropout* (Srivastava *et al.* 2014) for obtaining uncertainty estimates from neural networks. Dropout is a regularization method which prevents overfitting by randomly dropping units during training — with probability p, dropout technique independently sets a hidden unit activation to zero. Monte-Carlo dropout estimates uncertainty by computing the variance of the predictions obtained by M stochastic forward passes through the network. This technique can be interpreted through the lens of Bayesian inference; that is, the dropout distribution approximates the Bayesian posterior (Gal 2016).

3.4.2 Ensemble of Neural Networks

In this approach, K randomly initialized neural networks are trained independently — the variance in the predictions of individual networks is then used to estimate parameter uncertainty (Lakshminarayanan *et al.* 2017; Osband *et al.* 2016; Pearce *et al.* 2018). Intuitively, the individual networks in an ensemble should make similar predictions in the regions of the input space where sufficient samples have been observed while making dissimilar predictions elsewhere.

3.4.3 Randomized Prior Functions

Randomized Prior Functions (RPF) (Osband *et al.* 2018) can be viewed as an extension to the ensemble method: each network in the ensemble is coupled with a random but fixed *prior* function — a randomly initialized neural network whose weights remain unchanged during training. Prediction of an individual ensemble member is now the sum of its trainable network and its prior function. Given sufficient samples, the ensemble members will agree in their predictions; on the other hand, for the regions of the input space where sufficient samples have not been observed, the generalization of the individual networks and the priors will lead to disagreements. For Gaussian linear models, this approach is equivalent to exact Bayesian inference (Osband *et al.* 2018).

3.4.4 Randomized Prior Functions with Bootstrapping

Randomized prior functions can be combined with bootstrapping. One common approach to bootstrapping involves random sampling from a given dataset D to construct several datasets $\hat{D}$, and using the resulting datasets to obtain as many estimators; the resulting ensemble of estimators can be used to estimate uncertainty. While boostrapping has been used with both the ensemble of neural nets (Osband *et al.* 2016) as well as with randomized prior functions

(Osband *et al.* 2018), we will only focus on the latter as it has been noted to provide better uncertainty estimates (Osband *et al.* 2018).

3.5 Parameter Uncertainty and a Limited Hypothesis Space

Parameter uncertainty, as described above, refers to the uncertainty in the hypotheses (parameter configurations); that is, there may be multiple hypotheses consistent with the observed data. As more data is observed, this uncertainty diminishes, ultimately going to zero in the limit of data: the posterior distribution concentrates at the true hypothesis in the hypothesis space *if* it contains the true hypothesis (De Finetti 1937). If, however, the hypothesis space is limited in that it does not contain the true hypothesis, one can expect the posterior distribution to concentrate at the hypothesis which is *closest*, in some sense, to the true hypothesis. While this closest hypothesis may not be fully consistent with the observed data, there would be no uncertainty in the plausible hypotheses in the hypothesis space — parameter uncertainty would resolve.

Nonetheless, parameter uncertainty can be an important component of a selective planning mechanism as it can express an important source of uncertainty: insufficient coverage. However, we need a separate uncertainty mechanism for structural uncertainty and aleatoric uncertainty. In the next chapter, we describe a mechanism which can effectively account for both types of uncertainties.

Chapter 4

Parameter Uncertainty is Not Enough

In the previous chapter, we outlined distinct types of uncertainty — namely, aleatoric uncertainty, parameter uncertainty, and structural uncertainty. We reviewed parameter uncertainty methods for neural networks, and argued that parameter uncertainty in and of itself may not be sufficient to signal the errors due to limited capacity. In this chapter, we present an approach for aleatoric uncertainty that can also be used to express structural uncertainty. We then use a simple regression problem to demonstrate that while parameter uncertainty methods can be predictive of insufficient coverage, the presented approach can be predictive not only of the inherent stochasticity but also of the inaccuracies due to limited capacity — suggesting that an effective selective planning solution needs to use a combination of uncertainty methods capturing different sources of uncertainty.

4.1 Estimating Structural Uncertainty using the Learned Variance of a Gaussian Distribution

Neural networks are typically trained to output a point estimate as a function of the input. When trained with mean-squared error, the probabilistic interpretation is that the point estimate corresponds to the mean of a Gaussian distribution with fixed input-independent variance σ^2: $p(y|\mathbf{x}) = \mathcal{N}(f_\mu(\mathbf{x}), \sigma^2)$;

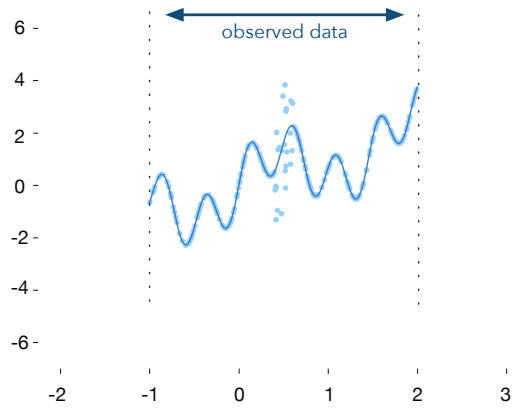

Figure 4.1: The target function $y = x + \sin(4x) + \sin(13x) + \epsilon$ — where $\epsilon \sim \mathcal{N}(0, 1)$ when $x \in (0.4, 0.6)$ and otherwise $\epsilon = 0$ — shown for the training interval (-1.0, 2.0). The points in blue represent 300 training samples drawn uniform randomly from the training interval.

maximizing the likelihood in this case leads to least-squares regression.

An alternative is to assume that the variance is also input-dependent: $p(y|\mathbf{x}) = \mathcal{N}(f_\mu(\mathbf{x}), f_\sigma(\mathbf{x})^2)$, where $f_\mu(\mathbf{x})$ is the predicted mean and $f_\sigma(\mathbf{x})^2$ is the predicted variance. Under this assumption, maximizing the likelihood leads to the following loss function (Nix and Weigend 1994):

$$L_i(\theta) = \frac{(y_i - f_\mu(\mathbf{x}_i))^2}{2f_\sigma(\mathbf{x}_i)^2} + \frac{1}{2}\log f_\sigma(\mathbf{x}_i)^2 \tag{4.1}$$

The learned variance $f_\sigma(\mathbf{x})$ can be predictive of stochasticity: the network can incur less penalty in high-noise regions of the input space by predicting high variance. Interestingly, the learned variance can be predictive of the errors in the context of limited capacity: a network can maintain a small loss by allowing the variance to be larger in regions where it lacks the capacity to make accurate predictions.

4.2 An Example Regression Problem

We now use a simple regression problem to investigate the utility of the learned variance to signal the errors due to limited capacity. The example also highlights the limitation of parameter uncertainty when the model capacity is limited. In particular, we qualitatively evaluate methods for parameter uncertainty: Monte-Carlo dropout — a Bayesian method (Section 3.4.1); ensemble-based methods — an ensemble of randomly initialized neural networks (Section 3.4.2), randomized prior functions (3.4.3), and randomized prior functions with bootstrapping (Section 3.4.4).

We construct a dataset of 5,000 training examples using the function $y = x + \sin(\alpha x) + \sin(\beta x) + \epsilon$, where $\alpha = 4$, $\beta = 13$, and $\epsilon \sim \mathcal{N}(0, 1)$ when $x \in (0.4, 0.6)$ and otherwise $\epsilon = 0$ (Figure 4.1). The inputs x are drawn from a uniform distribution over the interval $(-1.0, 2.0)$. We would like to find a predictor $f : \mathbb{R} \to \mathbb{R}$, from some function class $\mathcal{F}$, which predicts the target y given the input x. The function class $\mathcal{F}$ is implied by the neural network architecture and the training procedure — for instance, $\mathcal{F}$ can consist of all functions reachable by the application of Adam optimizer (Kingma and Ba 2015) for 100 epochs, over a fully connected neural network with two hidden layers of 50 ReLU units each, initialized using Glorot initialization (Glorot and Bengio 2010).

4.3 Experiment Setup

We vary the effective capacity of the model by reducing the number of layers and the number of hidden units. In particular, we use neural networks with three degrees of complexity: 3 hidden layers with 64 hidden units each (referred to as *large*), a single hidden layer with 2048 hidden units (*medium*), and a single hidden layer with 64 hidden units (*small*). We use Adam optimizer for training the model. We set the batch size to 16. We consider the learning rates 0.01, 0.001, and 0.0001. We use ReLU activations for non-linearities,

and initialize the networks with Glorot initialization.

In MC-Dropout, we set the dropout probability $p = 0.1$. For obtaining the variance , we perform 10 stochastic forward passes.

For the ensemble-based methods, we use ensembles of 10 neural networks. All networks in the ensemble are trained using the squared-error loss.

For randomized prior functions, we simply modify each member of the ensemble to also have an additional network representing the prior function. In case of randomized prior functions with bootstrapping, we train each member of the ensemble on a bootstrapped dataset generated from the original dataset by randomly sampling with replacement.

For the aleatoric-uncertainty method, which learns the mean and the variance under the Gaussian assumption, we train separate neural networks for the mean and the variance, and optimize them jointly using the loss from Equation 4.1. While we change the capacity of the mean network across the three regimes (large, medium, and small), we restrict the variance network to be small — a single hidden layer with 64 hidden units — in all three regimes.

For each uncertainty method, every configuration is evaluated using 5 independent runs initialized with a different random seed. While the results remain consistent across the independent runs, we present results for a single run chosen randomly. The results are shown in Figure 5.2 (large capacity), Figure 5.3 (medium capacity), and Figure 3.4 (small capacity) for a single configuration of learning rate. We found the results to be consistent across learning rate configurations (see the appendix).

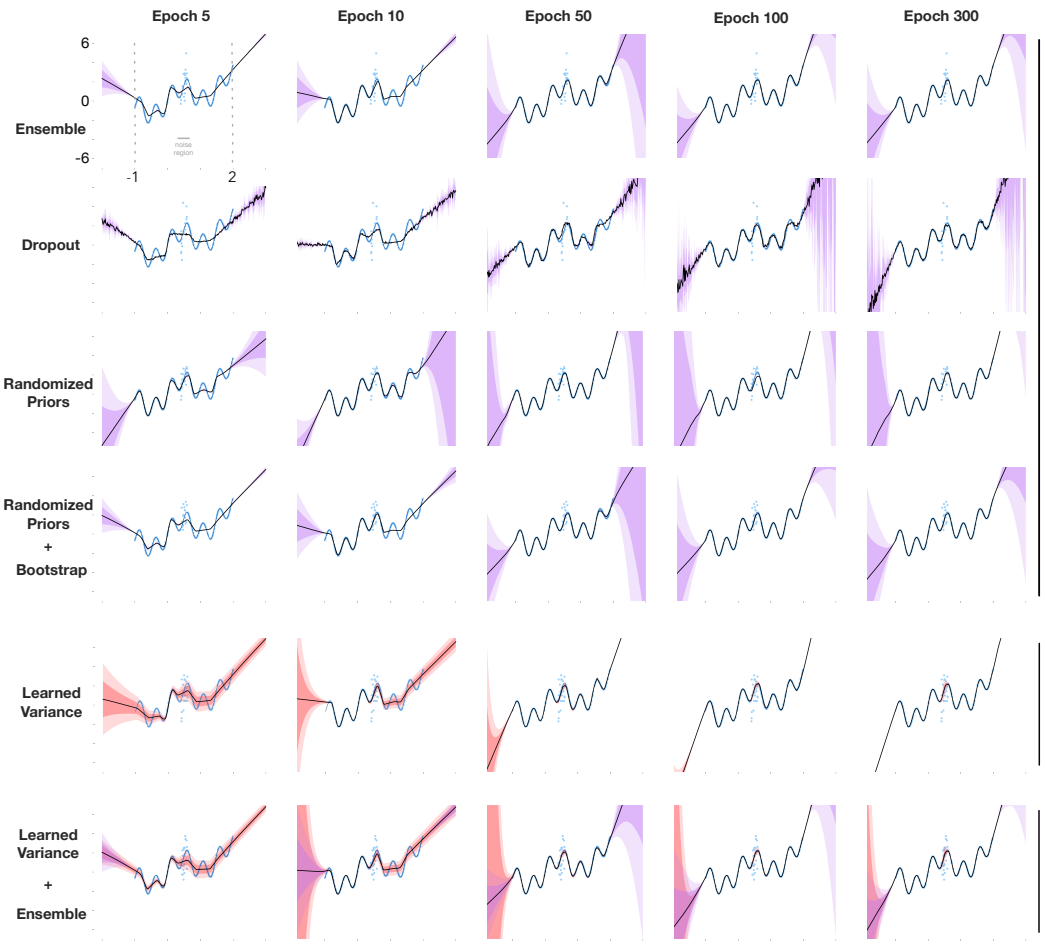

Figure 4.2: Large Capacity Results. The network architecture consists of 3 hidden layers with 64 hidden units each. The ground truth function is in blue in all plots. Each row represents the mean predictions and uncertainty estimates of a particular modeling approach over the course of training. Uncertainty estimates are represented by shaded intervals; parameter uncertainty is in purple; Learned variance is in red; darker purple/red intervals show mean $\pm$ 1 standard-deviation and lighter intervals show mean $\pm$ 2 standard-deviation. Learning rate is 0.001 for all methods; the results for other configurations of the learning rate, which are consistent with the results in this figure, can be found in the appendix

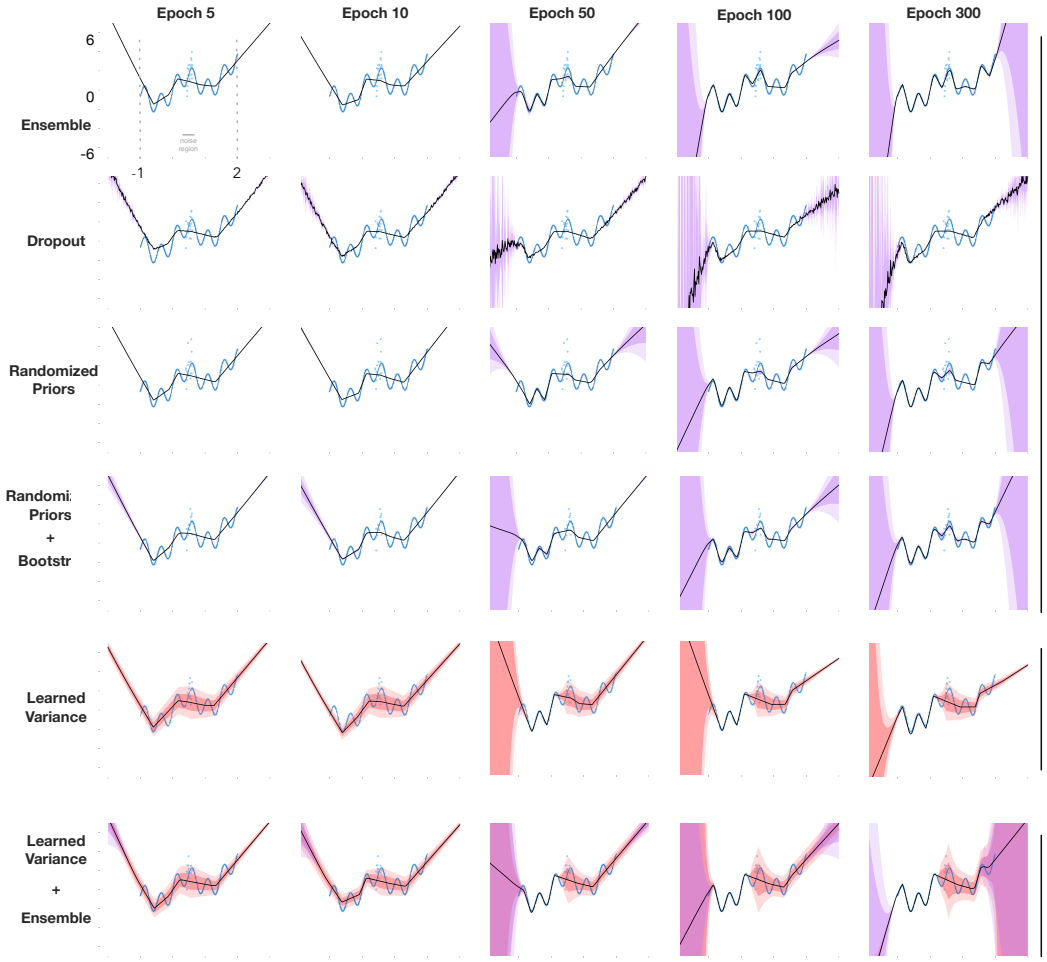

Figure 4.3: Medium capacity results. The network architecture consists of a single hidden layers with 2048 hidden units. The ground truth function is in blue in all plots. Each row represents the mean predictions and uncertainty estimates of a particular modeling approach over the course of training. Uncertainty estimates are represented by shaded intervals; parameter uncertainty is in purple; Learned variance is in red; darker purple/red intervals show mean $\pm$ 1 standard-deviation and lighter intervals show mean $\pm$ 2 standard-deviation. Learning rate is 0.01 for all methods; the results for other configurations of the learning rate, which are consistent with the results in this figure, can be found in the appendix

