HYPERDQN: A RANDOMIZED EXPLORATION METHOD FOR DEEP REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Randomized least-square value iteration (RLSVI) is a provably efficient exploration method. However, it is limited to the case where 1) a good feature is known in advance and 2) this feature is fixed during the training: if otherwise, RLSVI suffers an unbearable computational burden to obtain the posterior samples of the parameter in the Q-value function. In this work, we present a practical algorithm named *HyperDQN*, addressing these two issues under the context of deep reinforcement learning, where the feature changes over iterations. HyperDQN is built on two parametric models: in addition to a non-linear neural network (i.e., base model) that predicts Q-values, our method employs a probabilistic hypermodel (i.e., meta model), which outputs the parameter of the base model. When both models are jointly optimized under a specifically designed objective, three purposes can be achieved. First, the hypermodel can generate approximate posterior samples regarding the parameter of the Q-value function. As a result, diverse Q-value functions are sampled to select exploratory action sequences. This retains the punchline of RLSVI for efficient exploration. Second, a good feature is learned to approximate Q-value functions. This addresses limitation 1. Third, the posterior samples of the Q-value function can be obtained in a more efficient way than the existing methods, and the changing feature does not affect the efficiency. This deals with limitation 2. On the Atari 2600 suite, after 20M samples, *HyperDQN* achieves about $2\times$ improvements over (double) DQN, the advanced method Bootstrapped DQN, and the SOTA exploration bonus method OB2I. For another challenging task SuperMarioBros, HyperDQN outperforms baselines on 7 out of 9 games.

1 INTRODUCTION

Reinforcement learning (RL) (Sutton & Barto, 2018) involves an agent that interacts with an unknown environment to maximize cumulative reward. The trade-off between exploration and exploitation is a fundamental problem in RL (Kakade, 2003). On the one hand, the agent needs to explore highly uncertain states and actions, which may sacrifice immediate reward. On the other hand, in the long term, the agent should take the best-known action; however, this action may be sub-optimal due to partial information. To this end, sample-efficient RL agents should qualify the epistemic uncertainty (i.e., subjective uncertainty due to limited samples (Osband, 2016a)) to address the trade-off.

Bayesian approaches like Thompson sampling (Russo et al., 2018) provide a nice way of encoding the epistemic uncertainty by posterior distribution. For instance, randomized least-square value iteration (RLSVI)[1] (Osband et al., 2016b; 2019) is a celebrated Bayesian algorithm. Specifically, RLSVI takes three steps to address the exploration and exploitation trade-off. First, conditioned on observed data, RLSVI solves a Bayesian linear regression problem and updates the posterior distribution over (the parameter of) the optimal Q-value functions by the Bayes update rule. Second, RLSVI samples a specific Q-value function from the posterior distribution. Third, to collect more data, greedy actions are selected based on this Q-value function. In theory, the randomness in posterior sampling could yield positive bias, which boosts optimistic behaviors (Osband et al., 2019; Russo, 2019; Zanette et al., 2020). However, RLSVI is restricted to the following case.

- A good feature is known in advance. Over this feature, the Q-values can be approximated as a linear function.

[1]For readers who are not familiar with RLSVI, please refer to Appendix B.1 for a brief introduction.

- This feature is fixed so that the posterior distribution is easy to compute in an incremental update way (see Remark 2 in Appendix B.1 or (1.1) for an illustration).

Unfortunately, these two requirements do *not* hold in the deep RL scenario. The challenges of extending RLSVI to deep RL are listed below.

1) **If the provided feature is not good and fixed, RLSVI is rarely competent.** Under the deep RL setting, only a raw feature outputted by a randomly initialized neural network is available. Such an unpolished feature has limited representation power, so the Q-value function cannot be approximated accurately. Further, the mechanism of RLSVI only involves updating the model parameter, while the feature is left to be fixed (as a raw feature in this setting); see Line 12 in Algorithm 1 in Appendix. As a result, the empirical performance of RLSVI is often poor; see Appendix F.1 for the evidence.

2) **When we update the feature over iterations in deep RL, the computational complexity of RLSVI becomes unbearable.** As the data accumulates over iterations, RLSVI needs to repeatedly compute the feature covariance matrix to update the posterior distribution. With the fixed feature mapping, this process can be efficiently implemented in an incremental way (see (1.1), where x_K denotes the state-action pair (s_K, a_K) at iteration K and $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$ is a feature mapping.). However, when the feature mapping ϕ_K is changing over iterations, we need to repeatedly calculate the matrix Φ_K using all historical data as in (1.2) (e.g. in Atari, this calculation could involve more than 1M samples with dimension 512), and this process results in a huge computation burden.

$$\text{fixed } \phi: \quad \Phi_K = \Phi_{K-1} + \phi(x_K)\phi(x_K)^\top \quad \text{with } \Phi_0 = \mathbf{0}, \tag{1.1}$$

$$\text{changing } \phi_K: \quad \Phi_K := \sum_{\ell=1}^{K} \phi_K(x_\ell)\phi_K(x_\ell)^\top, \ \Phi_{K-1} := \sum_{\ell=1}^{K-1} \phi_{K-1}(x_\ell)\phi_{K-1}(x_\ell)^\top, \cdots \tag{1.2}$$

To address these two issues in deep RL, Bootstrapped DQN (BootDQN) (Osband et al., 2016a; 2018) takes a remarkable first step. To bypass the hurdle of issues 1 and 2, BootDQN simultaneously trains tens of ensembles (i.e., differently initialized neural networks) and views them as approximate posterior samples of Q-value functions. However, with limited computation resources, BootDQN typically uses ensembles with a strictly limited number of elements. As such, the quality of the learned posterior distribution could be poor (Lu & Van Roy, 2017).

In this work, we present a principled approach named *HyperDQN* that addresses the above issues under the context of deep RL. Specifically, HyperDQN is built on two parametric models: in addition to a non-linear neural network (i.e., base model) that predicts Q-values, it employs a hypermodel (Dwaracherla et al., 2020) (i.e., meta model). This hypermodel maps a vector z (from the Gaussian distribution) to a parameter instance of the Q-value function. As shown in Figure 1, this hypermodel aims to serve as a proxy for generating the posterior samples of θ_{predict}. To achieve this goal, both models are jointly optimized by a specially designed temporal difference (TD) objective (see (4.2)).

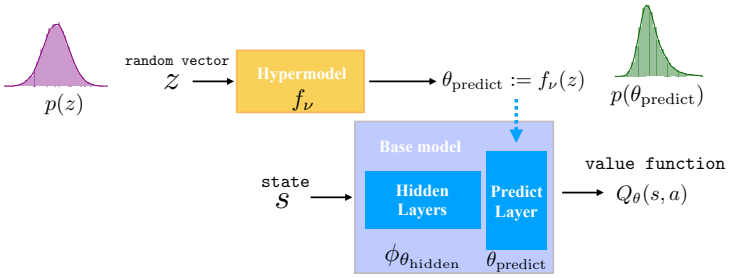


Figure 1: Illustration for HyperDQN .