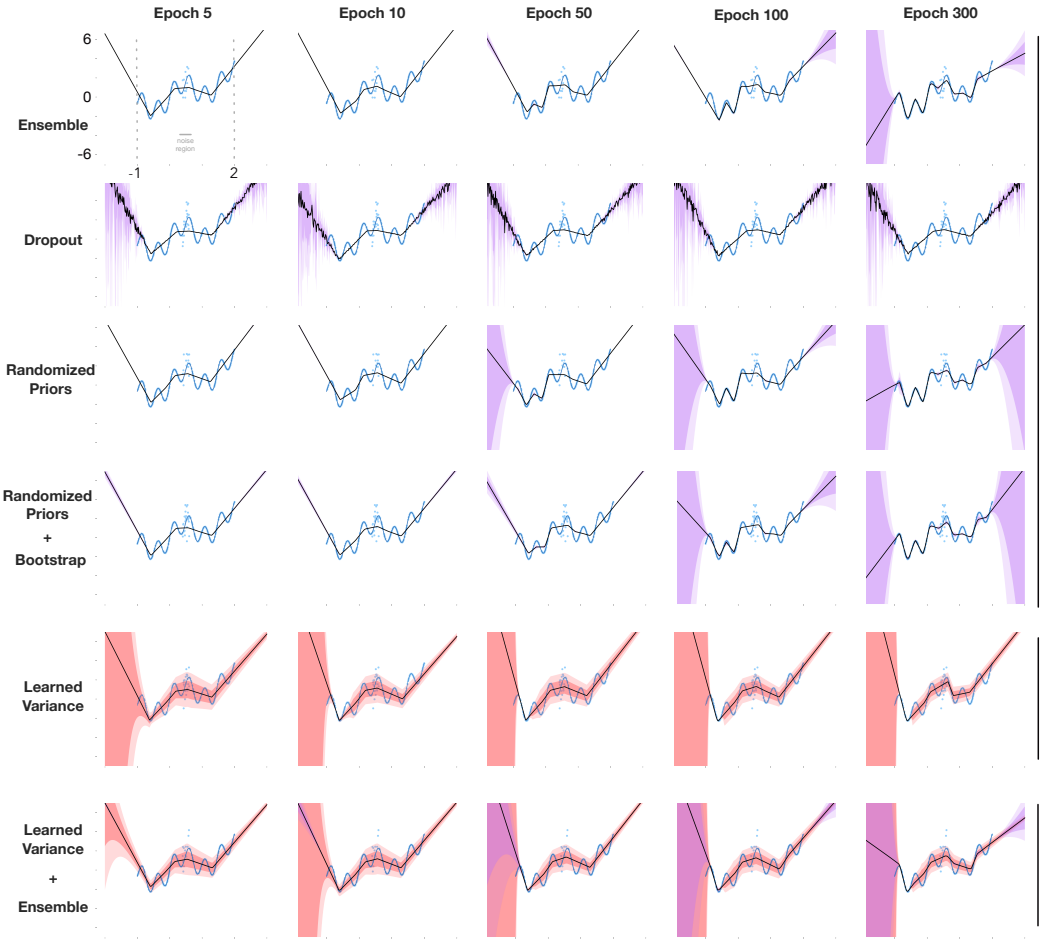

Figure 4.4: Small capacity results (learning 0.001). The network architecture consists of a single hidden layer with 64 hidden units. The ground truth function is in blue in all plots. Each row represents the mean predictions and uncertainty estimates of a particular modeling approach over the course of training. Uncertainty estimates are represented by shaded intervals; parameter uncertainty is in purple; Learned variance is in red; darker purple/red intervals show mean $\pm$ 1 standard-deviation and lighter intervals show mean $\pm$ 2 standard-deviation. Learning rate is 0.01 for all methods; the results for other configurations of the learning rate, which are consistent with the results in this figure, can be found in the appendix

4.4 Results and Conclusion

With a sufficiently powerful network (Figure 5.2), the ensemble learns to accurately predict the mean, and the variance of the ensemble (purple) appropriately assesses the parameter uncertainty — the ensemble variance is large outside the observed training distribution. However, the variance of the ensemble fails to express uncertainty in the input region with stochasticity; that is, the individual members of the ensemble make similar predictions in the noisy input interval. We observe the same effect for the other ensemble-based methods, as well as Monte-Carlo dropout. In contrast, while the learned variance (light red) fails to reflect parameter uncertainty, it still expresses aleatoric uncertainty in the noisy interval.

As the capacity is reduced (Figure 5.2 and Figure 5.3), all methods fail to fit the mean function accurately over the entire observed input space. In case of ensemble-based methods, the variance of the ensemble within the training distribution remains small. A similar effect is observed for Monte-Carlo dropout. On the other hand, the learned variance reliably reflects the errors within the input distribution; that is, it expresses structural uncertainty. In this case, while the capacity is insufficient to learn the target function, it is enough to express the inability to do so.

These results support the idea that relying parameter uncertainty (MC-dropout, ensemble of randomly initialized networks, randomized prior functions with and without bootstrapping) is an insufficient metric for selective planning. Instead, they suggest that a combination of one of these methods, which tends to accurately reflect parameter uncertainty, with learned variance, which tends to accurately reflect aleatoric and structural uncertainty, yields a much more robust error detection mechanism than either do individually. While the former method would represent uncertainty in case of insufficient coverage, the latter would express uncertainty in case of stochasticity and limited capacity. In order to demonstrate this, we use the loss function in Equation 4.1 to train an ensemble of neural networks which predict both mean and variance, and query the model for both the learned variance and ensemble

variance. The results are shown in the last rows of figures 5.2, 5.3 and 5.4 — they suggest that a combination such as this one can provide a comprehensive estimate of uncertainty.

In the next chapter, we study the utility of the learned variance in the context model-based reinforcement learning; in particular, we investigate if the learned variance can be used to plan selectively with a low-capacity model that otherwise leads to planning failures.

Chapter 5

Combating Planning Failures under Limited Model Capacity

In the previous chapter, we discussed how the diversity of networks in an ensemble — by virtue of random initialization, for example — can reflect parameter uncertainty in situations where the model has not seen sufficient samples. On the other hand, the learned variance (Section 4.1) can express uncertainty not only in the case of stochasticity but also in the case of limited capacity. In this chapter, we explore the utility of the learned variance in combating planning failures that are caused by inadequate model capacity.

We choose Model-based Value Expansion (MVE) (Feinberg *et al.* 2018) (Section 2.10), a planning algorithm, as the subject of study. To isolate the effect of model capacity, we progressively reduce the size of the neural network used for model learning, eventually leading to planning failures. To investigate if the learned variance can help recover the performance of low capacity models, we instantiate Selective MVE — an MVE extension that uses the learned variance to regulate model usage.

5.1 Experiment Design

5.1.1 Environment

In this chapter, experiments are done using Acrobot (Sutton 1996), a classic environment loosely based on a gymnast swinging on a highbar. It consists of two links and two joints, as depicted in Figure 5.1. The goal is to swing the

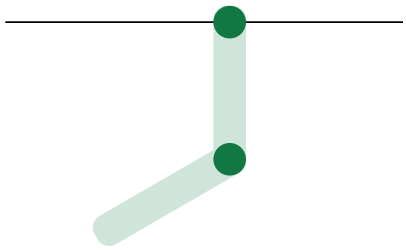

Figure 5.1: A pictorial depiction of Acrobot

endpoint of the lower link above the bar up to a height equal to the length of one of the links. The state consists of sines and cosines of two rotational joint angles, and the joint angular velocities, leading to a 6-dimensional state representation. The joint between the two links can be actuated by the agent; the agent's action space is discrete and consists of three actions: positive torque, negative torque, and no torque. The reward is -1 on all time steps. In our experiments, we use OpenAI gym's implementation of Acrobot (Brockman *et al.* 2016).

5.1.2 Baseline Algorithms

Deep Q-Networks (DQN)

We use Deep Q-Networks (DQN) (Mnih *et al.* 2015) as the baseline model-free algorithm; we estimate the action-value function using a fully connected neural network with ReLU activations. We repeat all experiments in this chapter for four different fully connected neural network architectures: 1 hidden layer with 64 hidden units, 1 hidden layer with 128 hidden units, 2 hidden layers with 64 hidden units each, and 2 hidden layers with 128 hidden units each. For each network architecture, we determine the best setting for the step-size, the batch size, and the replay memory size by sweeping over possible parameter configurations (Section 5.1.3 describes the parameter sweep strategy). For DQN baseline, the range of values for the parameter sweep, and the configuration of the rest of the hyperparameters are presented in Table 5.1.

Table 5.1: DQN hyperparameters used in the experiments. The step-size, the batch size, and the replay memory size were determined by sweeping over the range specified in the respective rows.

Hyperparameter	**Values**
Optimizer	RMSProp
Step-size (α)	0.03, 0.01, 0.003, 0.001, 0.0003, 0.0001
Batch size	16, 32, 64
Replay memory size	10000, 20000, 50000
Target network update frequency	256 environment steps
Training frequency	1 update for every environment step
Exploration rate (ϵ)	0.1
Discount factor (γ)	1.0

Model-Based Value Expansion (MVE)

We implement MVE by extending the DQN algorithm with the model-based policy evaluation technique exemplified in Figure 2.1. For simplicity, we instantiate MVE with a deterministic model learned using the squared error loss, which is equivalent to learning an expectation model of the next state features. We assume the reward signal to be known; that is, we only learn the state dynamics. We present MVE's pseudocode in Appendix B. As alluded to earlier, we study the effect of model capacity by progressively reducing the size of the neural network used for model learning. In particular, we use four variants of a single hidden layer neural network, which vary only in the number of hidden units: 128 hidden units, 64 hidden units, 16 hidden units, and 4 hidden units. In all cases, we learn the model online using the experience gathered in the replay buffer: at every time-step, alongside the MVE value function update, we separately sample a batch of transitions to update the model.

Once we have identified the best hyper-parameter configuration for the DQN baselines which vary in their value function architecture, we keep the same hyper-parameter configuration for their MVE extensions, and only sweep over the model learning rate for each of the four model architectures. The range of values for the parameter sweep, and the configuration of the rest of the hyperparameters are presented in Table 5.2.

Table 5.2: MVE specific hyperparameters. For each simulated trajectory length (rollout length), the model learning step-size (β) was determined by sweeping over the range specified in the respective row.

Hyperparameter	**Values**
Optimizer	Adam
Model learning step-size (β)	0.1, 0.01, 0.001, 0.0001
Batch size	16
Loss function	Squared error
Model learning frequency	1 update for every environment step
Simulated trajectory length	2, 3, 4

5.1.3 Parameter Sweep Strategy

All experiments in this chapter are run for 100,000 environment interactions. The resolution of the reported results is 2,000 steps: we log the average returns of the last 20 episodes every 2,000 steps. To determine the best-performing hyper-parameter setting, each configuration is evaluated using 10 independent runs initialized with a different random seed, leading to as many learning curves. The learning curves are averaged, and the second half of the averaged learning curve is summed up to obtain a single number representing the performance of the particular configuration. If the best-performing parameter setting falls on the boundary of the range of tested values for any hyper-parameter, we widen the range until this is not true. The best-performing configuration is evaluated using 30 additional runs, each initialized with a different random seed, and the average learning curve, along with its standard error, is reported.

5.2 Evaluation of MVE under Capacity Constraints

Figure 5.2 depicts the effect of reducing model-capacity on the performance of MVE for which the action-value function is approximated using a single hidden layer neural network with 128 units. (The results for other value function network architectures are presented at the end of this chapter.) While MVE instances with 64 and 128 hidden units offer modest sample efficiency benefits for rollouts consisting of 2 model simulations (Figure 5.2 b), MVE instances

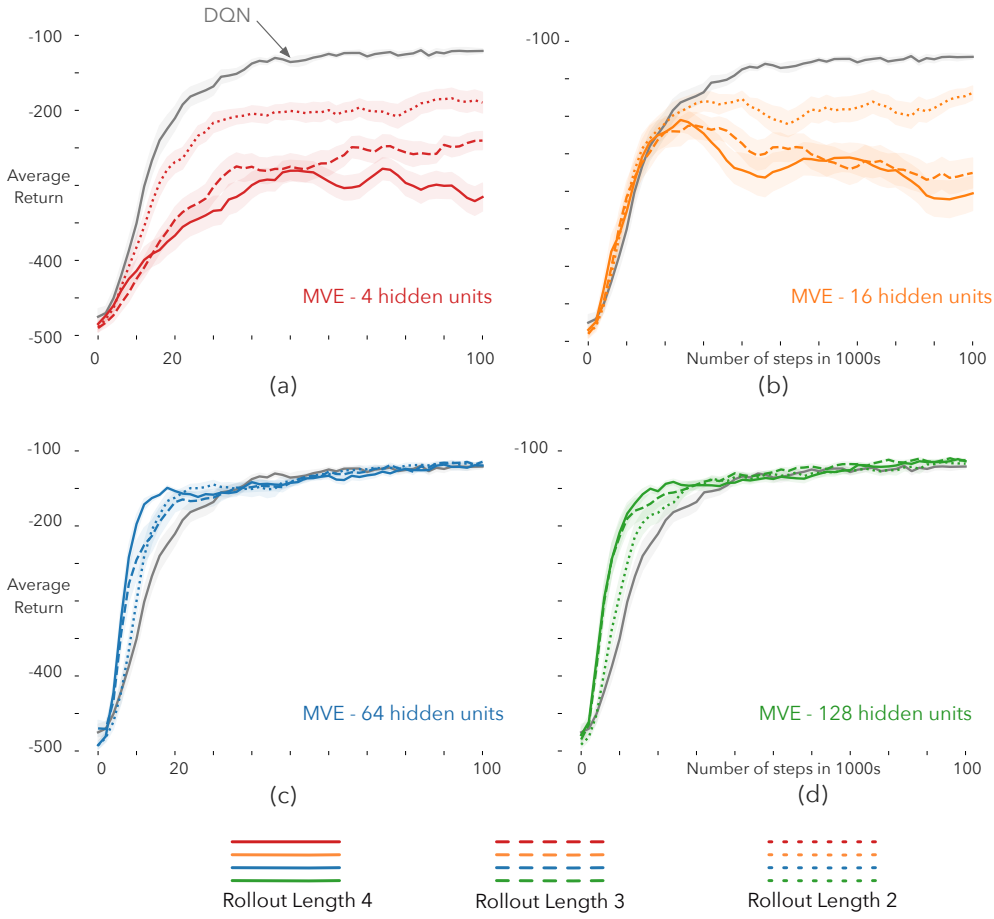

Figure 5.2: The effect of model capacity on MVE's performance. The learning curves are averaged over 30 runs; the shaded regions show the standard error. As we increase the rollout length, the sample-efficiency of MVE improves in case of larger models of 64 and 128 hidden units, whereas planning failures are observed for smaller models of 4 and 16 hidden units.

with 4 and 16 hidden units achieve subpar asymptotic performance. As we increase the rollout length (Figure 5.2 c-d), the sample efficiency benefits of MVE become more pronounced for larger models. On the other hand, the performance of MVE with smaller models deteriorates as we increase the rollout length. In the next section, we investigate if selective planning can salvage the performance of the smaller models.