After optimization, our approach has the following essential merits.

- First, we prove that the optimized hypermodel can approximate the posterior distribution under the Bayesian linear regression case (see Theorem 1). Thus, diverse Q-value functions can be sampled to select exploratory action sequences. This retains the punchline of RLSVI for efficient exploration. We remark that this theoretical guarantee is missing in (Dwaracherla et al., 2020).
- Second, the feature mapping $\phi_{\theta_{\text{hidden}}}(\cdot)$ is updated during the training. This deals with the feature learning problem of RLSVI (i.e., issue 1).

- Third, the posterior samples of θ_{predict} are obtained without involving (1.2). Specifically, the hypermodel could directly output an approximate posterior sample by taking z as its input. This addresses the computational problem of RLSVI when the feature is changing (i.e., issue 2).
- Finally, compared with the finite ensembles in BootDQN, our approach measures the parameter uncertainty through "infinite" ensembles. This is because the hypermodel could generate "infinite" parameter instances by mapping z from a continuous distribution. Consequently, our approach could have a better posterior approximation with the same computation resources. As such, our method is more computationally efficient to achieve a near-optimal performance (see Appendix E.3).

To evaluate the efficiency of algorithms, we firstly consider the Atari 2600 suite (Bellemare et al., 2013). We assess the efficiency in terms of the median performance over all 56 tasks in the suite provided 20M training frames. The empirical result suggests HyperDQN achieves about $2\times$ improvements over its baseline version Double DQN (van Hasselt et al., 2016), the advanced randomized exploration method BootDQN, and the SOTA exploration bonus method OB2I (Bai et al., 2021).

Despite the wide coverage of Atari tasks, we find the widely used trick *reward clipping* (i.e., map the game score to $\{-1, 0, +1\}$) could preclude a meaningful comparison. Surprisingly, there are 15 out of 56 tasks where the training and evaluation performance are weakly correlated or even uncorrelated (see Remark 6 in Appendix)! To this end, we consider an alternative platform: SuperMarioBros (Kauten, 2018). It provides 9 video games, covering both sparse and dense reward tasks[2]. With the same parameter configuration, HyperDQN outperforms baselines on 7 out of 9 games.

2 Background

Markov Decision Process. In the standard reinforcement learning (RL) framework (Sutton & Barto, 2018), a learning agent interacts with an Markov Decision Process (MDP) to improve its performance via maximizing cumulative reward. The sequential decision process is characterized as follows: at each timestep t, the agent receives a state s_t from the environment and selects an action a_t from its policy $\pi(a|s) = \Pr\{a = a_t | s = s_t\}$; this decision is sent back to the environment, and the environment gives a reward signal $r(s_t, a_t)$ and transits to the next state s_{t+1} based on the state transition probability $p^a_{s,s'} = \Pr\{s' = s_{t+1} | s = s_t, a = a_t\}$. The main target of RL is to maximize the (expected) discounted return $\mathbb{E}\big[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \big| s_0 \sim \rho(\cdot)\big]$, where $\rho(\cdot)$ is initial state distribution and $\gamma \in (0, 1)$ is a discount factor.

Deep Q-Networks (DQN). In Deep Q-Networks (DQN) (Mnih et al., 2015), it employs a neural network to approximate the Q-value function, which is defined as $Q^\pi(s, a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s, a_0 = a]$. In particular, a *temporal difference* (TD) based objective is applied:

$$\min_{\theta} \sum_{(s,a,r,s') \sim \mathcal{D}} \big(r(s, a) + \gamma \max_{a'} Q_{\bar{\theta}}(s', a') - Q_\theta(s, a)\big)^2, \tag{2.1}$$

where $\mathcal{D}$ is the experience replay buffer, Q_θ is a prediction network parameterized by θ, and $Q_{\bar{\theta}}$ is the so-called target network, which is a delayed copy of Q_θ for stable training.

3 Related Work

Epistemic uncertainty qualification. It is justified that dithering strategies like ϵ-greedy (Mnih et al., 2015) are inefficient (Osband et al., 2019). This is intuitive since they do not have any epistemic uncertainty measure and hence cannot "write-off" sub-optimal actions after experimentation. Importantly, epistemic uncertainty reflects the confidence about the unknown environment in online decision-making (Osband, 2016a). For tabular MDPs with the one-hot feature, "count" provides an epistemic uncertainty measure (Brafman & Tennenholtz, 2002; Jaksch et al., 2010; Azar et al., 2017; Jin et al., 2018) and "covariance" serves as the counterpart for linear MDPs (Jin et al., 2020; Cai et al., 2020). Currently, we do not have a perfect epistemic uncertainty qualification tool for deep RL, where a good feature is unknown in advance. As argued in (Osband et al., 2018), approaches like dropout (Srivastava et al., 2014) and distributional operator (Bellemare et al., 2017) are not suitable since they are designed for risk estimation. In this paper, we focus on the extension of hypermodel (Dwaracherla et al., 2020), which could capture the epistemic uncertainty for bandit tasks.

[2]The raw score range of SuperMarioBros is well controlled so that it suffices to rescale the raw score rather than clipping. In this way, the training performance and evaluation performance are much correlated.

OFU-based exploration. Based on the mentioned uncertainty qualification, optimism in the face of uncertainty (OFU) based methods (Jaksch et al., 2010; Azar et al., 2017; Jin et al., 2018) construct upper confidence bound (UCB) to direct exploration. These theoretical works have inspired many empirical studies that encourage exploration by adding "reward/exploration bonus" to mimic UCB (Stadie et al., 2015; Pathak et al., 2017; Tang et al., 2017; Burda et al., 2019a;b). The challenges in this direction include: how to obtain task-relevant reward bonus (O'Donoghue et al., 2018)? how to get an optimistic initialization with neural network (Rashid et al., 2020)? how to properly back-propagate the uncertainty over periods to induce temporally extended behaviors (Bai et al., 2021)? Though some of SOTA exploration bonus methods have achieved superior performance on *some* hard exploration tasks, (Taïga et al., 2020) report that these algorithms do not provide meaningful gains over the ϵ-greedy scheme on the *whole* Atari 2600 suite.