5.3 Selective Model-Based Value Expansion

The learned variance, as discussed in the previous chapter, can be predictive of the accuracy of the model. In this section, we propose a technique which uses the learned variance to enable selective model-based value expansion: trust a multi-step target only when the simulated trajectory is expected to be accurate. Given a maximum rollout length H, consider the weighted-average of h-step targets:

$$U_{avg}(s_0, a_0) = \sum_{h=1}^{H} w_h(s_0, a_0) U_h(s_0, a_0)$$

We would like the weight on an h-step target to be inversely proportional to the cumulative uncertainty in the h-step simulated trajectory $\sigma_{1:h}(s_0, a_0)$ originating from (s_0, a_0):

$$w_h(s_0, a_0) \propto \frac{1}{\sigma_{1:h}(s_0, a_0)}$$

The cumulative uncertainty $\sigma_{1:h}$ on an h-step simulated trajectory is the sum of the predicted uncertainty $\sigma(s_i, a_i)$ of the state-action pairs that constitute the simulated trajectory:

$$\sigma_{1:h}(s_0, a_0) = \sum_{k=0}^{h-1} \sigma(s_k, a_k)$$

Given the cumulative uncertainty of the targets, we can determine the weight of an individual target by computing the softmax:

$$w_h(s_0, a_0) = \frac{\exp(-\sigma_{1:h}(s_0, a_0)/\tau)}{\sum_{i=1}^{H} \exp(-\sigma_{1:i}(s_0, a_0)/\tau)} \tag{5.1}$$

where τ is a hyper-parameter which regulates the weight's sensitivity to the predicted uncertainty: the sensitivity to the uncertainty in the target increases as we reduce the value of τ.

In order to achieve the desired weighting, the model has to predict $\sigma(s, a)$ for a given (s, a). This can be achieved by extending the assumptions described in Section 4.1 — for regressing over a scalar — to account for d-dimensional predictions. Let $\mathbf{s}$ and $\mathbf{a}$ be the feature vector representation

of state s and action a. We assume that next-state features are normally distributed with mean $\mu_\theta(\mathbf{s}, \mathbf{a})$ and diagonal covariances $\Sigma_\theta(\mathbf{s}, \mathbf{a})$; that is, $p(\mathbf{s}'|\mathbf{s}, \mathbf{a}) = \mathcal{N}(\mu_\theta(\mathbf{s}, \mathbf{a}), \Sigma_\theta(\mathbf{s}, \mathbf{a}))$. Under this assumption, given a transition $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$, maximizing the likelihood leads to the following loss function:

$$L_{(\mathbf{s},\mathbf{a},\mathbf{s}')}(\theta) = [\mu_\theta(\mathbf{s}, \mathbf{a}) - \mathbf{s}']^T \Sigma_\theta^{-1}(\mathbf{s}, \mathbf{a})[\mu_\theta(\mathbf{s}, \mathbf{a}) - \mathbf{s}'] + \log\det \Sigma_\theta(\mathbf{s}, \mathbf{a}) \quad (5.2)$$

The intuition discussed in Section 4.1 also translates to the d-dimensional case: to reduce the loss, the predicted covariance will be large in regions where the model lacks the capacity to make accurate predictions.

Table 5.3: Hyperparameters for Selective-MVE. Model learning step-size (β) was determined by sweeping over the range specified in the respective row.

Hyperparameter	**Values**
Optimizer	Adam
Model learning step-size (β)	0.1, 0.01, 0.001, 0.0001
Batch size	16
Loss function	Equation 5.2
Model learning frequency	1 update for every environment step
Simulated trajectory length	4
Softmax temperature (τ)	0.1

Selective MVE Implementation: We modify the base neural networks to output the diagonal covariances alongside the mean next-state feature vector. We enforce the positivity constraint on the covariances by passing the corresponding output through the *softplus* function $\log(1 + \exp(.))$; and, for numerical stability, we also add a small constant value of 10^{-6} to the predicted covariances (Lakshminarayanan *et al.* 2017). The model is optimized using the loss function in equation 5.2. For input (s_i, a_i), $\sigma(s_i, a_i)$ is simply the trace of $\Sigma_\theta(\mathbf{s}_i, \mathbf{a}_i)$. The range of values for the parameter sweep, and the configuration of the rest of the hyperparameters are presented in Table 5.3.

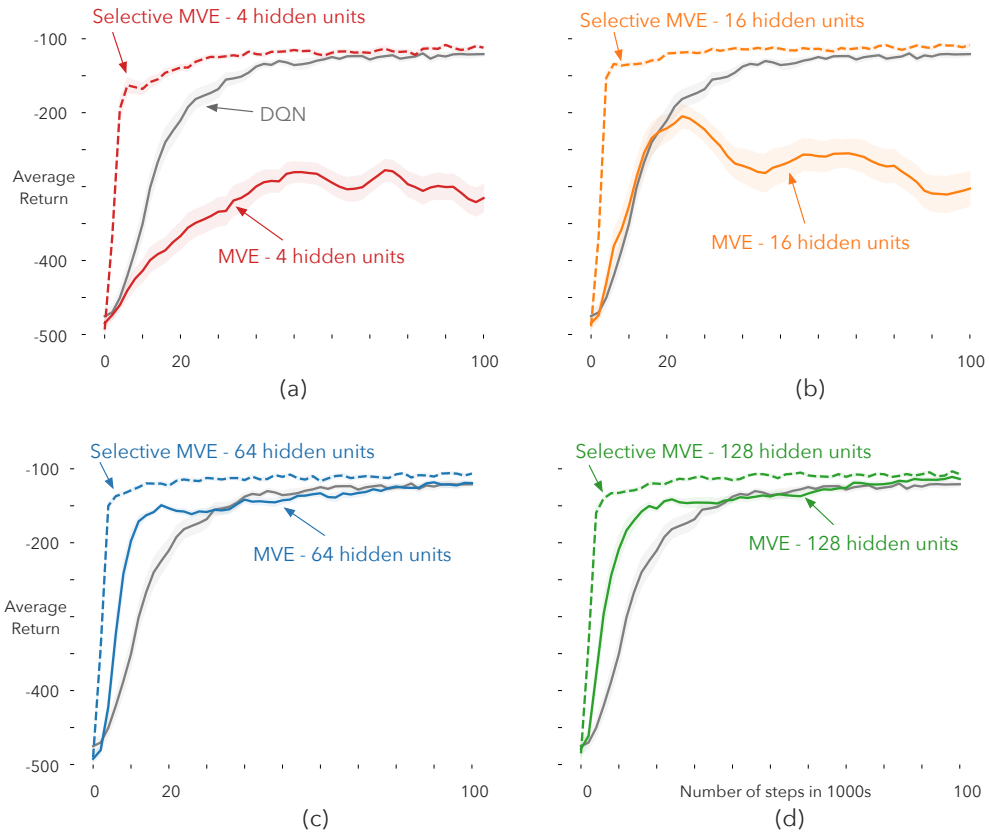

Figure 5.3: Results of Selective MVE ($\tau = 0.1$). The learning curves are averaged over 30 runs; the shaded regions show the standard error. Selective MVE with models of 4 hidden units (a) and 16 hidden units (b) not only matches the asymptotic performance of DQN, but it also achieves better sample-efficiency than the DQN baseline. More interestingly, however, Selective MVE improves the sample-efficiency even in the case of larger models consisting of 64 hidden units (c) and 128 hidden units (d).

5.4 Evaluation of Selective MVE under Capacity Constraints

The results for Selective MVE — for which the action-value function is approximated using a single hidden layer neural network with 128 units — are presented in Figure 5.3. (The results for other value function network architectures are presented at the end of this chapter.) We note that Selective MVE under capacity constraints (models with 4 and 16 hidden units) not only matches the asymptotic performance of DQN — effectively avoiding planning

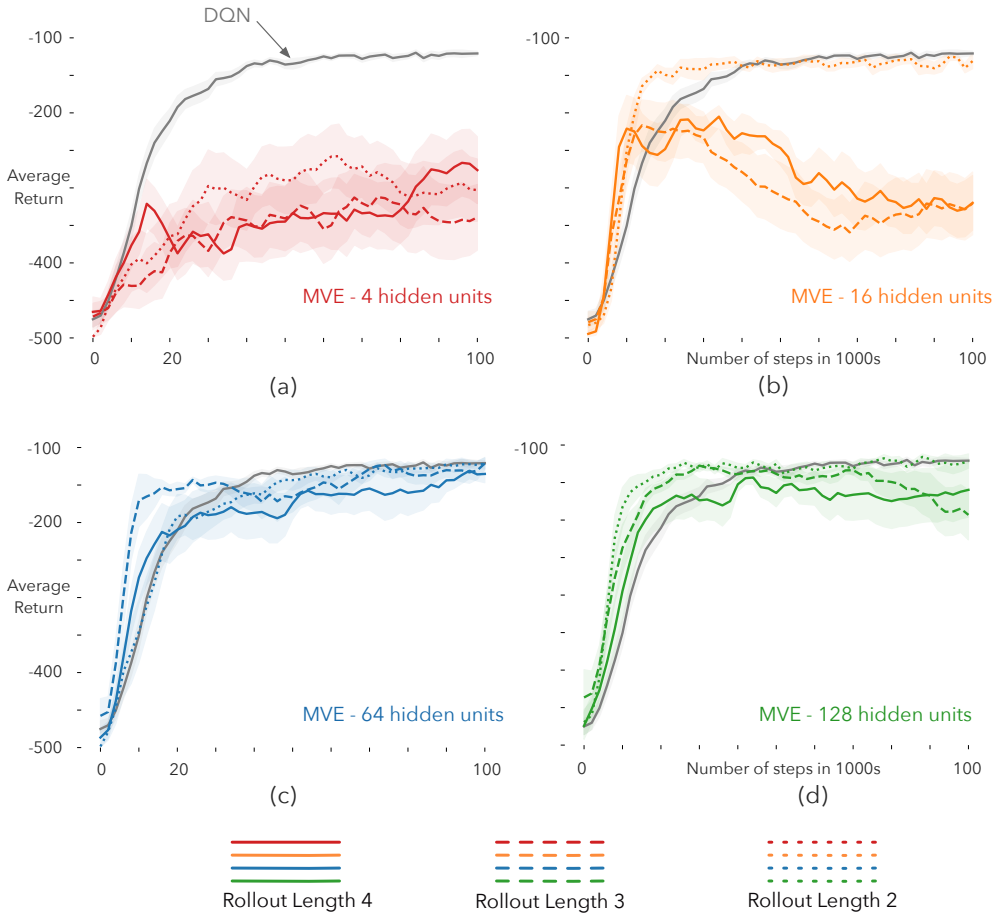

Figure 5.4: Performance of MVE when the model is learned using the loss function from Equation 5.2. The learning curves are averaged over 10 runs; the shaded regions show the standard error. Similar to the squared error loss, the performance deteriorates as we increase the rollout length. However, unlike the squared error loss, the performance of MVE with models of 64 and 128 hidden units also deteriorates as the rollout length is increased. This result suggests that the loss function alone does not explain the superior performance of Selective MVE.

failures — but it is also more sample-efficient than the DQN baseline (Figure 5.3 a-b). More interestingly, however, Selective MVE improves the sample-efficiency even in the case of larger models consisting of 64 and 128 hidden units (Figure 5.3 c-d). To verify that the gains in performance are not due to the change in loss function — Selective MVE uses the loss from Equation 5.2, whereas MVE used the squared error loss — we evaluate MVE with the same loss function as that of Selective MVE. The results, presented in Figure 5.4, suggest that simply changing the loss function does not lead to an accurate model, and that the model still needs to be used selectively.

Expected Rollout Length: To get a better of sense of how Selective MVE is robust to model errors, we compute the expected rollout length of each of the four model sizes. Recall that we compute a weighted average of h-step returns using Equation 5.1; that is, the weights sum to 1. For a sampled (s_0, a_0), we can therefore compute the expected rollout length $\hat{h}$:

$$\hat{h} = 1 \times w_1(s_0, a_0) + 2 \times w_2(s_0, a_0) + 3 \times w_3(s_0, a_0) + 4 \times w_4(s_0, a_0)$$

The expected rollout length of an update is simply the mean of the expected rollout lengths of individual state-action pairs in the sampled batch. We log the mean of the expected rollout length of all updates in the intervals of 2000 steps, and average the curves of independent runs to obtain the reported curve. The learning curves of the expected rollout length are reported in Figure 5.5.

For Selective MVE, there is a clear ordering in the expected rollout length of the four models; $h-$step targets consisting of longer trajectories are given relatively more weight when the model is larger and, therefore, more accurate. Selective MVE with the smallest model of 4 hidden units does not use the model as much as its variants with bigger models, but the limited use is still sufficient to improve the sample efficiency of DQN, while preventing the model inaccuracies to hurt the control performance. The dashed horizontal line at the top, labelled as *Uniform Average*, at the expected rollout length of 2.5, represents MVE when the length of the simulated trajectory is 4; this is be-

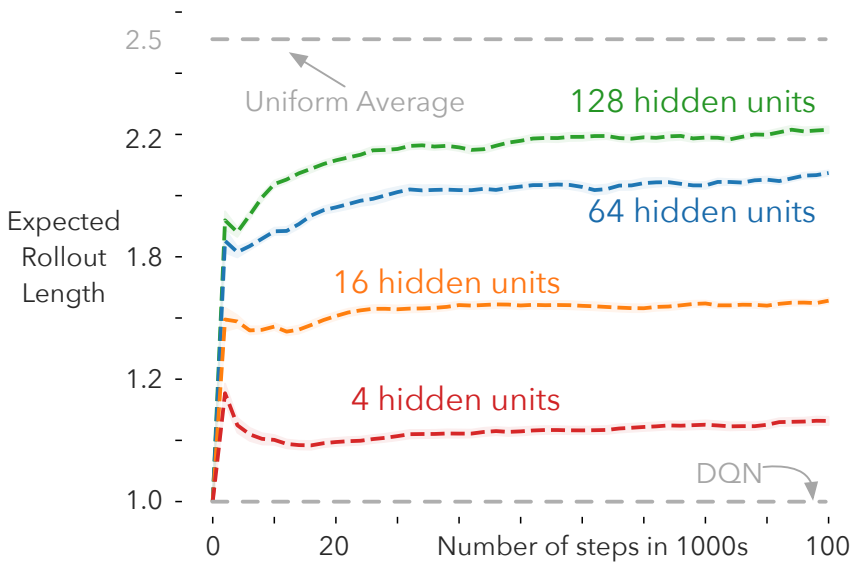

Figure 5.5: Expected Rollout Length of Selective MVE for $\tau = 0.1$. Each reported curve is the average of 30 runs; the shaded regions show the standard error. h-step targets consisting of longer trajectories are given relatively more weight when the model is larger and, therefore, more accurate.

cause MVE uniformly weights all h-step targets. On other hand, the expected rollout length of DQN can be interpreted to be 1: DQN only uses one-step transitions for updating the value function.

This result suggests that the learned covariances can provide a meaningful signal for selective planning.

It is important to point out that simply reducing the rollout length of MVE does not make a limited capacity model useful. Consider Selective MVE with a model consisting of 16 hidden-units for which the expected rollout length is roughly 1.6 (Figure 5.3 b), and contrast this with MVE with a maximum rollout length of 2 leading to an expected rollout length of 1.5 (Figure 5.2 b). While Selective MVE improves sample efficiency and matches DQN's asymptotic's performance, vanilla MVE does neither, suggesting that even a short rollout of length 2 can hurt the performance. This particular comparison suggests that while Selective MVE considers longer rollouts when the model is accurate, it rejects even the shorter ones when the model is inaccurate.

Effect of τ on the performance of Selective MVE:

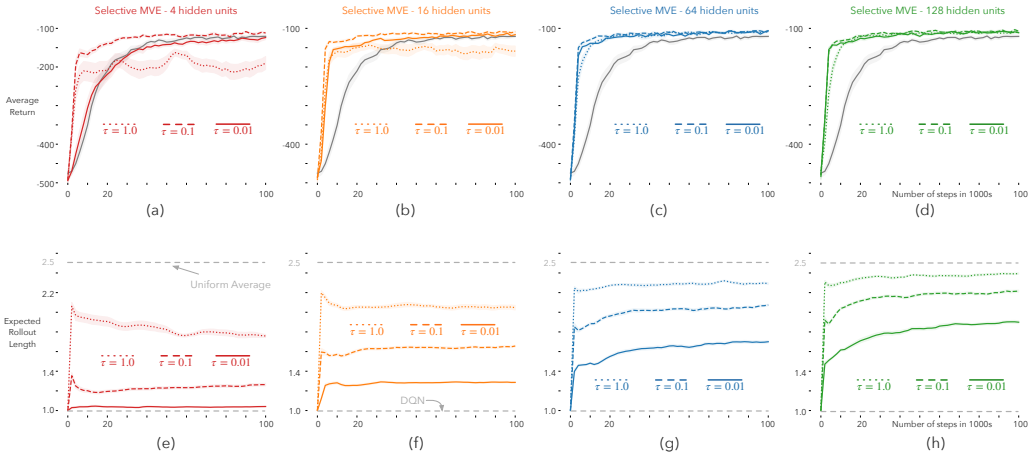

Figure 5.6: Effect of τ on the performance of Selective MVE. Each reported curve is the average of 30 runs; the shaded regions show the standard error. As $\tau \to 0$, Selective MVE reduces to the DQN baseline; on the other hand, as τ is increased, Selective MVE reduces to vanilla MVE.

As mentioned earlier, τ regulates the sensitivity of Selective MVE to the predicted uncertainty. In the previous experiments, we used $\tau = 0.1$ for all model variants. To get a better sense of how τ relates to the robustness of Selective MVE, we now present the results for $\tau = 1.0$ and $\tau = 0.01$ alongside $\tau = 0.1$ in Figure 5.6. Consider the learning curve and the expected rollout length of Selective MVE with 4 hidden units (Figure 5.6-a & 5.6-e). For $\tau = 0.01$, the expected rollout length almost reduces to 1, and the performance becomes indistinguishable from that of DQN. On the other hand, for $\tau = 1.0$, the expected rollout length grows enough to enable the model's inaccuracies to hurt the performance. Contrast this with the Selective MVE variant using a model of 128 hidden units (Figure 5.6-d & 5.6-h). In this case, even $\tau = 0.01$ permits sufficient planning since the model is considerably more accurate than the one with mere 4 hidden units. Interestingly, for $\tau = 1.0$, Selective MVE begins to suffer a noticeable loss in sample efficiency as the expected rollout length is now approaching that of vanilla MVE, which uses the model a bit too excessively.

Concluding Remark: This extended example suggests that a model with inadequate capacity can be selectively useful if reliable estimates of its structural uncertainty can be obtained — the learned variance may satisfy that desideratum.

5.5 Additional Results

We now present results for value function networks of a) single hidden layer with 64 hidden units, b) 2 hidden layer with 64 hidden units each, and c) 2 hidden layer with 128 hidden units each. All reported curves are obtained by averaging 30 runs; the shaded regions represent the standard error. These results are consistent with the discussion in the preceding sections, and provide additional evidence for the utility of selective planning. For instance, they suggest that selective planning is useful even when the value function itself has restricted capacity — the network with only 64 hidden units, for example.

Results for Value Function Network of Single Hidden Layer and 64 Hidden Units

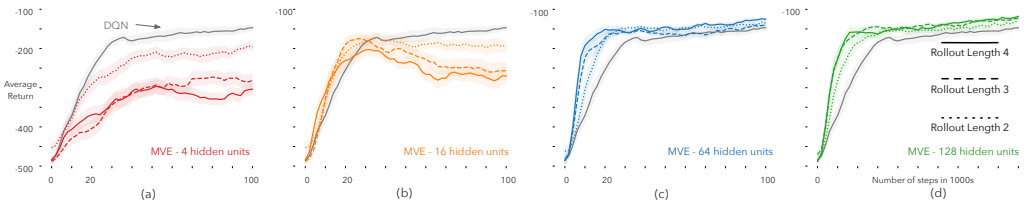

Figure 5.7: The effect of model capacity on MVE's performance.

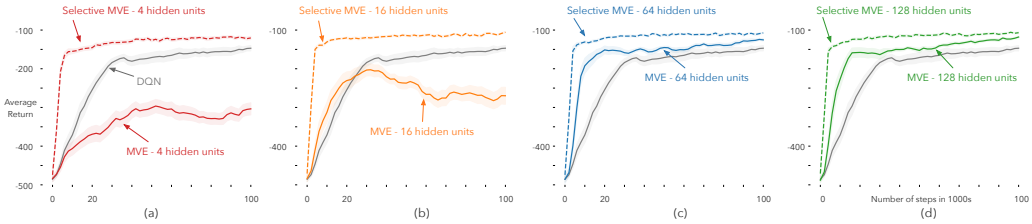

Figure 5.8: Results of selective MVE ($\tau = 0.1$).

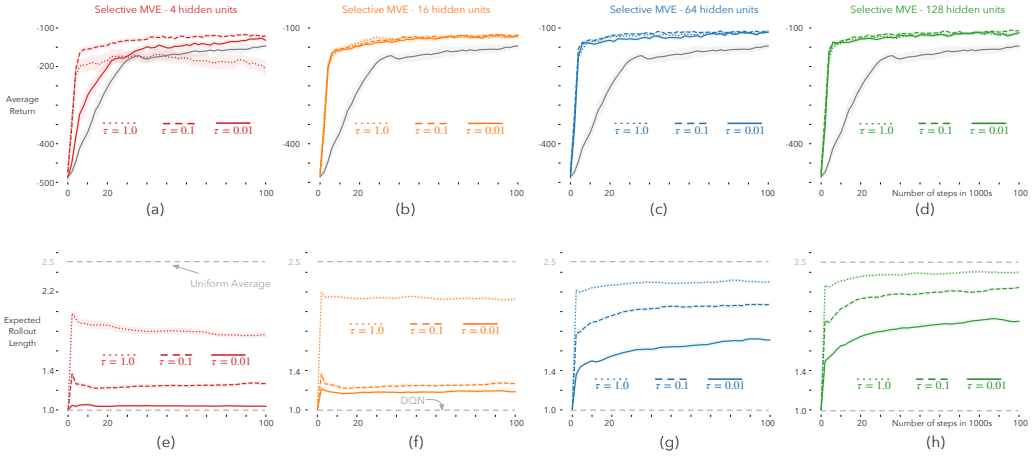

Figure 5.9: Effect of τ on the performance of Selective MVE

Results for Value Function Network of 2 Hidden Layer and 64 Hidden Units Each:

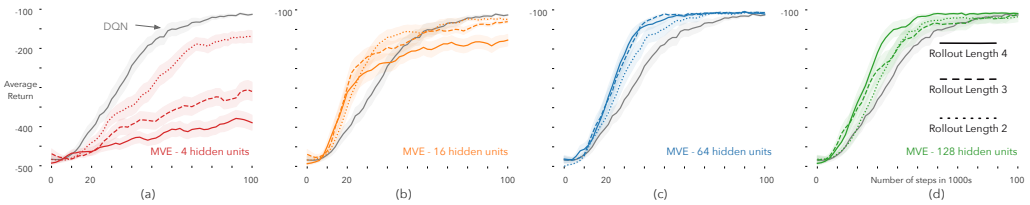

Figure 5.10: The effect of model capacity on MVE's performance.

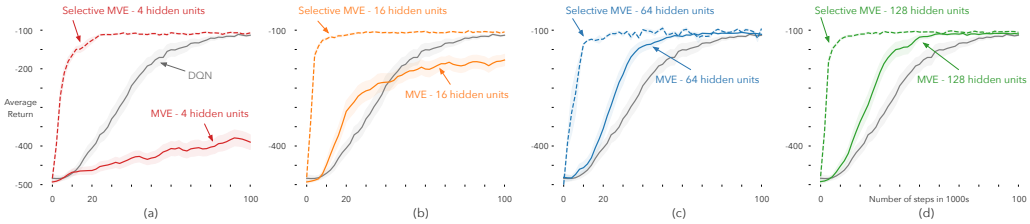

Figure 5.11: Results of selective MVE ($\tau = 0.1$).

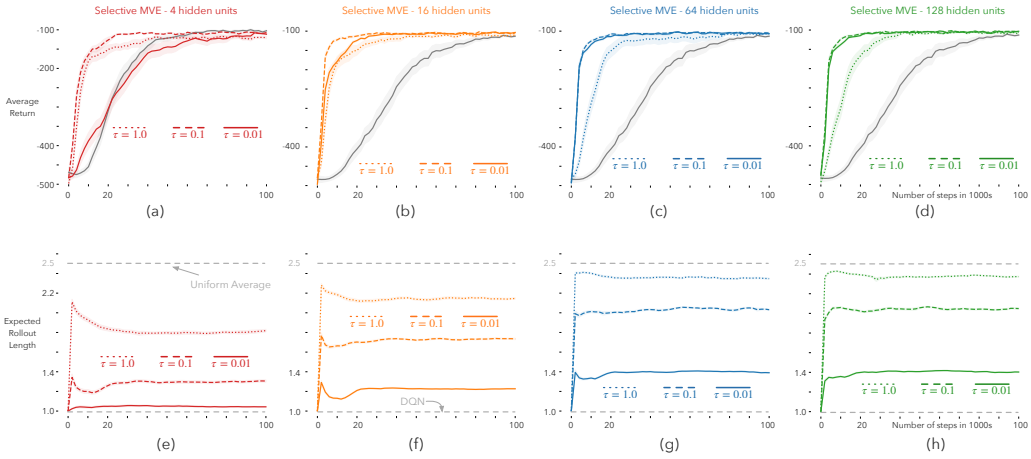

Figure 5.12: Effect of τ on the performance of Selective MVE.

Results for Value Function Network of 2 Hidden Layer and 128 Hidden Units Each:

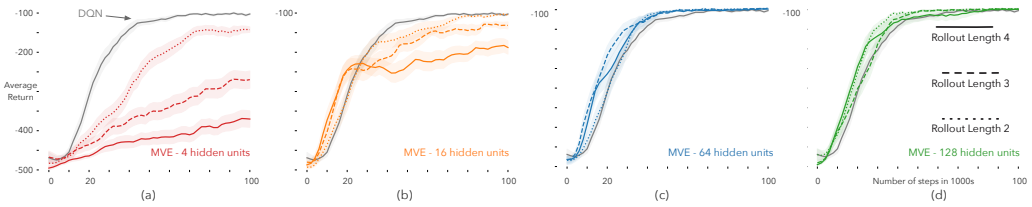

Figure 5.13: The effect of model capacity on MVE's performance.

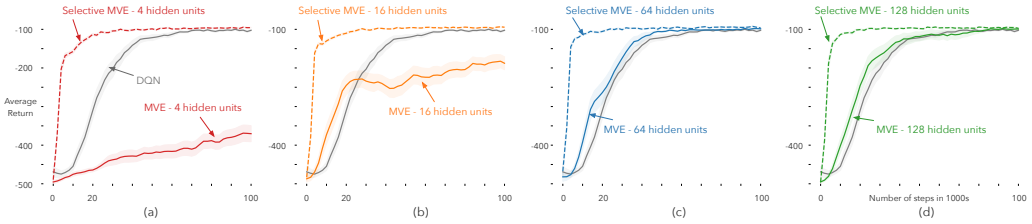

Figure 5.14: Results of selective MVE ($\tau = 0.1$).

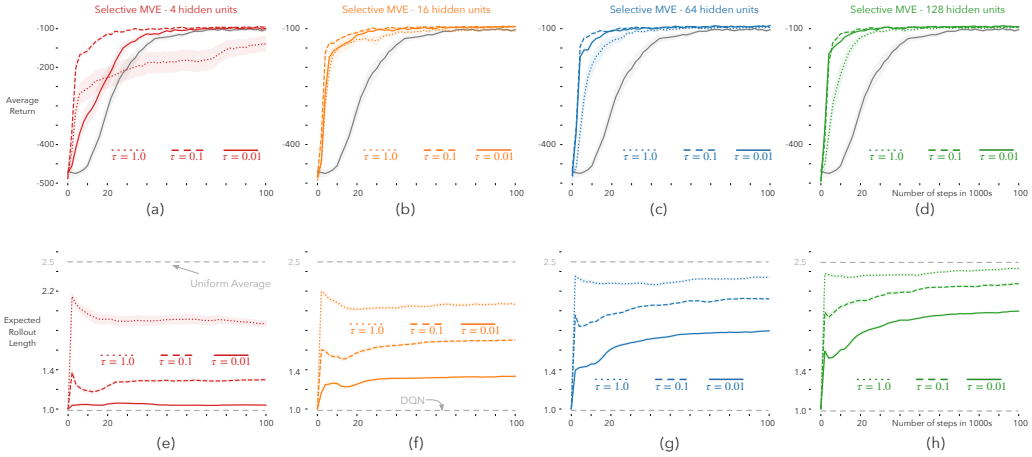

Figure 5.15: Effect of τ on the performance of Selective MVE.

Chapter 6

An Empirical Comparison of Selective Planning Mechanisms

In Chapter 3, we argued that parameter uncertainty by itself is insufficient for selective planning under capacity constraints, and it needs to be used in combination with structural uncertainty. In Chapter 4, we used a simple regression problem to evaluate specific methods that capture parameter uncertainty; we found that, under capacity constraints, the estimated parameter uncertainty resolved when the model made inaccruate predictions: all members of the ensemble converged to the same wrong prediction. In the previous chapter, we empirically demonstrated how the learned variance can provide reliable estimates of structural uncertainty that can enable effective selective planning.

In this chapter, we contrast the ensemble variance with the learned variance in a relatively more complicated setup. The analysis in this chapter is quantitative rather than qualitative: we assess the two approaches primarily in terms of their selective planning performance. We learn the model online using the experience gathered in the replay buffer so the data distribution changes as the agent's policy changes.

In particular, we compare *Selective MVE with learned variance*, as described in the previous chapter, with *Selective MVE with ensemble variance*. The ensemble-based Selective MVE is exactly like the learned-variance variant except for one key difference: the uncertainty, $\sigma(s, a)$, is the variance of the predictions made by the individual members of the ensemble[1].

[1] we simply add the components of the variance vector to obtain a single number

While Selective MVE with ensemble variance is similar to STEVE (Section 2.12) in that it uses ensemble variance to weight the multi-step TD targets, it only uses an ensemble of models and not the value functions to compute the weightings.

6.1 Acrobot Experiments

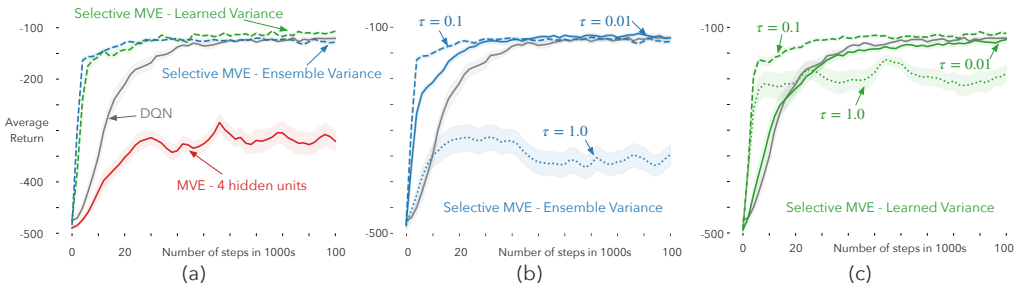

Figure 6.1: Acrobot results. (a) Comparison of ensemble variance and learned variance, in terms of Selective MVE performance, for $\tau = 0.1$. (b) Effect of τ on Selective MVE with ensemble variance. (c) Effect of τ on Selective MVE with learned variance. Similar to the learned variance, the ensemble variance also improves sample-efficiency while avoiding planning failures. The learning curves are averaged over 30 runs; the shaded regions show the standard error.

We first extend the results from the previous chapter by evaluating the ensemble-based Selective MVE on Acrobot. In particular, we extend the results in which the action-value function is approximated with a single layer neural network with 128 units, and the size of the neural network for the model is restricted to 4 hidden units (Section 5.4). The range of values for the parameter sweep, and the configuration of the rest of the hyperparameters are presented in Table 6.1 (all results in this chapter follow the sweep strategy described in Section 5.1.3).

6.1.1 Results

Learning performance: the results, for ensembles consisting of 5 networks, are presented in Figure 6.1. Similar to the learned-variance variant, we note

Table 6.1: Hyperparameters for ensemble-based Selective MVE in Acrobot. Model learning step-size (β) was determined by sweeping over the range specified in the respective row.