TS-based exploration. On the other hand, Thompson sampling (TS) based methods, which originate in the bandit literature (see (Russo et al., 2018) and reference therein), design the exploration strategy in a Bayesian way (Osband et al., 2013; Osband & Van Roy, 2017). Randomized least-square value iteration (RLSVI) provides a promising direction (Osband et al., 2016b; 2019). As we mentioned in Section 1, this method, however, can only be applied when a good feature is known and fixed during the training. Following RLSVI, Osband et al. (2016a) develop a practical algorithm called Bootstrapped DQN, which uses finite ensembles to generate the randomized value functions. As discussed earlier, the main bottleneck of Bootstrapped DQN is its computation complexity: it requires lots of ensembles to obtain an accurate approximation of the posterior samples (Lu & Van Roy, 2017). Besides, (Azizzadenesheli et al., 2018) propose a direct extension of RLSVI, which is also computationally inefficient and suffers the feature learning issue.

Our method is closely related to NoisyNet (Fortunato et al., 2018) as both methods inject noise to the parameter. However, we cannot view NoisyNet as an extension of RLSVI. The main reason is that NoisyNet is not ensured to approximate the posterior distribution as stated in (Fortunato et al., 2018). We discuss other algorithmic differences and provide the numerical comparison in Appendix F.4.

4 Methodology

4.1 Architecture Design

In this part, we explain the architecture design of HyperDQN shown in Figure 1. First, the hypermodel $f_\nu(\cdot)$ maps a N_z-dimension random vector $z \sim p(z)$ (e.g., a Gaussian distribution) to a specific d-dimension parameter $\theta := f_\nu(z)$. For example, if z follows an isotropic Gaussian distribution $\mathcal{N}(0, I)$ and the hypermodel $f_\nu(z) = \nu_w^\top z + \nu_b$ is a linear model, then $\theta := f_\nu(z) \sim \mathcal{N}(\nu_b, \nu_w^\top \nu_w)$, where $\nu_w \in \mathbb{R}^{N_z \times d}$ and $\nu_b \in \mathbb{R}^d$ are the weight and bias of the linear hypermodel with $\nu = (\nu_w, \nu_b)$. In general, the hypermodel could be a non-linear mapping so that $f_\nu(z)$ could follow an arbitrary distribution. In this paper, unless stated we only consider the linear hypermodel. This is because the hypermodel would be applied to model the posterior distribution of a linear function like RLSVI does and a linear hypermodel has such a representation power to approximate the posterior distribution.

Second, the base model (i.e., a deep neural network) operates in a standard way: it takes a state s as its input and outputs $Q_\theta(s, a) \in \mathbb{R}$ for some action a. Concretely, the base model employs a feature extractor $\phi_{\theta_{\text{hidden}}} : \mathcal{S} \to \mathbb{R}^d$ (i.e., the hidden layers in Figure 1) to learn a good representation. Then, the base model outputs $Q(s, a)$ by $Q(s, \cdot) = \theta_{\text{predict}}^\top \phi_{\theta_{\text{hidden}}}(s)$, where $\theta_{\text{predict}} \in \mathbb{R}^{d \times N_a}$ is the parameter of the prediction layer (i.e., the last layer of the base model in Figure 1). By our formulation, θ_{predict} is also the output of the hypermodel. That is, $\theta_{\text{predict}} = f_\nu(z) + f_{\nu_{\text{prior}}}(z)$, where $f_{\nu_{\text{prior}}}(z)$ is a fixed prior hypermodel; see the discussion below. Numerically, we fix $v_{\text{prior}} = v_0$, where v_0 is a random initialization. In the sequel, we will introduce the difference between the hypermodel in Figure 1 and that in Dwaracherla et al. (2020).

4.2 Differences with Dwaracherla et al. (2020): How the Direct Extension Fails

Readers may notice that our architecture design is different from the original one in (Dwaracherla et al., 2020). Concretely, we only apply the hypermodel for the *last* layer of the base model but in (Dwaracherla et al., 2020) the hypermodel is applied for *all* layers of the base model. This modification is motivated by the trainability issue: the training will fail if otherwise (see Appendix F.2 for the numerical support). To understand why, recall that parameter initialization matters for successfully training a deep neural network (Glorot & Bengio, 2010; Goodfellow et al., 2016). For

instance, the famous LeCun initialization rule (LeCun et al., 1998) suggests that we should initialize the parameter of i-th layer by sampling from the Gaussian distribution $\mathcal{N}(0, \sigma^2)$ with $\sigma = 1/\sqrt{d_{i-1}}$, where d_{i-1} is the width of $(i-1)$-th layer. As a result, the input signal and the output signal have the same order ℓ_2-norm after processing. Now, if we use the architecture in (Dwaracherla et al., 2020) with this technique, the output of the hypermodel is expected to lie between -1 and 1 (up to constants). This implies that the parameter is in $[-1, 1]$ for each layer of the base model, which disobeys the principle $1/\sqrt{d_{i-1}}$. Consequently, the input signal amplifies over layers and the gradient explodes (Sun, 2019). In fact, Dwaracherla et al. (2020) only use a two-layer base model with the width of 10 for bandit tasks so that the trainability issue is not severe in their applications. But for deep RL, this training issue is severe and we bypass this challenge by the proposed architecture shown in Figure 1, in which the hidden layers of the base model are initialized with common techniques and the last layer could be properly initialized by normalizing the output of the hypermodel. This deals with the parameter initialization issue. Fortunately, this simple architecture still retains the main ingredient of RLSVI (i.e., capturing the posterior distribution over a linear prediction function). We refer readers to Appendix F.2 for more discussion.

4.3 Training Objective

In this part, we introduce the objective function. For the sake of better presentation, let us first consider a regression task. Let $x \in \mathbb{R}^d$ be the feature and $y \in \mathbb{R}$ be the label, the objective function developed in (Dwaracherla et al., 2020) for training the hypermodel is

$$L(\nu; \mathcal{D}) = \int_z p(z) \Bigg[\sum_{(x,y,\xi)\in\mathcal{D}} \big(y + \underbrace{\sigma_\omega z^\top \xi}_{(a)} - \underbrace{(g_{f_{\nu_{\text{prior}}}(z)}(x) + g_{f_\nu(z)}(x))}_{(b)}\big)^2 + \underbrace{\frac{\sigma_\omega^2}{\sigma_p^2} \|f_\nu(z)\|^2}_{(c)} \Bigg] (\mathrm{d}z), \tag{4.1}$$

where ξ is a random vector independently sampled from the unit hypersphere, paired with each (x, y) in the dataset $\mathcal{D} = \{(x_i, y_i, \xi_i) : i = 1, \ldots, |\mathcal{D}|\}$; $\sigma_\omega > 0$ is the noise scale and $\sigma_p > 0$ is the regularization scale. Next, we briefly explain three key parts in (4.1). First, (a) is an artificial noise term exerted on the label y, which is introduced for pure technical purpose. We defer more discussion of (a) to Remark 1. Second, (b) contains two base models denoted by g. Concretely, $g_{f_{\nu_{\text{prior}}}(z)}(\cdot)$ is a *prior* model parameterized by the output of a fixed hypermodel $f_{\nu_{\text{prior}}}$. This term stores the prior information. At the same time, $g_{f_\nu(z)}(\cdot)$ is a *differential* model linked by the output of a trainable hypermodel f_ν. We remark that term (b) is called "additive prior models" in (Osband et al., 2018; Dwaracherla et al., 2020); see (Dwaracherla et al., 2020, Section 2.5) for a detailed discussion. Finally, (c) is a regularization term, which is an essential design in Bayesian learning.