Hyperparameter	Values
Optimizer	Adam
Model learning step-size (β)	0.1, 0.01, 0.001, 0.0001
Batch size	16
Loss function	Squared error
Model learning frequency	1 update for every environment step
Simulated trajectory length	4
Softmax temperature (τ)	1.0, 0.1, 0.01
Number of networks	2, 3, 4, 5

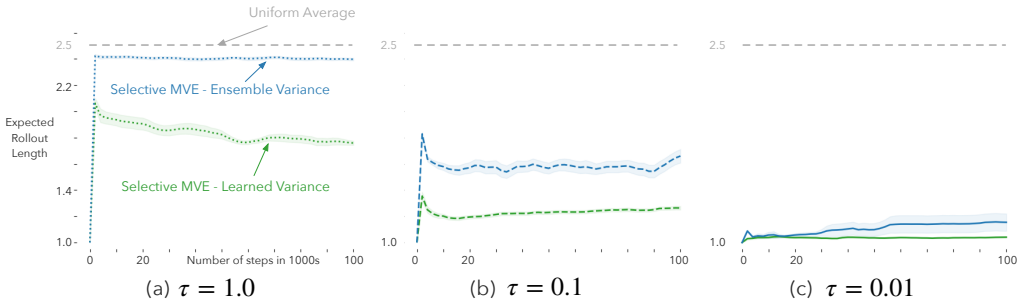

Figure 6.2: Comparison of expected rollout length of the two variants Selective MVE, for fixed values of τ, in Acrobot. For a given value of τ, ensemble-based selective MVE uses longer rollouts on average than Selective MVE with learned variance. Each reported curve is the average of 30 runs; the shaded regions show the standard error.

that an intermediate value of τ provides sample-efficiency benefits while avoiding planning failures; the results show that, in this example, ensemble variance also effectively identifies model errors.

Effect of τ on the expected rollout length: we find that, for a given value of τ, ensemble-based Selective MVE uses longer rollouts on average than Selective MVE with learned variance (Figure 6.2); while the relatively longer rollouts are innocuous in this example, it may still affect the performance negatively in some problems, as we will see in the next couple of examples.

Effect of the number of networks in the ensemble: we find that the performance of ensemble-based Selective MVE improves as we increase the

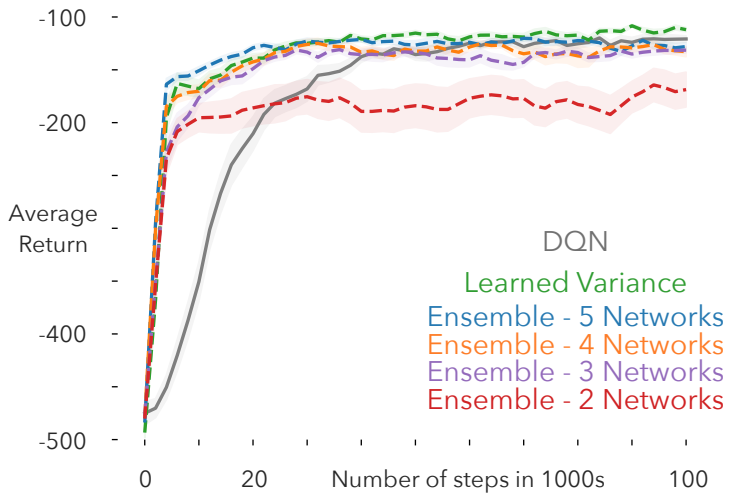

Figure 6.3: Effect of the number of networks on the performance of ensemble-based Selective MVE in Acrobot. The performance improves as we increase the number of networks in the ensemble. Each reported curve is the average of 30 runs; the shaded regions show the standard error.

number of networks in the ensemble (Figure 6.3).

Figure 6.4: A pictorial depiction of Cartpole (left) and Navigation (right)

6.2 Cartpole Experiments

We evaluate the two variants of Selective MVE on Cartpole (Barto *et al.* 1983), a classic control environment. The goal is to balance a pole attached to a movable cart, as depicted in Figure 6.4 (left). The state consists of the current position of the cart on the x-axis, the velocity of the cart, the angle of the pole with respect to the vertical axis, and the rate of change of the angle, leading to a 4-dimensional state representation. The agent can move the cart to the left or to the right; the agent's action space is discrete and consists of two actions. The reward is +1 on all time steps. The episode terminates when the pole's angle with respect to the vertical axis exceeds 15°, or when the environment interaction exceeds 200 time-steps, whichever occurs first. In our experiments, we use OpenAI gym's implementation of Cartpole (Brockman *et al.* 2016).

For the DQN baseline, we estimate the action-value function using a single hidden layer network of 128 hidden units; we determine the best setting for the step-size, the batch size, and the replay memory size by sweeping over possible parameter configurations (see Table 6.2).

For MVE variants, we extend the DQN baseline and evaluate two neural network sizes for the model: a single hidden layer network with only 2 hidden units, and a single hidden layer network with 64 hidden units. The range of values for the parameter sweep, and the configuration of the rest of the hyperparameters are presented in Table 6.3 (MVE), Table 6.4 (Selective MVE

with learned variance), and Table 6.5 (Selective MVE with ensemble variance).

Table 6.2: DQN hyperparameters in Cartpole. The step-size, the batch size, and the replay memory size were determined by sweeping over the range specified in the respective rows.

Hyperparameter	**Values**
Optimizer	RMSProp
Step-size (α)	0.03, 0.01, 0.003, 0.001, 0.0003, 0.0001
Replay memory size	10000, 20000, 50000
Target network update frequency	512, 1024, 2048
Batch size	16
Training frequency	1 update for every environment step
Exploration rate (ϵ)	0.1
Discount factor (γ)	0.95

Table 6.3: MVE specific hyperparameters in Cartpole. The model learning step-size (β) was determined by sweeping over the range specified in the respective row.

Hyperparameter	**Values**
Optimizer	Adam
Model learning step-size (β)	0.1, 0.01, 0.001, 0.0001
Batch size	16
Loss function	Squared error
Model learning frequency	1 update for every environment steps
Simulated trajectory length	4

Table 6.4: Hyperparameters for Selective MVE with learned variance. Model learning step-size (β) was determined by sweeping over the range specified in the respective row.

Hyperparameter	**Values**
Optimizer	Adam
Model learning step-size (β)	0.1, 0.01, 0.001, 0.0001
Batch size	16
Loss function	Equation 5.2
Model learning frequency	1 update for every environment step
Simulated trajectory length	4
Softmax temperature (τ)	0.1, 0.01, 0.001, 0.0001

Table 6.5: Hyperparameters for Selective MVE with ensemble variance. Model learning step-size (β) was determined by sweeping over the range specified in the respective row.

Hyperparameter	**Values**
Optimizer	Adam
Model learning step-size (β)	0.1, 0.01, 0.001, 0.0001
Batch size	16
Loss function	Squared error
Model learning frequency	1 update for every environment step
Simulated trajectory length	4
Number of networks	5
Softmax temperature (τ)	0.1, 0.01, 0.001, 0.0001

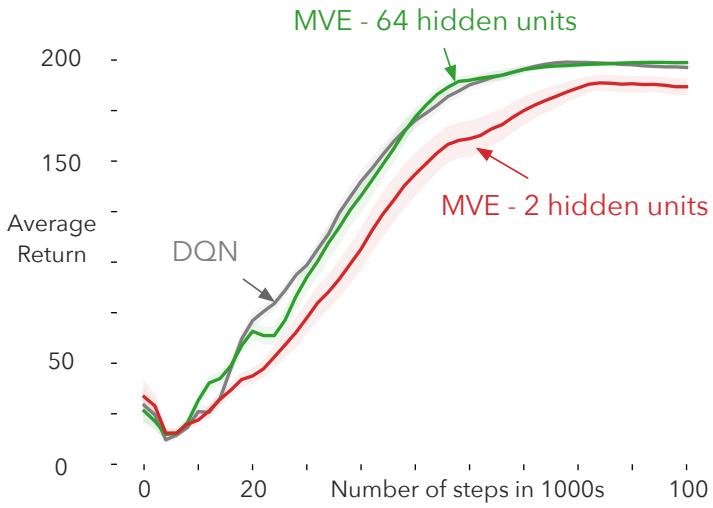

Figure 6.5: The performance of MVE with the two model sizes: 2 hidden units and 64 hidden units. The MVE variant with the smaller model (2 hidden units) performs worse than the DQN baseline. The MVE variant with the bigger model (64 hidden units) improves the sample-efficiency only slightly. The learning curves are averaged over 30 runs; the shaded regions show the standard error.

6.2.1 Results

Figure 6.5 shows the performance of MVE for the two model sizes. While the MVE variant with the bigger model (64 hidden units) does not improve sample-efficiency over the DQN baseline, the MVE variant with the smaller model (2 hidden units) performs worse than the DQN baseline.

Results for the smaller model of 2 hidden units: Figure 6.6 compares the performance of the two variants of Selective MVE for the smaller model of 2 hidden units. As we progressively reduce the value of τ from 0.1 to 0.0001, Selective MVE with learned variance improves, and eventually outperforms the DQN baseline. On the other hand, the performance of Selective MVE with ensemble variance does not improve noticeably. Figure 6.7 helps explain this result by showing the expected rollout length for each value of τ. The expected rollout length of Selective MVE with learned variance decreases as we reduce the value of τ. On the other hand, the decrease in the expected rollout length for the ensemble-based variant plateaus, and is not as systematic

as it is in the case of the learned variance.

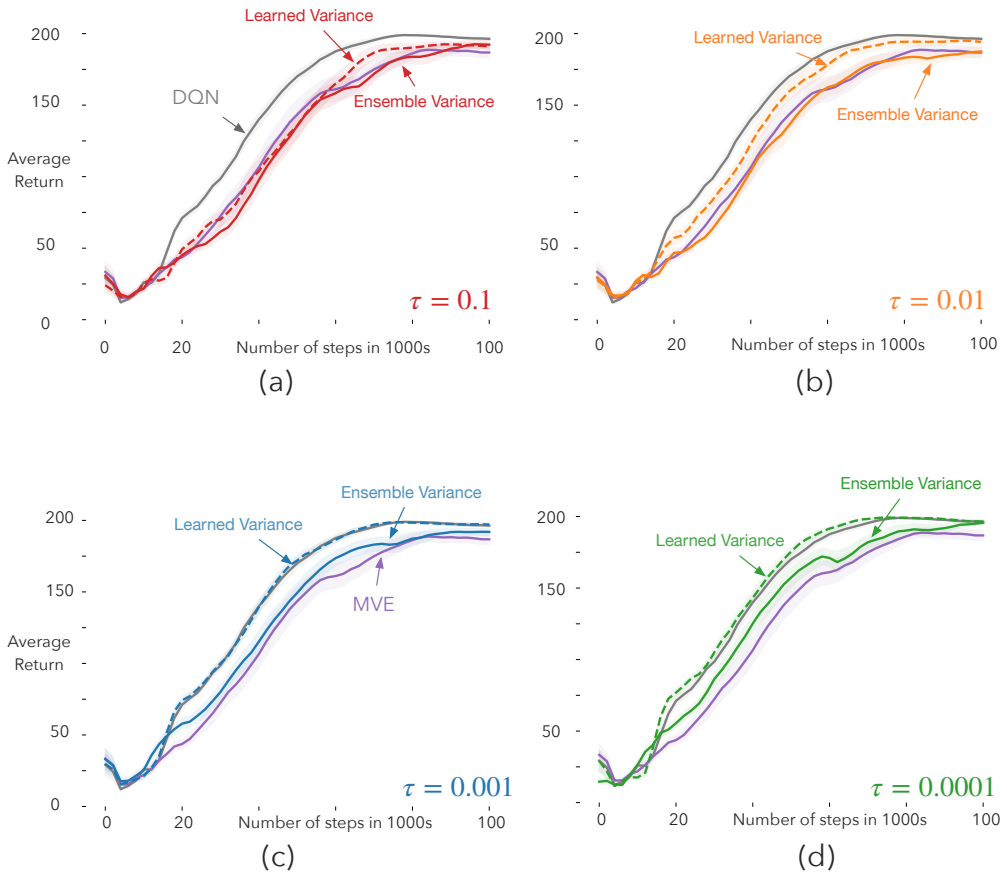

Figure 6.6: Evaluation of the two variants of Selective MVE, in Cartpole, for neural network models consisting of 2 hidden units. The performance of Selective MVE with learned variance improves as we reduce the value of τ from 0.1 to 0.0001, and eventually eclipses the performance of the DQN baseline. On the other hand, the performance Selective MVE with ensemble variance does not improve noticeably. Each reported curve is the average of 30 runs; the shaded regions show the standard error.

Results for the bigger model of 64 hidden units: Figure 6.8 compares the performance of the two variants of Selective MVE for the bigger model of 64 hidden units; Figure 6.9 shows the expected rollout length for each value of τ, for each of the two variants of Selective MVE.

In this case, Selective MVE with learned variance outperforms the DQN and MVE baselines for all values of τ, except for $\tau = 0.01$, where it matches the

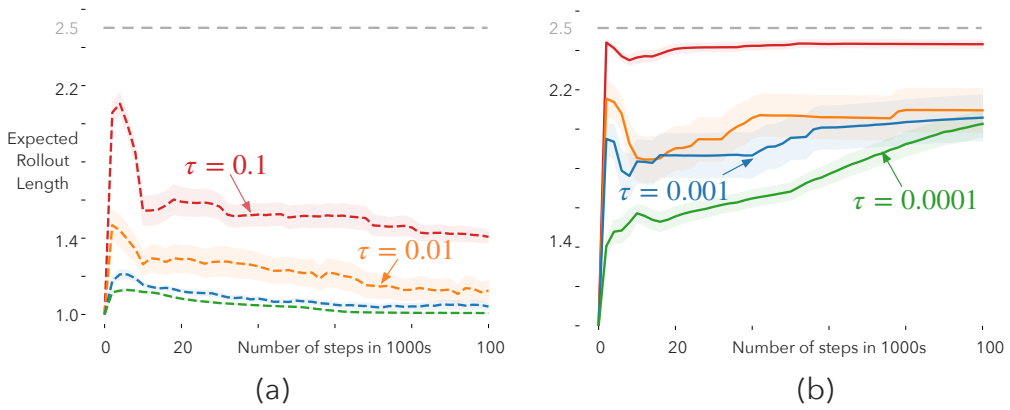

Figure 6.7: Comparison of the expected rollout length of the two variants of Selective MVE, in Cartpole, for neural network models consisting of 2 hidden units. While the expected rollout length of Selective MVE with learned variance reduces as we reduce the value of τ, the decrease in the expected rollout length for the ensemble-based variant plateaus, and is not as systematic as it is in the case of the learned variance. The curves are averaged over 30 runs; the shaded regions show the standard error.

performance of the baselines. On the other hand, the performance Selective MVE with ensemble variance is not any better than the performance of the DQN and MVE baselines for any of the τ values. Particularly interesting is the contrast between the ensemble-based Selective MVE with $\tau = 0.0001$ and the learned-variance variant with $\tau = 0.001$ and $\tau = 0.0001$. While the expected rollout of length of the ensemble-based Selective MVE is somewhere between the expected rollout lengths of the learned-variance variants (Figure 6.9), its performance is worse than either of the two learned-variance variants (Figure 6.8-c and 6.8-c).