Now we explain why (4.1) is a good objective to generate approximate posterior samples. As a "sanity check", we provide an analysis under the linear case.

Assumption 1. Suppose the underlying data generation follows $y = x^\top \theta^\star + \omega^\star$, $\omega^\star \sim \mathcal{N}(0, \sigma_\omega^2)$ and the prior distribution over $\theta^\star$ is $\mathcal{N}(\bar{\theta}_p, \sigma_p^2 I)$. Furthermore, assume the base model is linear, i.e., $g_{f_\nu(z)}(x) = x^\top f_\nu(z)$ and $g_{f_{\nu_{\text{prior}}}(z)}(x) = x^\top f_{\nu_{\text{prior}}}(z)$. Moreover, assume the hypermodel is also linear, i.e., $f_\nu(z) = \nu_w^\top z + \nu_b$ and $f_{\nu_{\text{prior}}}(z) = \nu_w^{\text{prior}\top} z + \nu_b^{\text{prior}}$; $\nu = (\nu_w, \nu_b)$ and $\nu_{\text{prior}} = (\nu_w^{\text{prior}}, \nu_b^{\text{prior}})$.

Based on Assumption 1, the posterior distribution over $\theta^\star$ is $\mathcal{N}(\mathbb{E}[\theta^\star \mid \mathcal{D}], \mathrm{Cov}[\theta^\star \mid \mathcal{D}])$ with

$$\mathbb{E}[\theta^\star \mid \mathcal{D}] = \left(\frac{1}{\sigma_\omega^2} X^\top X + \frac{1}{\sigma_p^2} I\right)^{-1} \left(\frac{1}{\sigma_\omega^2} X^\top Y + \frac{1}{\sigma_p^2} \bar{\theta}_p\right), \mathrm{Cov}[\theta^\star \mid \mathcal{D}] = \left(\frac{1}{\sigma_\omega^2} X^\top X + \frac{1}{\sigma_p^2} I\right)^{-1},$$

where $X \in \mathbb{R}^{|\mathcal{D}| \times d}$ and $Y \in \mathbb{R}^{|\mathcal{D}|}$ are concatenation of x_i and y_i, respectively. Also, we have that $g_{f_{\nu_{\text{prior}}}(z)}(x) + g_{f_\nu(z)}(x) = g_{f_{\nu_{\text{prior}}}(z) + f_\nu(z)}(x) = g_\theta(x) = x^\top \theta$, where $\theta := f_{\nu_{\text{prior}}}(z) + f_\nu(z)$. Next, we show that $\theta \sim \mathcal{N}(\nu_b^{\text{prior}} + \nu_b, (\nu_w^{\text{prior}} + \nu_w)^\top (\nu_w^{\text{prior}} + \nu_w))$ is an approximate posterior sample.

Theorem 1 (Informal)**.** Under Assumption 1, set $\nu_w^{\text{prior}} = \sigma_p I$ and $\nu_b^{\text{prior}} = \bar{\theta}_p$. Let $\nu^\star = (\nu_w^\star, \nu_b^\star)$ be the optimal solution of (4.1) conditioned on specific realizations of ξ , then $\theta := f_{\nu_{\text{prior}}}(z) + f_{\nu^\star}(z) \sim \mathcal{N}(\nu_b^{\text{prior}} + \nu_b^\star, (\nu_w^{\text{prior}} + \nu_w^\star)^\top (\nu_w^{\text{prior}} + \nu_w^\star))$ satisfies

$$\nu_b^{\text{prior}} + \nu_b^\star = \mathbb{E}[\theta^\star \mid \mathcal{D}], \quad (\nu_w^{\text{prior}} + \nu_w^\star)^\top (\nu_w^{\text{prior}} + \nu_w^\star) = \mathrm{Cov}[\theta^\star \mid \mathcal{D}] + \mathtt{err}(\xi).$$

Furthermore, the error term satisfies $\mathbb{E}_\xi[\mathtt{err}(\xi)] = 0$.

Theorem 1 states that the optimized hypermodel can approximate the true posterior distribution. The formal statement and proof are given in Appendix C.3. To put our result in the literature, we remark that Dwaracherla et al. (2020) prove that a linear hypermodel has sufficient representation power to approximate any distribution (over functions) so it is unnecessary to use a non-linear one. However, this guarantee is unrelated to the objective (4.1): Dwaracherla et al. (2020) do not elaborate on whether the linear hypermodel can approximate the posterior distribution *after being trained under objective (4.1)*. This question is affirmatively answered by Theorem 1.

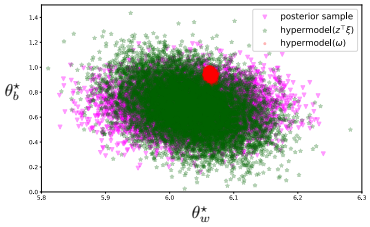


Figure 2: Visualization of true posterior samples and learned posterior samples.

Remark 1. Similar to (4.1), in one of versions of RLSVI (Osband et al., 2019), it also adds an artificial noise ω on each label y, but such a noise ω is independently sampled from $\mathcal{N}(0, \sigma_\omega^2)$ (see Appendix C.1). It is worth noting that Theorem 1 does *not* hold if we use such a noise ω rather than $z^\top \xi$ in (4.1). This is because ω will introduce exogenous random sources into the objective function, which jeopardizes the learning goal of mapping an index z to a posterior sample θ; see Appendix C.4 for a formal argument. To get a grasp of the fundamental issue involved here, consider a toy 2-dimensional Bayesian linear regression problem (Assumption 1) with $\theta^\star = (\theta_w^\star, \theta_b^\star)$; see Appendix D.4 for experiment details. After optimizing (4.1), we visualize the posterior samples generated by the linear hypermodel in Figure 2. In particular, `hypermodel(`$z^\top\xi$`)` (in green) and `hypermodel(`ω`)` (in red) correspond to learned samples via solving (4.1) with a z-dependent noise $z^\top \xi$ and an independent Gaussian noise ω, respectively. The variant `posterior sample` (in purple) is obtain by the true posterior distribution $\mathcal{N}(\mathbb{E}[\theta^\star \mid \mathcal{D}], \mathrm{Cov}[\theta^\star \mid \mathcal{D}])$. We observe that the z-dependent noise is indispensable for the hypermodel to approximate the posterior distribution.

To address the limitations of RLSVI for deep RL tasks, we first employ a feature mapping (parameterized by θ_{hidden}) and approximate the Q-value function by a linear function (parameterized by θ_{predict}) over the learned feature. Furthermore, the parameter of this linear function is modeled by the hypermodel (i.e., $\theta_{\text{predict}} = f_{\nu_{\text{prior}}}(z) + f_\nu(z)$) as discussed in Section 4.1. To this end, we extend (4.1) and implement the following optimization problem:

$$\min_{\nu, \theta_{\text{hidden}}} \int_z p(z) \left[\sum_{(s,a,r,\xi,s') \in \mathcal{D}} \left(Q_{\text{target}}(s', z) + \sigma_\omega z^\top \xi - Q_{\text{prediction}}(s, a, z) \right)^2 + \frac{\sigma_\omega^2}{\sigma_p^2} \left\| f_\nu(z) \right\|^2 \right] (\mathrm{d}z), \tag{4.2}$$

where

$$\begin{aligned} Q_{\text{prediction}}(s, a, z) &= Q_{\theta_{\text{prior}}, f_{\nu_{\text{prior}}}(z)}(s, a) + Q_{\theta_{\text{hidden}}, f_\nu(z)}(s, a), \\ Q_{\text{target}}(s', z) &= r + \gamma \max_{a'} \left[Q_{\theta_{\text{prior}}, f_{\nu_{\text{prior}}}(z)}(s', a') + Q_{\bar{\theta}_{\text{hidden}}, f_{\bar{\nu}}(z)}(s', a') \right]. \end{aligned} \tag{4.3}$$

Similar to the formulation in the regression problem, the Q-value function is the sum of the prior $Q_{\theta_{\text{prior}}, f_{\nu_{\text{prior}}}(z)}$ and the differential $Q_{\theta_{\text{hidden}}, f_\nu(z)}$. Following the target network in DQN, $(\bar{\theta}_{\text{hidden}}, \bar{\nu})$ is the delayed copy of $(\theta_{\text{hidden}}, \nu)$. Compared with the TD loss in (2.1), there is an additional noise term $\sigma_\omega z^\top \xi$ for Bayesian learning and the prediction layer of the Q-value function is modeled as a probabilistic layer by the hypermodel. In experiments, we use the average of finite samples of z to approximate (4.2); see the empirical loss function in (B.6) in Appendix. We choose the stochastic gradient descent algorithm to optimize such constructed empirical loss function.

4.4 The Algorithm

We outline the proposed method in Algorithm 2 in Appendix. At the beginning of each episode, a random vector z is sampled to obtain the Q-value function, which is further used for interactions with environments. Subsequently, the randomized temporal difference objective in (4.2) is optimized to train the hypermodel and the hidden layers based on the collected data.

Next, we interpret HyperDQN from an algorithmic level introduced in (Osband et al., 2018). In particular, HyperDQN achieves the desiderata:

- **Task-relevant feature.** A task-specific feature can be obtained by the end-to-end training in (4.2). Such a feature is good in the sense that it is optimized to approximate multiple value functions as

different invocations of z yield several individual TD loss functions. As discussed previously, this feature learning procedure is beyond the scope of RLSVI.

- **Multi-step uncertainty.** After the initialization, all state-action pairs tend to be uniformly uncertain by random features. During the interaction, the agent updates its belief and the epistemic uncertainty could temporally propagate by the TD-style objective in (4.2) to induce "far-sighted" behaviors. This is crucial for the deep exploration behaviors (see Figure 15b in Appendix E.3 for the visualization of the multi-step uncertainty).
- **Commitment.**[3] The agent executes action sequences that span multiple periods by obeying its intent. In each episode, HyperDQN samples a specific value function and takes greedy actions according to this value function until the episode terminates. Unlike BootDQN (Osband et al., 2016a) [4] and OFU-based algorithms (Taïga et al., 2020; Rashid et al., 2020; Bai et al., 2021)[5], HyperDQN does *not* combine ϵ-greedy as the perturbation in ϵ-greedy ruins "far-sighted" behaviors generated by the original algorithm.

HyperDQN is closely related to BootDQN, which adopts finite ensembles to generate randomized value functions. When the computation resources are limited, the quality of generated posterior samples by BootDQN is poor (Lu & Van Roy, 2017). However, our approach has the advantage of measuring the uncertainty through "infinite" ensembles. As such, our method tends to be more efficient; see the evidence in Table 4 in Appendix E. As a side note in (Dwaracherla et al., 2020), if the random index z is repeatedly sampled from a finite set, the hypermodel degenerates to finite ensembles. As such, BootDQN can be viewed as a special case of HyperDQN. Following (Osband et al., 2018), we summarize the characteristics of BootDQN and HyperDQN in Table 1.

Table 1: Important issues in posterior approximations for deep RL. A green tick indicates a satisfying result, a red cross implies an undesirable result and a yellow circle means something in between.

	Task-relevant feature	Multi-step uncertainty	Commitment	Cheap computation
BootDQN (Osband et al., 2018)	✓	✓	●	✗
HyperDQN	✓	✓	✓	●
RLSVI (Osband et al., 2016b)	✗	✓	✓	✗✗

5 Experiments

In this section, we present numerical experiments to validate the efficiency of the proposed method. Experiment details can be found in Appendix D.

5.1 Atari 2600 Suite

Our first experiment is on the Arcade Learning Environment (Bellemare et al., 2013), which provides a platform to assess the general competence. As suggested by (Taïga et al., 2020), we evaluate algorithms across *all* 56 in the Atari 2600 suite. The training and evaluation procedures (e.g., observation preprocessing and reward clipping) follows (Mnih et al., 2015) and (van Hasselt et al., 2016). Unless stated, reported results are based on the evaluation performance from each training epoch. Experiments are run with 3 random seeds.

We measure the sample efficiency in terms of the human-normalized score during the interaction. In particular, the raw score of each game is normalized so that 0% corresponds to a random agent and 100% to a human expert. The final performance is aggregated over 56 games by the robust statistics *median*, which is used in (Hessel et al., 2018; O'Donoghue et al., 2018; Taïga et al., 2020). By this metric, Taïga et al. (2020) point out that advanced exploration bonus methods do not provide meaningful gains over the simple ϵ-greedy scheme and conclude that the advances in some hard exploration tasks are due to the specific architecture design.

We consider the advanced exploration method BootDQN and the "SOTA"[6] OFU-based algorithm OB2I (Bai et al., 2021). We do not involve the Bayesian DQN (Azizzadenesheli et al., 2018) since its

[3] This concept is originally defined in the context of concurrent RL (Dimakopoulou & Roy, 2018). We borrow this concept to help discuss the combination with ϵ-greedy under the standard RL framework.

[4] BootDQN uses ϵ-greedy for complicated tasks like Atari; see (Osband et al., 2016a, Appendix D.1).

[5] We conjecture this is because of imperfect imitation of theoretical algorithms; see Appendix F.3.