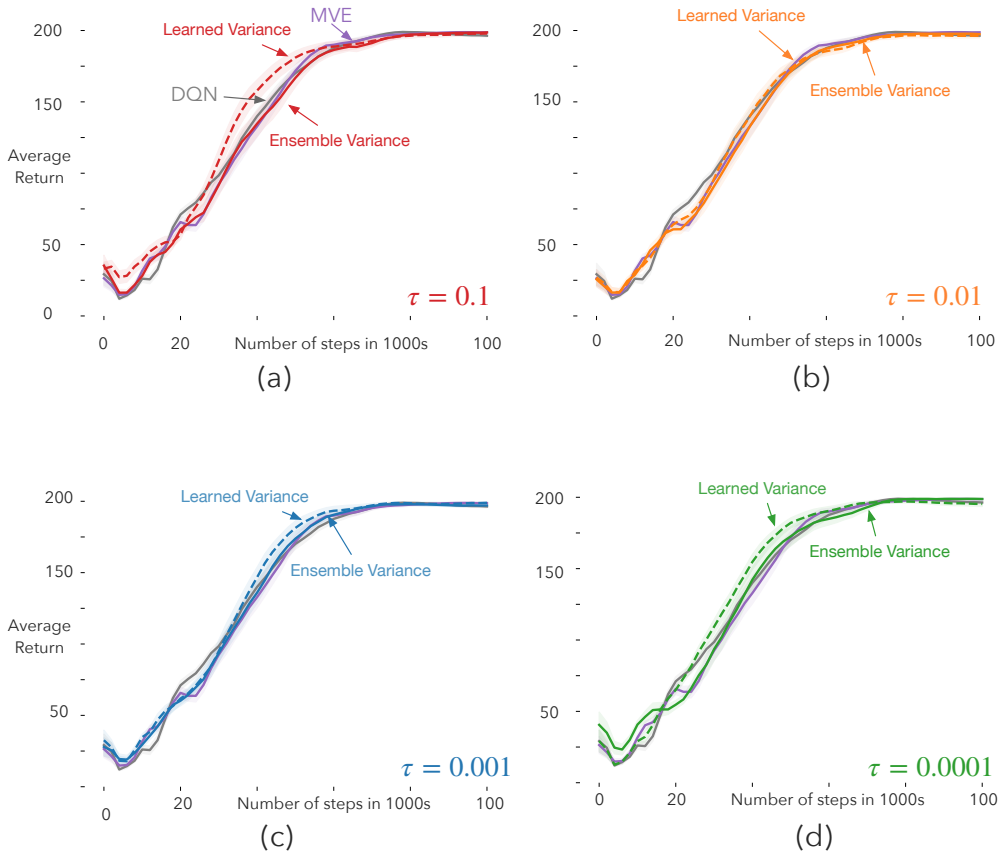

Figure 6.8: Evaluation of the two variants of Selective MVE in Cartpole for neural network models consisting of 64 hidden units. Each reported curve is the average of 30 runs; the shaded regions show the standard error.

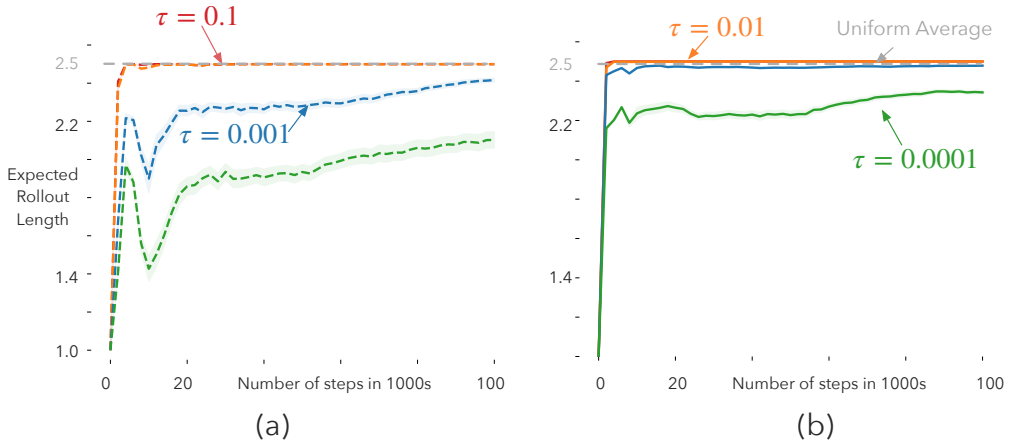

Figure 6.9: Comparison of the expected rollout length of the two variants of Selective MVE in Cartpole for neural network models consisting of 64 hidden units. The curves are averaged over 30 runs; the shaded regions show the standard error.

6.3 Navigation Experiments

We now evaluate the two variants of selective MVE in a continuous two-dimensional navigation environment, as depicted in Figure 6.4 (right). The state consists of the (x, y) coordinates of the agent, where $x, y \in (0, 1)$. The goal is to navigate to the green block, where $x > 0.9$ and $y > 0.9$. Each episode starts in one of the four blue blocks chosen at random. The agent can move in any of the four cardinal directions; the agent's action space is discrete and consists of four actions. An action moves the agent in the intended direction with an offset of 0.025, except when the agent runs into a wall, in which case the agent stays in its position. The reward is zero everywhere except when the agent transitions into the green block, where the reward is $+1$. The episode terminates when the agent reaches the goal, or when the environment interaction exceeds 1000 time-steps, whichever occurs first.

For the DQN baseline, we estimate the action-value function using a single hidden layer network of 256 hidden units; we determine the best setting for the step-size, the batch size, and the replay memory size by sweeping over possible parameter configurations (see Table 6.6).

For MVE variants, we extend the DQN baseline and evaluate two neural network sizes for the model: a single hidden layer network with only 4 hidden units, and a single hidden layer network with 64 hidden units. The range of values for the parameter sweep, and the configuration of the rest of the hyperparameters are presented in Table 6.7 (MVE), Table 6.8 (Selective MVE with learned variance), and Table 6.9 (Selective MVE with ensemble variance).

6.3.1 Results

Figure 6.10 shows the performance of MVE for the two model sizes. While the MVE variant with the smaller model (4 hidden units) performs worse than the DQN baseline, the MVE variant with the bigger model (64 hidden units) results in improved sample-efficiency.

Figure 6.11 compares the performance of the two variants of Selective MVE for the smaller model of 4 hidden units. As we progressively reduce the value

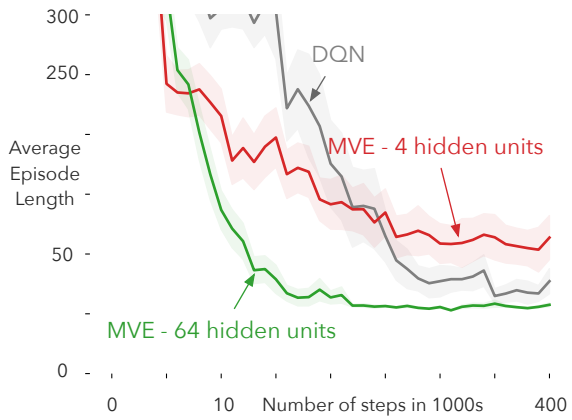

Figure 6.10: The performance of MVE with the two model sizes: 4 hidden units and 64 hidden units. The curves are averaged over 30 runs; the shaded regions show the standard error.

of τ from 0.01 to 0.00001, Selective MVE with learned variance improves, and eventually achieves stable asymptotic performance (Figure 6.11-d), while still being sample-efficienct. On the other hand, while the performance of Selective MVE with ensemble variance is better than vanilla MVE with 4 hidden units, it does not improve as consistently as the learned variance, when the value τ is decreased (compare, for example, the performance of $\tau = 0.0001$ (Figure 6.11-c) and $\tau = 0.00001$ (Figure 6.11-d). Figure 6.12 shows the expected rollout length for each value of τ. The expected rollout length of the learned-variance variant decreases as we reduce the value of τ. On the other hand, exepected rollout length of the ensemble-based variant also decreases, but the decrease is not as dramatic as it is in the case of the learned variance.

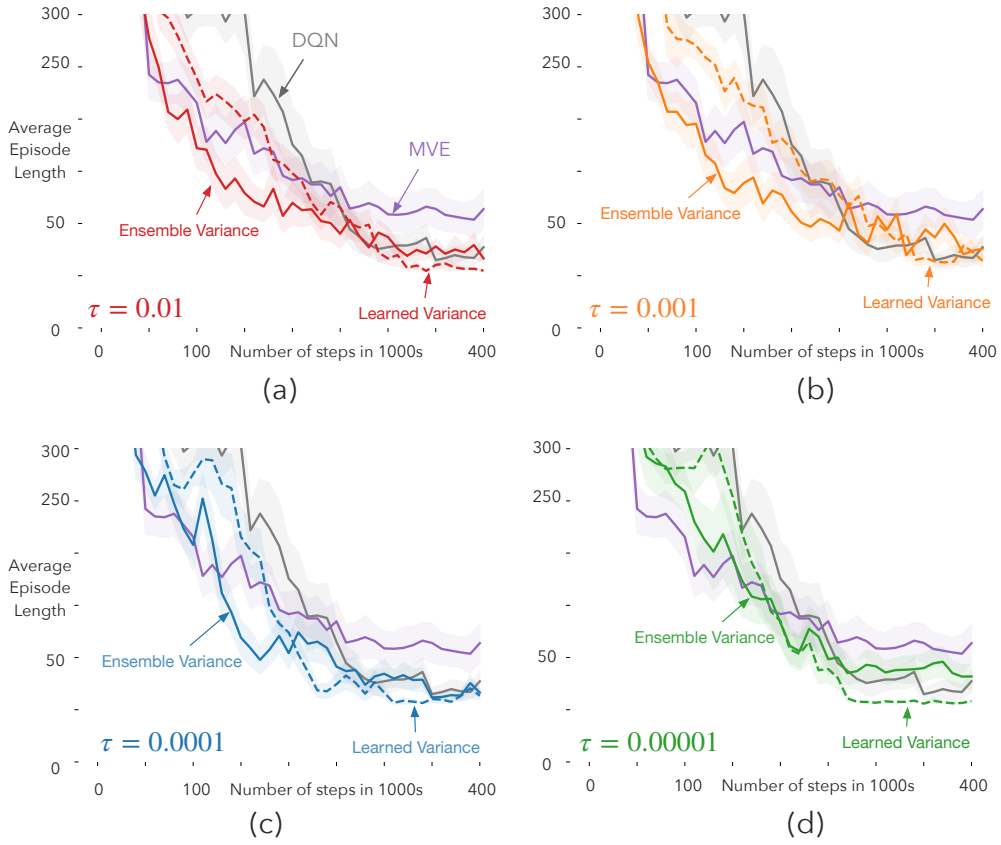

Figure 6.11: Evaluation of the two variants of Selective MVE, in Navigation, for neural network models consisting of 4 hidden units. Each reported curve is the average of 30 runs; the shaded regions show the standard error.

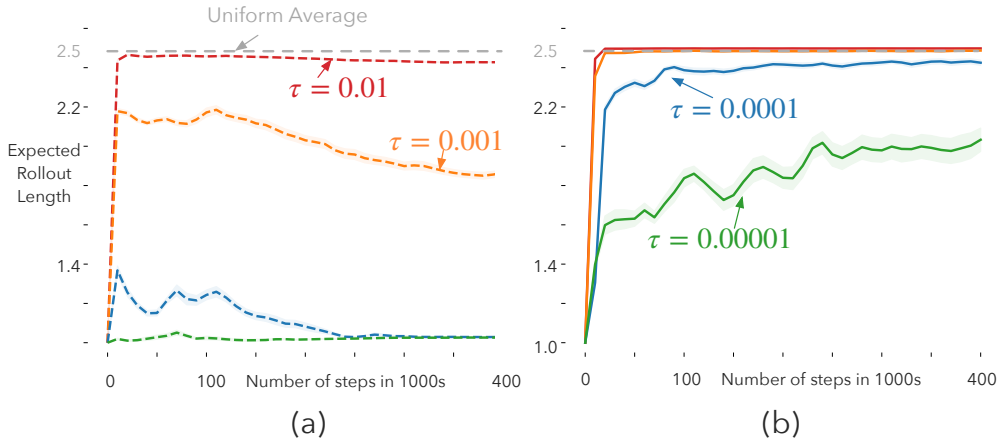

Figure 6.12: Comparison of the expected rollout length of the two variants of Selective MVE, in Navigation, for neural network models consisting of 4 hidden units. The curves are averaged over 30 runs; the shaded regions show the standard error.

Concluding Remark: The results in this chapter show that, under limited capacity settings, the learned variance can be more effective as a selective planning mechanism than the ensemble variance. More importantly, these results provide additional evidence in the ability of the learned variance for enabling effective selective planning. More work needs to be done for understanding the trade-offs between the two mechanisms, and devising a selective planning method that combines that two approaches; we leave that for future work.

Table 6.6: DQN hyperparameters in Navigation. The step-size, the batch size, and the replay memory size were determined by sweeping over the range specified in the respective rows.

Hyperparameter	**Values**
Optimizer	RMSProp
Step-size (α)	0.03, 0.01, 0.003, 0.001, 0.0003, 0.0001, 0.00003
Replay memory size	10000, 20000, 50000
Target network update frequency	512, 1024, 2048
Batch size	16
Training frequency	1 update for every environment step
Exploration rate (ϵ)	0.1
Discount factor (γ)	0.99

Table 6.7: MVE specific hyperparameters in Navigation. The model learning step-size (β) was determined by sweeping over the range specified in the respective row.

Hyperparameter	**Values**
Optimizer	Adam
Model learning step-size (β)	0.1, 0.01, 0.001, 0.0001
Batch size	16
Loss function	Squared error
Model learning frequency	1 update for every environment steps
Simulated trajectory length	4

Table 6.8: Hyperparameters for selective MVE with learned variance. Model learning step-size (β) was determined by sweeping over the range specified in the respective row.

Hyperparameter	**Values**
Optimizer	Adam
Model learning step-size (β)	0.1, 0.01, 0.001, 0.0001
Batch size	16
Loss function	Equation 5.2
Model learning frequency	1 update for every environment step
Simulated trajectory length	4
Softmax temperature (τ)	0.01, 0.001, 0.0001, 0.00001

Table 6.9: Hyperparameters for selective MVE with ensemble variance. Model learning step-size (β) was determined by sweeping over the range specified in the respective row.

Hyperparameter	**Values**
Optimizer	Adam
Model learning step-size (β)	0.1, 0.01, 0.001, 0.0001
Batch size	16
Loss function	Squared error
Model learning frequency	1 update for every environment step
Simulated trajectory length	4
Number of networks	5
Softmax temperature (τ)	0.1, 0.01, 0.001, 0.0001

Chapter 7

Conclusion and Future Directions

In this thesis, we investigated the idea of selective planning: the agent should plan only in parts of the state space where the model is accurate. We treated the problem of determining when the model is accurate as that of uncertainty estimation; we discussed three types of uncertainty: parameter, aleatoric, and structural. We highlighted the importance of *structural uncertainty* for selective planning under limited model capacity: structural uncertainty signals the model errors due to limited capacity. We showed that the learned input-dependent variance, under the standard Gaussian assumption, can reveal the presence of structural uncertainty.

We performed a suite of experiments to investigate the ability of the learned variance to combat planning failures that are caused by inadequate model capacity. Specifically, we evaluated the performance of Model-based Value Expansion (MVE), a planning algorithm that uses the learned model to construct multi-step TD targets for evaluating the greedy policy (Feinberg *et al.* 2018); we found that MVE can fail when the model is subject to capacity constraints. We then evaluated the performance of Selective MVE, an instance of selective planning which weights the multi-step TD targets according to the structural uncertainty in the model's predictions; we found that Selective MVE avoids planning failures and, at the same time, improves sample-efficiency. The results show that selective planning with the learned variance can be useful, even when planning with the model non-selectively would cause catastrophic

failure.

The core idea of selective planning is general and there are many aspects to be examined. We point out a couple of research directions that warrant further investigation.

While we focused exclusively on the limited capacity scenario, insufficient coverage and stochasticity are also important sources of model errors. Devising a selective planning mechanism that combines parameter, aleatoric, and structural uncertainties requires further work.

The efficacy of the learned variance for robust selective planning hinges on the state representation: the predicted variance is proportional to the squared error in the representation space (see Equation 4.1). It is not difficult to think of a state representation for which small errors in the representation space lead to large biases in the TD targets. Thus, structural uncertainty estimates need to be *value aware*: predicted variance should be proportional to the error in the TD targets for the value function update.