[6] "SOTA" means OB2I is a strong baseline with the 20M frames training budget as claimed in (Bai et al., 2021).

performance is significantly inferior to OB2I as shown in (Bai et al., 2021) and it is not evaluated in the whole Atari suite in the original paper. The baseline is Double DQN (van Hasselt et al., 2016) since BootDQN and our method are built on it.

The learning curve with the 20M frames training budget is displayed in Figure 3. Note that this configuration is commonly used in (Lee et al., 2019; Rashid et al., 2020; Bai et al., 2021). This is because algorithms can achieve satisfactory performance on many tasks with 20M frames, while 200M frames training suffers heavy financial burden and time cost[7]. We see that BootDQN, OB2I, and HyperDQN could quickly explore at the initial stage and outperform the baseline Double DQN[8]. However, only HyperDQN achieves substantial (i.e., about $2\times$) improvement over Double DQN after 20M frames. We notice the performance of OB2I is poor at the final stage[9], which is partially because the exploration bonus may guide the agent in a wrong direction.

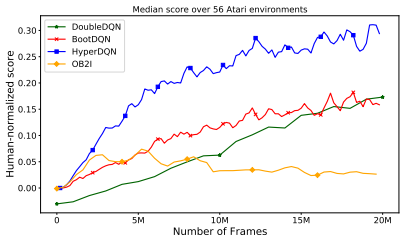


Figure 3: Human-normalized score over 56 environments in Atari 2600 suite.

We also consider the metric of the best performance in hindsight, which is used in (van Hasselt et al., 2016; Hessel et al., 2018; Bai et al., 2021). The exact results are reported in Table 5 in Appendix. Under this metric, HyperDQN is the best on 31 out of 56 games. According to the taxonomy in (Bellemare et al., 2016), the relative improvements of HyperDQN on "easy exploration" and "hard exploration" environments are also visualized in Figure 11 and Figure 12 in Appendix E . We see that HyperDQN could perform well on both easy exploration and hard exploration tasks. However, HyperDQN does not work for Montezuma's Revenge, in which the reward is very sparse so that there is limited feedback for feature selection. For the same reason, randomized exploration methods in (Osband et al., 2016a; 2018; Fortunato et al., 2018) do not work well on this task, either.

Since the score range over tasks in the Atari suite varies a lot, people typically use the clipped score (i.e., clipping the one-step score to $\{-1, 0, +1\}$) for training and the raw score for evaluation. We observe this trick makes the training and evaluation performance weakly correlated or even uncorrelated for 15 out of 56 tasks; see the evidence and Remark 6 in Appendix E. Briefly, an intelligent agent may obtain lots of $+1$ training reward. But the evaluation performance could be bad because each training reward corresponds to a low raw game score. To this end, we consider an alternative test platform: SuperMarioBros (Kauten, 2018), which is introduced as follows.

5.2 SuperMarioBros

In this part, we evaluate algorithms on the SuperMarioBros suite (Kauten, 2018). Since the raw game scores of SuperMarioBros tasks are well controlled, we use $0.01 \times$ `raw score` as the training reward. In this way, the training and evaluation performance are quite correlated. Other environment preprocessing and algorithm parameters are almost the same as Atari 2600 suite. We note that SuperMarioBros-1-3 and SuperMarioBros-2-2 are two hard exploration tasks due to the long planning horizon and sparse reward. Experiments are run with 3 random seeds.

We visualize the raw episode score in Figure 4. We observe that HyperDQN outperforms all baselines for 7 out of 9 tasks; see exact numbers in Table 3 in Appendix. Interestingly, Double DQN performs well on SuperMarioBros-2-1-v1. By watching generated action sequences, we know that Double DQN unintentionally finds the internal "bugs" of this game, i.e., playing the tortoise forever (see the video recorder at https://sites.google.com/view/hyperdqn). Thus, Double DQN obtains an immediate positive reward at each time step and has a good performance.

5.3 Deep Sea

In this part, we conduct experiments to highlight the properties discussed in Section 4.4. We consider a new testbed: deep sea (Osband et al., 2020) because this task is "simple" so that we can have a better understanding of algorithmic properties. In particular, there are two actions in this environment:

[7] Running one algorithm on an environment with 200M frames as in (Mnih et al., 2015; Hessel et al., 2018; Taïga et al., 2020) requires about \$20,160 and 30 days.

[8] Since running 56 experiments requires lots of computation resources, we use training results of Double DQN from DQN Zoo: https://github.com/deepmind/dqn_zoo/blob/master/results.tar.gz.

[9] The training results of OB2I are shared by authors in (Bai et al., 2021) (private communication). Note that reported results in (Bai et al., 2021) are based on the best policy in hindsight after 20M frames.

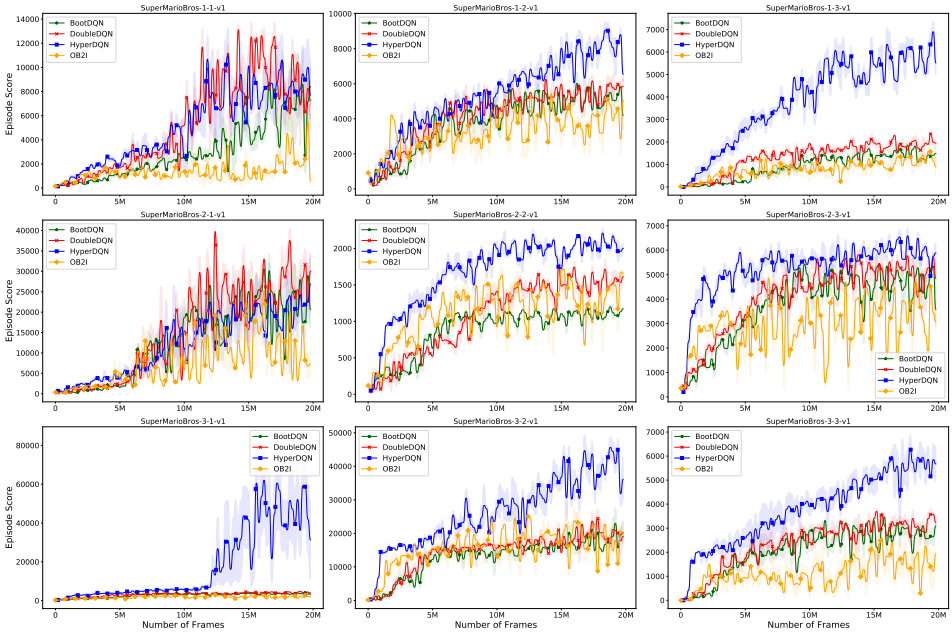
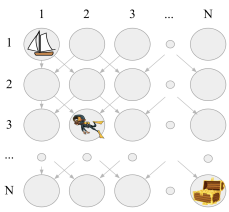


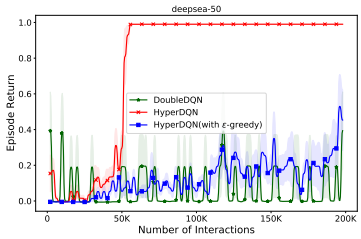
Figure 4: Episode score over 9 environments in SuperMarioBros suite.

move left and move right; see Figure 5a. The reward is sparse and is only released when the agent always takes the "right" action to obtain the treasure at the corner.