References

[1] Asadi, K. Strengths, weaknesses, and combinations of model-based and model-free reinforcement learning, Master's thesis, University of Alberta, 2015. 14

[2] Barber, D., AND Bishop, C. M., Ensemble learning in bayesian neural networks, *Nato ASI Series F Computer and Systems Sciences*, vol. 168, 215–238, 1998. 24, 25

[3] Barto, A. G., Sutton, R. S., AND Anderson, C. W., Neuronlike adaptive elements that can solve difficult learning control problems, *IEEE transactions on systems, man, and cybernetics*, no. 5, 834–846, 1983. 57

[4] Bayes, T., Lii. an essay towards solving a problem in the doctrine of chances. by the late rev. mr. bayes, frs communicated by mr. price, in a letter to john canton, amfr s, *Philosophical transactions of the Royal Society of London*, no. 53, 370–418, 1763. 22, 23

[5] Bellemare, M. G., Naddaf, Y., Veness, J., AND Bowling, M., The arcade learning environment: An evaluation platform for general agents, *Journal of Artificial Intelligence Research*, vol. 47, 253–279, 2013. 2, 11

[6] Bishop, C. M. *Pattern recognition and machine learning*. Springer Science+ Business Media, 2006. 23

[7] Bishop, C. M., Lawrence, N. D., Jaakkola, T., AND Jordan, M. I. Approximating posterior distributions in belief networks using mixtures, in *Advances in Neural Information Processing Systems*, 1998, 416–422. 24

[8] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., AND Zaremba, W., Openai gym, *arXiv preprint arXiv:1606.01540*, 2016. 38, 57

[9] Brown, N., AND Sandholm, T. Libratus: The superhuman ai for no-limit poker, in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, 5226–5228. DOI: 10.24963/ijcai.2017/772. [Online]. Available: https://doi.org/10.24963/ijcai.2017/772. 2

[10] Buckman, J., Hafner, D., Tucker, G., Brevdo, E., AND Lee, H. Sample-efficient reinforcement learning with stochastic ensemble value expansion, in *Advances in Neural Information Processing Systems*, 2018, 8224–8234. 15

[11] De Finetti, B. La prévision: Ses lois logiques, ses sources subjectives, in *Annales de l'institut Henri Poincaré*, vol. 7, 1937, 1–68. 27

[12] Deisenroth, M., AND Rasmussen, C. Pilco: A model-based and data-efficient approach to policy search, in *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*, Omnipress, 2011, 465–472. 2

[13] Efron, B. *The jackknife, the bootstrap, and other resampling plans*. Siam, 1982, vol. 38. 25

[14] Feinberg, V., Wan, A., Stoica, I., Jordan, M. I., Gonzalez, J. E., AND Levine, S., Model-based value estimation for efficient model-free reinforcement learning, *arXiv preprint arXiv:1803.00101*, 2018. 3, 12, 14, 37, 71

[15] Fushiki, T. *et al.*, Bootstrap prediction and bayesian prediction under misspecified models, *Bernoulli*, vol. 11, no. 4, 747–758, 2005. 25

[16] Fushiki, T., Komaki, F., Aihara, K., *et al.*, Nonparametric bootstrap prediction, *Bernoulli*, vol. 11, no. 2, 293–307, 2005. 25

[17] Gal, Y. Uncertainty in deep learning, PhD thesis, University of Cambridge, 2016. 25

[18] Gal, Y., AND Ghahramani, Z. Dropout as a bayesian approximation: Representing model uncertainty in deep learning, in *International Conference on Machine Learning (ICML)*, 2016, 1050–1059. 25

[19] Gal, Y., Hron, J., AND Kendall, A. Concrete dropout, in *Advances in Neural Information Processing Systems*, 2017, 3581–3590. 25

[20] Geman, S., Bienenstock, E., AND Doursat, R., Neural networks and the bias/variance dilemma, *Neural Computation*, vol. 4, no. 1, 1–58, 1992. 15, 20, 21

[21] Glorot, X., AND Bengio, Y. Understanding the difficulty of training deep feedforward neural networks, in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010, 249–256. 30

[22] Graves, A. Practical variational inference for neural networks, in *Advances in Neural Information Processing Systems*, 2011, 2348–2356. 25

[23] Hafner, D., Lillicrap, T. P., Fischer, I., Villegas, R., Ha, D., Lee, H., AND Davidson, J. Learning latent dynamics for planning from pixels, in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, 2019, 2555–2565. 2

[24] Hasselt, H. van, Hessel, M., AND Aslanides, J. When to use parametric models in reinforcement learning? In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, 2019, 14322–14333. 2, 12

[25] Hinton, G., AND Van Camp, D. Keeping neural networks simple by minimizing the description length of the weights, in *in Proc. of the 6th Ann. ACM Conf. on Computational Learning Theory*, Citeseer, 1993. 24, 25

[26] Holland, G. Z., Talvitie, E. J., AND Bowling, M., The effect of planning shape on dyna-style planning in high-dimensional state spaces, *arXiv preprint arXiv:1806.01825*, 2018. 12

[27] Kapturowski, S., Ostrovski, G., Quan, J., Munos, R., AND Dabney, W. Recurrent experience replay in distributed reinforcement learning, in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019. 2

[28] Kingma, D. P., AND Ba, J. Adam: A method for stochastic optimization, in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. 30

[29] Lakshminarayanan, B., Pritzel, A., AND Blundell, C. Simple and scalable predictive uncertainty estimation using deep ensembles, in *Advances in Neural Information Processing Systems*, 2017, 6402–6413. 25, 26, 43

[30] Li, Y., AND Gal, Y. Dropout inference in bayesian neural networks with alpha-divergences, in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, 2052–2061. 25

[31] Lin, L.-J., Self-improving reactive agents based on reinforcement learning, planning and teaching, *Machine learning*, vol. 8, no. 3-4, 293–321, 1992. 11

[32] Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M. J., AND Bowling, M., Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents, *J. Artif. Intell. Res.*, vol. 61, 523–562, 2018. DOI: 10.1613/jair.5699. 2

[33] MacKay, D. J. Bayesian methods for adaptive models, PhD thesis, California Institute of Technology, 1992. 24

[34] 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.*, Human-level control through deep reinforcement learning, *Nature*, vol. 518, no. 7540, 529, 2015. 11, 38

[35] Moore, A. W., AND Atkeson, C. G., Prioritized sweeping: Reinforcement learning with less data and less time, *Machine learning*, vol. 13, no. 1, 103–130, 1993. 9

[36] Moravčík, M., Schmid, M., Burch, N., Lisỳ, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M., AND Bowling, M., Deepstack: Expert-level artificial intelligence in heads-up no-limit poker, *Science*, vol. 356, no. 6337, 508–513, 2017. 2

[37] Neal, R. M. Bayesian learning for neural networks, PhD thesis, 1995. 24

[38] Nix, D. A., AND Weigend, A. S. Estimating the mean and variance of the target probability distribution, in *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, IEEE, vol. 1, 1994, 55–60. 3, 29

[39] Oh, J., Guo, X., Lee, H., Lewis, R. L., AND Singh, S. Action-conditional video prediction using deep networks in atari games, in *Advances in Neural Information Processing Systems*, 2015, 2863–2871. 14

[40] Osband, I., Blundell, C., Pritzel, A., AND Van Roy, B. Deep exploration via bootstrapped dqn, in *Advances in Neural Information Processing Systems*, 2016, 4026–4034. 25, 26

[41] Osband, I., Aslanides, J., AND Cassirer, A. Randomized prior functions for deep reinforcement learning, in *Advances in Neural Information Processing Systems*, 2018, 8617–8629. 25–27

[42] Pearce, T., Zaki, M., Brintrup, A., AND Neel, A., Uncertainty in neural networks: Bayesian ensembling, *arXiv preprint arXiv:1810.05546*, 2018. 25, 26

[43] Peng, J., AND Williams, R. J., Efficient learning and planning within the dyna framework, *Adaptive Behavior*, vol. 1, no. 4, 437–454, 1993. 9

[44] Puterman, M. L. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014. 5

[45] Rumelhart, D. E., Hinton, G. E., AND Williams, R. J., Learning representations by back-propagating errors, *nature*, vol. 323, no. 6088, 533–536, 1986. 11

[46] Schaul, T., Quan, J., Antonoglou, I., AND Silver, D. Prioritized experience replay, in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. 2

[47] Silver, D., Sutton, R. S., AND Müller, M. Sample-based learning and search with permanent and transient memories, in *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, 968–975. 1

[48] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., *et al.*, A general reinforcement learning algorithm that masters chess, shogi, and go through self-play, *Science*, vol. 362, no. 6419, 1140–1144, 2018. 2

[49] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., AND Salakhutdinov, R., Dropout: A simple way to prevent neural networks from overfitting, *The Journal of Machine Learning Research*, vol. 15, no. 1, 1929–1958, 2014. 25

[50] Sutton, R. S., Learning to predict by the methods of temporal differences, *Machine learning*, vol. 3, no. 1, 9–44, 1988. 8

[51] ——, Dyna, an integrated architecture for learning, planning, and reacting, *ACM Sigart Bulletin*, vol. 2, no. 4, 160–163, 1991. 9

[52] ——, Generalization in reinforcement learning: Successful examples using sparse coarse coding, in *Advances in Neural Information Processing Systems*, 1996, 1038–1044. 37

[53] Sutton, R. S., AND Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018. 5, 8

[54] Sutton, R. S., Szepesvári, C., Geramifard, A., AND Bowling, M. H. Dyna-style planning with linear function approximation and prioritized sweeping, in *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, 2008, 528–536. 14

[55] Vapnik, V. Principles of risk minimization for learning theory, in *Advances in Neural Information Processing Systems*, 1992, 831–838. 15, 20

[56] Wan, Y., Zaheer, M., White, A., White, M., AND Sutton, R. S. Planning with expectation models, in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 2019, 3649–3655. DOI: 10.24963/ijcai.2019/506. 14

[57] Watkins, C. J.C. H. Learning from delayed rewards, PhD thesis, 1989. 8

Appendix A

Additional Results for the Regression Example

We now present results for additional configurations of the learning rate for the example regression problem discussed in Chapter 4.

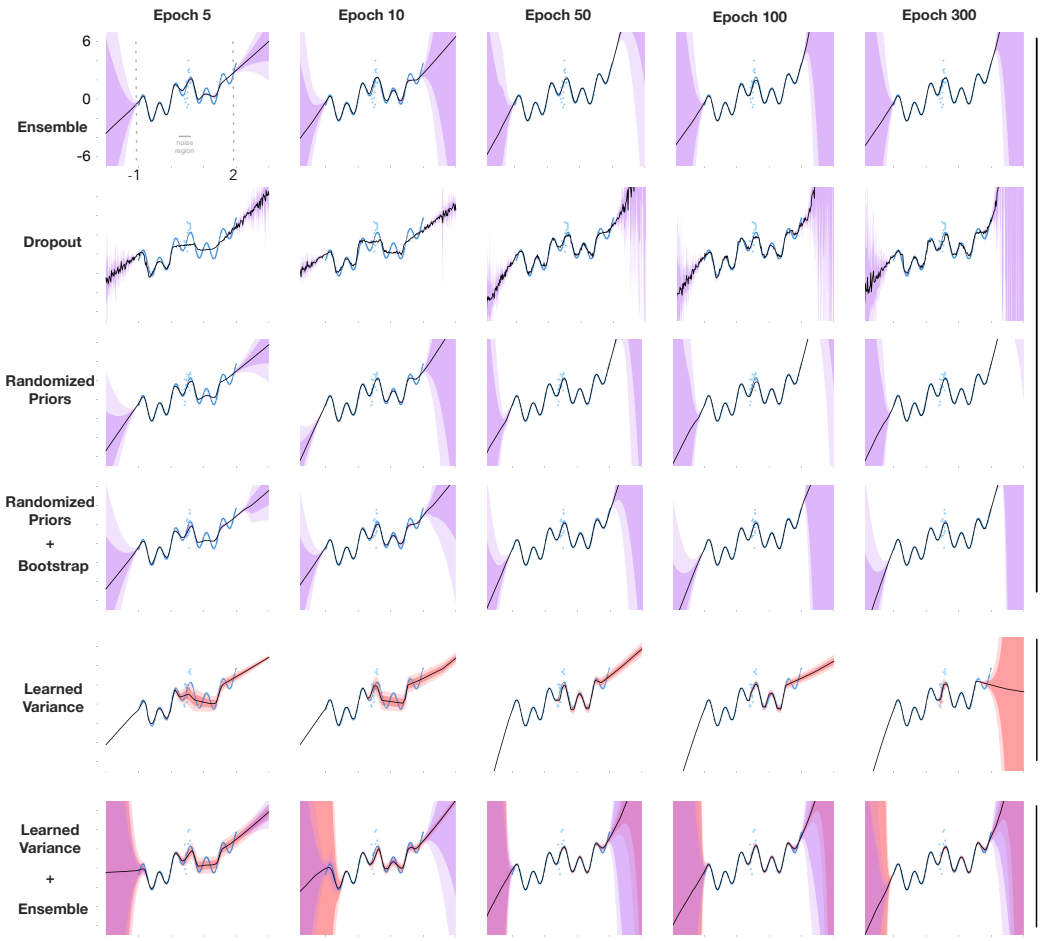

Figure A.1: Large Capacity Resultsfor learning rate 0.01. The network architecture consists of 3 hidden layers with 64 hidden units each. The ground truth function is in blue in all plots. Each row represents the mean predictions and uncertainty estimates of a particular modeling approach over the course of training. Learning rate is 0.01 for all methods

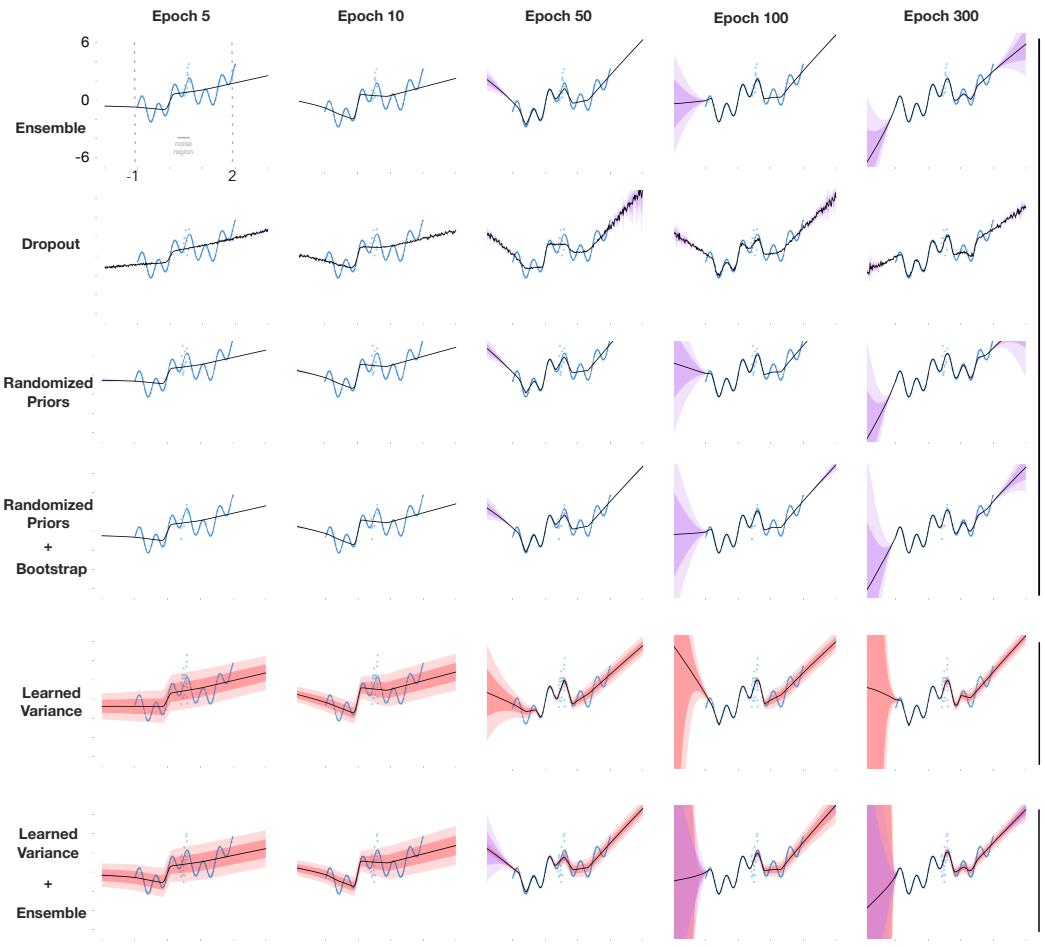

Figure A.2: Large Capacity Results for learning rate 0.0001. The network architecture consists of 3 hidden layers with 64 hidden units each. The ground truth function is in blue in all plots. Each row represents the mean predictions and uncertainty estimates of a particular modeling approach over the course of training. Learning rate is 0.0001 for all methods