We provide evidence that the dithering strategy could violate the commitment property required by deep exploration. By the design of deep sea, once the agent takes a wrong action, it cannot obtain a positive reward during this episode. Hence, the perturbation by ϵ-greedy could ruin far-sighted behaviors generated by the algorithm.



(a) Illustration for deep sea.



(b) Episode return.

Figure 5: Ablation study on deep sea (Osband et al., 2020).

The numerical result (with 5 random seeds) is displayed in Figure 5b. For completeness, we also show the performance of Double DQN (with ϵ-greedy) as a reference. We observe the performance of HyperDQN becomes bad if we combine ϵ-greedy with the randomized exploration strategy. Similar results are also observed on Atari and SuperMarioBros (see Figure 10 and Figure 13 in Appendix).

6 Conclusion

In this work, we present a practical exploration method to address the limitations of RLSVI and BootDQN. To reinforce the central idea, we leverage the hypermodel (Dwaracherla et al., 2020) and extend it from the bandit tasks to RL problems. Several algorithmic designs are developed to release the power of randomized exploration. Further directions include extending the idea for continuous control (Lillicrap et al., 2016; Haarnoja et al., 2018) (i.e., building a randomized actor-critic with a similar architecture with HyperDQN; see Appendix F.5 for preliminary results), utilizing the developed epistemic uncertainty qualification tool for offline RL (Fujimoto et al., 2019; Levine et al., 2020), and acquiring an informative prior with human demonstrations to accelerate exploration (Hester et al., 2018; Sun et al., 2018) (see Appendix F.6 for the discussion).

ETHICS STATEMENT

This work focuses on designing an efficient exploration method to improve the sample efficiency of reinforcement learning. This work may help reinforcement learning to be better used in the real world. There could be some unexpected consequences if the reinforcement learning is abused. For example, a robot is trained maliciously to hurt people.

REPRODUCIBILITY STATEMENT

First, a formal statement and proof of Theorem 1 are given in Appendix C.3. Second, the implementation details of the proposed algorithm HyperDQN can be found in Appendix B.2. Third, the experiment details of numerical results in Section 5 are provided in Appendix D.

REFERENCES

Mohammad Gheshlaghi Azar, Ian Osband, and Rémi Munos. Minimax regret bounds for reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, pp. 263–272, 2017.

Kamyar Azizzadenesheli, Emma Brunskill, and Animashree Anandkumar. Efficient exploration through bayesian deep q-networks. In *Information Theory and Applications Workshop*, pp. 1–9, 2018.

Chenjia Bai, Lingxiao Wang, Lei Han, Jianye Hao, Animesh Garg, Peng Liu, and Zhaoran Wang. Principled exploration via optimistic bootstrapping and backward induction. In *Proceedings of the 38th International Conference on Machine Learning*, pp. 577–587, 2021.

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

Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems 29*, pp. 1471–1479, 2016.

Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, pp. 449–458, 2017.

Ronen I. Brafman and Moshe Tennenholtz. R-MAX - A general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Maching Learning Research*, 3:213–231, 2002.

Yuri Burda, Harrison Edwards, Deepak Pathak, Amos J. Storkey, Trevor Darrell, and Alexei A. Efros. Large-scale study of curiosity-driven learning. In *Proceedings of the 7th International Conference on Learning Representations*, 2019a.

Yuri Burda, Harrison Edwards, Amos J. Storkey, and Oleg Klimov. Exploration by random network distillation. In *Proceedings of the 7th International Conference on Learning Representations*, 2019b.

Qi Cai, Zhuoran Yang, Chi Jin, and Zhaoran Wang. Provably efficient exploration in policy optimization. In *Proceedings of the 37th International Conference on Machine Learning*, pp. 1283–1294, 2020.

Maria Dimakopoulou and Benjamin Van Roy. Coordinated exploration in concurrent reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, pp. 1270–1278, 2018.

Vikranth Dwaracherla, Xiuyuan Lu, Morteza Ibrahimi, Ian Osband, Zheng Wen, and Benjamin Van Roy. Hypermodels for exploration. In *Proceedings of the 8th International Conference on Learning Representations*, 2020.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, Volodymyr Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In *Proceedings of the 36th International Conference on Machine Learning*, pp. 2052–2062, 2019.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning*, pp. 1856–1865, 2018.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, pp. 3215–3222, 2018.

Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, Gabriel Dulac-Arnold, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep q-learning from demonstrations. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, pp. 3223–3230, 2018.

Thomas Jaksch, Ronald Ortner, and Peter Auer. Near-optimal regret bounds for reinforcement learning. *Journal of Maching Learning Research*, 11:1563–1600, 2010.

Chi Jin, Zeyuan Allen-Zhu, Sébastien Bubeck, and Michael I. Jordan. Is q-learning provably efficient? In *Advances in Neural Information Processing Systems 30*, pp. 4868–4878, 2018.

Chi Jin, Zhuoran Yang, Zhaoran Wang, and Michael I. Jordan. Provably efficient reinforcement learning with linear function approximation. In *Proceedings of the 33rd Annual Conference on Learning Theory*, pp. 2137–2143, 2020.

Sham Machandranath Kakade. *On the sample complexity of reinforcement learning*. PhD thesis, University of London London, England, 2003.

Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018. URL https://github.com/Kautenja/gym-super-mario-bros.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pp. 9–48. Springer, 1998.

Su Young Lee, Choi Sungik, and Sae-Young Chung. Sample-efficient deep reinforcement learning via episodic backward update. In *Advances in Neural Information Processing Systems 32*, pp. 2110–2119, 2019.

Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv*, 2005.01643, 2020.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *Proceedings of the 4th International Conference on Learning Representations*, 2016.

Xiuyuan Lu and Benjamin Van Roy. Ensemble sampling. In *Advances in Neural Information Processing Systems 30*, pp. 3260–3268, 2017.

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

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, pp. 1928–1937, 2016.

Brendan O'Donoghue, Ian Osband, Rémi Munos, and Volodymyr Mnih. The uncertainty bellman equation and exploration. In *Proceedings of the 35th International Conference on Machine Learning*, pp. 3836–3845, 2018.

Ian Osband. Risk versus uncertainty in deep learning: Bayes, bootstrap and the dangers of dropout. In *NIPS workshop on bayesian deep learning*, 2016a.