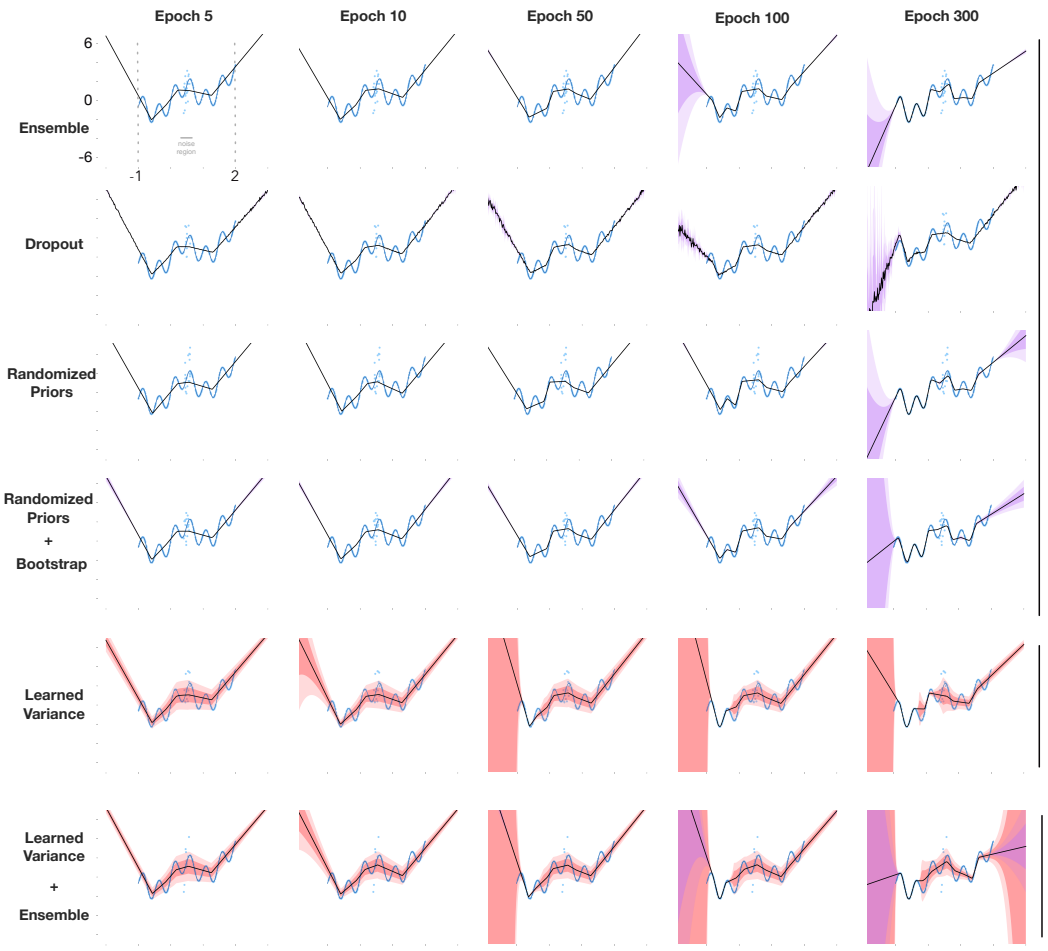

Figure A.3: Medium Capacity Results for learning rate 0.001. The network architecture consists of a single hidden layer with 2048 hidden units each. The ground truth function is in blue in all plots. Each row represents the mean predictions and uncertainty estimates of a particular modeling approach over the course of training. Learning rate is 0.001 for all methods

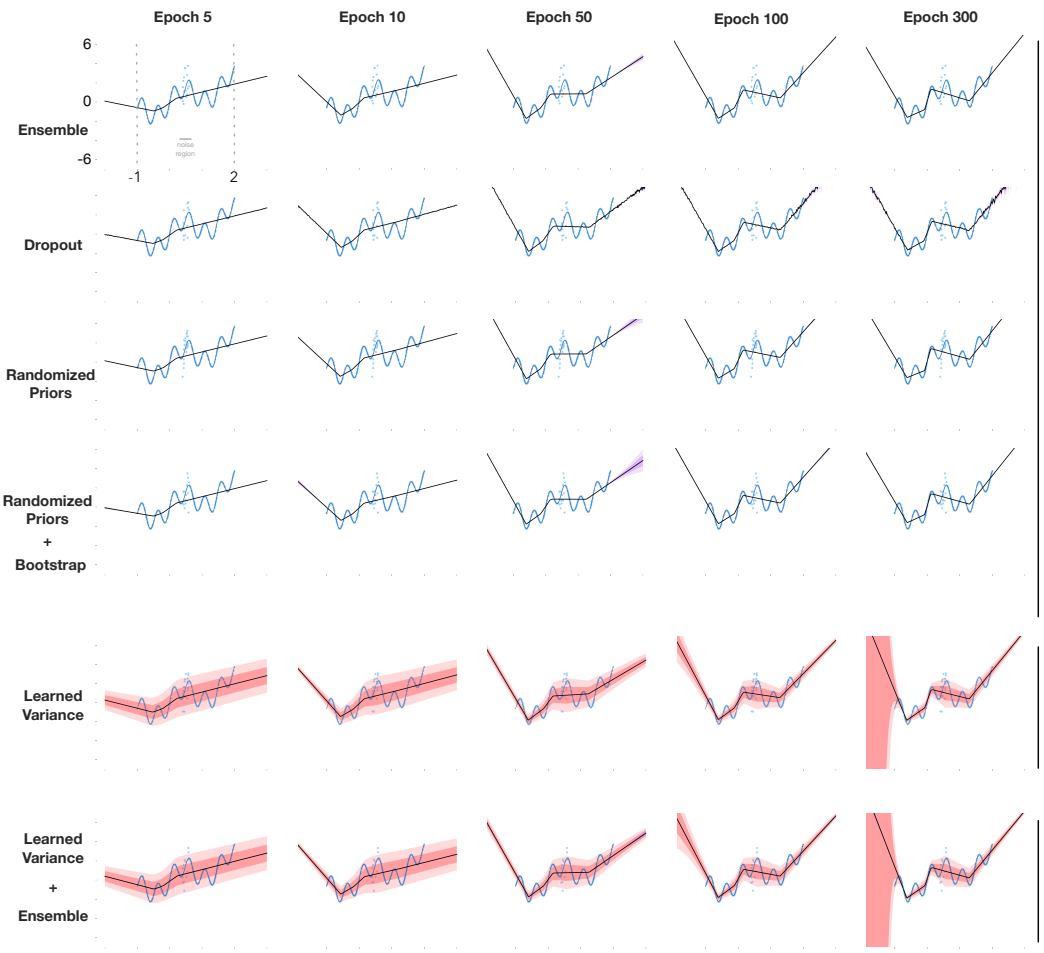

Figure A.4: Medium Capacity Results for learning rate 0.0001. The network architecture consists of a single hidden layer with 2048 hidden units each. The ground truth function is in blue in all plots. Each row represents the mean predictions and uncertainty estimates of a particular modeling approach over the course of training. Learning rate is 0.0001 for all methods

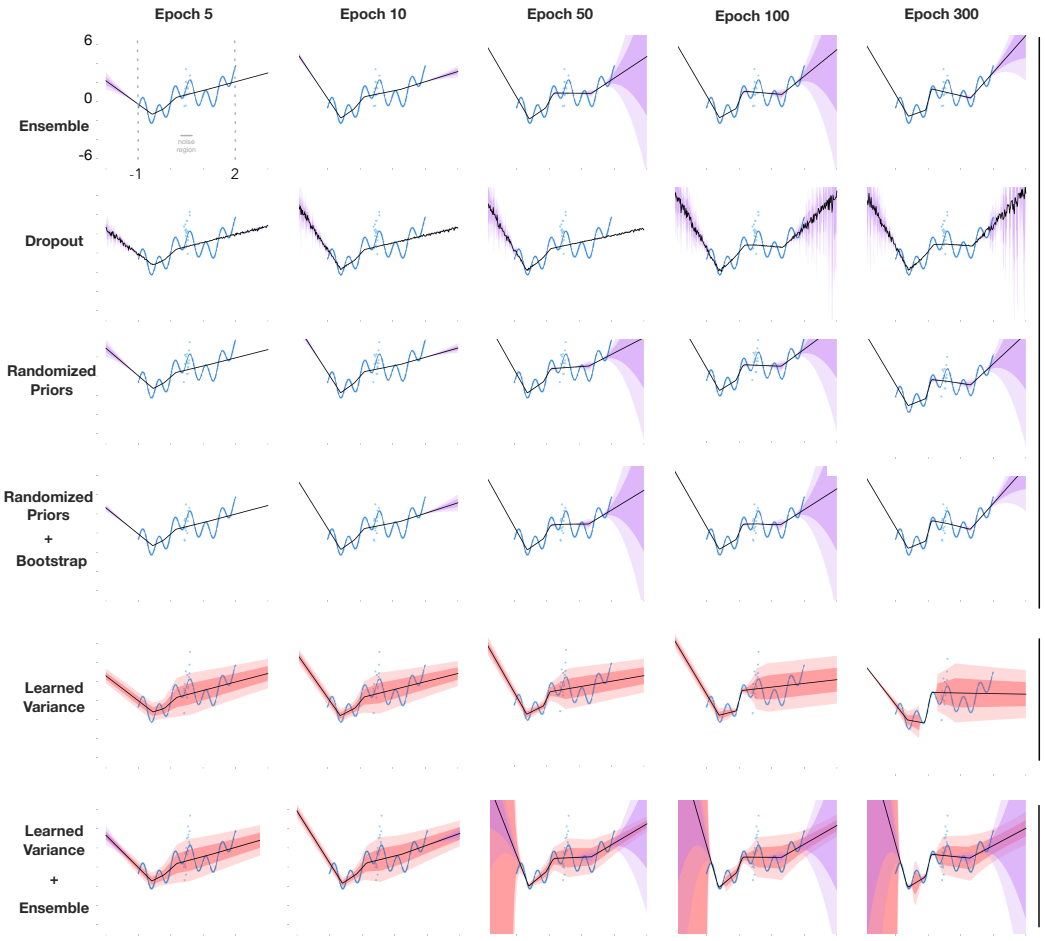

Figure A.5: Small Capacity Results for learning rate 0.001. The network architecture consists of a single hidden layer with 64 hidden units each. The ground truth function is in blue in all plots. Each row represents the mean predictions and uncertainty estimates of a particular modeling approach over the course of training. Learning rate is 0.001 for all methods

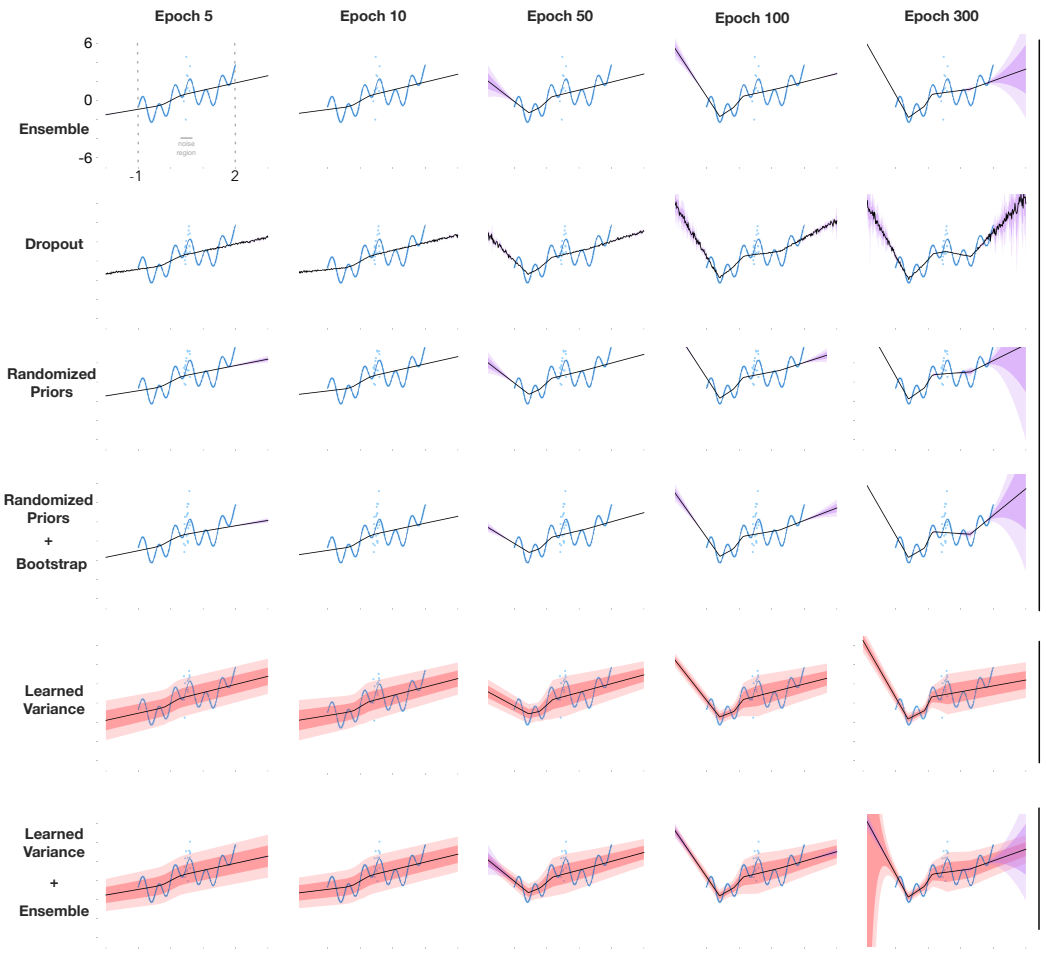

Figure A.6: Small Capacity Results for learning rate 0.0001. The network architecture consists of a single hidden layer with 64 hidden units each. The ground truth function is in blue in all plots. Each row represents the mean predictions and uncertainty estimates of a particular modeling approach over the course of training. Learning rate is 0.0001 for all methods

Appendix B

MVE Psuedocode

Algorithm 1 Model-based Value Expansion

Initialize replay memory D to capacity N
Initialize action-value function $Q_{\mathbf{w}}$
Initialize target action-value function $Q_{\mathbf{w}^-}$
Initialize the dynamics model f_θ

for episode=1, M **do**
  for t=1, T **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \arg\max_a Q_{\mathbf{w}}(\mathbf{x}(s_t), a)$
    Execute action a_t in the environment and observe reward r_{t+1} and feature $\mathbf{x}(s_{t+1})$
    Store transition $\big(\mathbf{x}(s_t), a_t, r_{t+1}, \mathbf{x}(s_{t+1})\big)$
    Sample a random mini-batch of transitions $\big(\mathbf{x}(s_j), a_j, r_{j+1}, \mathbf{x}(s_{j+1})\big)$ from D
    Simulate H-step trajectories starting from $\mathbf{x}(s_{j+1})$, with the model f_θ and a policy greedy w.r.t $Q_{\mathbf{w}}$, to obtain $\mathbf{x}(s_{j+1}), a_{j+1}, r_{j+2}, \mathbf{x}(s_{j+2}), ..., \mathbf{x}(s_{j+H}), a_{j+H}, r_{j+H+1}, \mathbf{x}(s_{j+H+1})$
    for k=1, H+1 **do**
      Compute multi-step targets: $U_h\big(\mathbf{x}(s_j), a_j\big) = \sum_{k=1}^{h} \gamma^{k-1} r_k + \gamma^h \max_{a \in \mathcal{A}} Q_{\mathbf{w}^-}(\mathbf{x}(s_h), a)$
    end for
    Average multi-step targets: $U_{avg}(s_j, a_j) = \frac{1}{H+1} \sum_{h=1}^{H+1} U_h(s_j, a_j)$
    Update action-values $Q_{\mathbf{w}}(s_j, a_j)$ for the sampled mini-batch towards $U_{avg}(s_j, a_j)$ using semi-gradient Q-learning update
    Every Z steps copy weights $\mathbf{w}$ to $\mathbf{w}^-$
    Sample a random mini-batch of transitions $\big(\mathbf{x}(s_j), a_j, r_{j+1}, \mathbf{x}(s_{j+1})\big)$ from D
    Learn model f_θ with the sampled transitions by minimizing the squared error using gradient descent
  end for
end for