Ian Osband. *Deep Exploration via Randomized Value Functions*. PhD thesis, Stanford University, USA, 2016b.

Ian Osband and Benjamin Van Roy. Why is posterior sampling better than optimism for reinforcement learning? In *Proceedings of the 34th International Conference on Machine Learning*, pp. 2701–2710, 2017.

Ian Osband, Daniel Russo, and Benjamin Van Roy. (more) efficient reinforcement learning via posterior sampling. In *Advances in Neural Information Processing Systems 26*, pp. 3003–3011, 2013.

Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped DQN. In *Advances in Neural Information Processing Systems 29*, pp. 4026–4034, 2016a.

Ian Osband, Benjamin Van Roy, and Zheng Wen. Generalization and exploration via randomized value functions. In *Proceedings of the 33rd International Conference on Machine Learning*, pp. 2377–2386, 2016b.

Ian Osband, John Aslanides, and Albin Cassirer. Randomized prior functions for deep reinforcement learning. In *Advances in Neural Information Processing Systems 31*, pp. 8626–8638, 2018.

Ian Osband, Benjamin Van Roy, Daniel J. Russo, and Zheng Wen. Deep exploration via randomized value functions. *Journal of Machine Learning Research*, 20(124):1–62, 2019.

Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepesvári, Satinder Singh, Benjamin Van Roy, Richard S. Sutton, David Silver, and Hado van Hasselt. Behaviour suite for reinforcement learning. In *Proceedings of the 8th International Conference on Learning Representations*, 2020.

Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning*, pp. 2778–2787, 2017.

John Quan and Georg Ostrovski. DQN Zoo: Reference implementations of DQN-based agents. GitHub, 2020. URL http://github.com/deepmind/dqn_zoo.

Tabish Rashid, Bei Peng, Wendelin Boehmer, and Shimon Whiteson. Optimistic exploration even with a pessimistic initialisation. In *Proceedings of the 8th International Conference on Learning Representations*, 2020.

Daniel Russo. Worst-case regret bounds for exploration via randomized value functions. In *Advances in Neural Information Processing Systems 32*, pp. 14410–14420, 2019.

Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. A tutorial on thompson sampling. *Foundations and Trends in Machine Learning*, 11(1):1–96, 2018.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *Proceedings of the 4th International Conference on Learning Representations*, 2016.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Bradly C. Stadie, Sergey Levine, and Pieter Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv*, 1507.00814, 2015.

Ruoyu Sun. Optimization for deep learning: theory and algorithms. *arXiv*, 1912.08957, 2019.

Wen Sun, J. Andrew Bagnell, and Byron Boots. Truncated horizon policy search: Combining reinforcement learning & imitation learning. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

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

Adrien Ali Taïga, William Fedus, Marlos C. Machado, Aaron C. Courville, and Marc G. Bellemare. On bonus based exploration methods in the arcade learning environment. In *Proceedings of the 8th International Conference on Learning Representations*, 2020.

Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. #exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in Neural Information Processing Systems 30*, pp. 2753–2762, 2017.

Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, Nicolas Heess, and Yuval Tassa. dm_control: Software and tasks for continuous control. *Software Impacts*, 6:100022, 2020.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pp. 2094–2100, 2016.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, pp. 1995–2003, 2016.

Andrea Zanette, David Brandfonbrener, Emma Brunskill, Matteo Pirotta, and Alessandro Lazaric. Frequentist regret bounds for randomized least-squares value iteration. In *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics*, pp. 1954–1964, 2020.

A APPENDIX ORGANIZATION

The appendix is organized as follows.

- Appendix B provides the preliminary about RLSVI (Osband et al., 2016b; 2019) and the algorithmic details of the proposed method HyperDQN.
- Appendix C reviews the Bayesian linear regression knowledge and gives the proof of Theorem 1, i.e., how the hypermodel is connected with the posterior distribution in Bayesian linear regression.
- Appendix D presents all kinds of experimental details such as environment preprocessing and the hyperparameters of algorithms.
- Appendix E offers additional empirical results to support the claims regarding the performance, algorithm design, and computation complexity covered in the main content.
- Appendix F discusses all kinds of related topics such as why ϵ-greedy is still used in existing efficient algorithms, why a direct extension of RLSVI could fail, why a direct extension of the architecture in (Dwaracherla et al., 2020) could fail, and possible extensions of HyperDQN.

B ALGORITHM DETAILS

B.1 RANDOMIZED LEAST-SQUARE VALUE ITERATION (RLSVI)

In this part, we offer a brief introduction to randomized least-square value iteration (RLSVI) (Osband et al., 2016b; 2019). We hope this helps readers understand how RLSVI works and the underlying limitations of RLSVI.

In its original form, RLSVI is designed for the episodic Markov Decision Process for easy analysis. Here the word of "episodic" means the planning horizon (i.e., the episode length) is a finite number $T > 0$. Given a feature map $\phi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}^d$ in advance, RLSVI assumes the optimal Q-value function is a linear model with respect to this feature map. That is, for any stage $t = 0, 1, \cdots, T-1$, RLSVI assumes

$$Q_t^\star(s, a) = \phi_t(s, a)^\top \theta_t^\star, \tag{B.1}$$

where $\theta_t^\star \in \mathbb{R}^d$ is the unknown optimal parameter and $Q_t^\star$ is the optimal value function at stage t. Note that ϕ_t differs from stage to stage, but they are all fixed along the training.

With the above assumption, RLSVI takes three steps to balance the trade-off between exploration and exploitation.

- **(Updating)** Conditioned on the collected data, RLSVI estimates the optimal parameter $\theta_t^\star$ in a Bayesian way. In particular, RLSVI obtains the posterior distribution over $\theta_t^\star$ via solving a Bayesian linear regression problem[10]. More specifically, the Bayesian linear regression problem consists of the fixed feature ϕ_t and the label y_t generated by dynamic programming; see (B.3). Furthermore, the posterior distribution can be obtained by the closed-form solution in (B.4).

- **(Sampling)** With the posterior distribution over $\theta_t^\star$, RLSVI samples a specific parameter $\widetilde{\theta}_t$, which forms the Q-value function $\widetilde{Q}_t(s, a) = \phi_t(s, a)^\top \widetilde{\theta}_t$.

- **(Interaction)** RLSVI takes the greedy action $\operatorname{argmax}_a \widetilde{Q}(s_t, a)$ at stage t to collect new data.

Let k indicates the episode count and t indicates the stage/period count. Let $[T]$ indicates the set $\{0, 1, 2, \cdots, T-1\}$. For instance, $\theta_{k,t}^\dagger \in \mathbb{R}^d$ means the posterior mean at episode k and stage t. With these notations, the procedure of RLSVI is outlined in Algorithm 1.

Algorithm 1 RLSVI (Osband et al., 2016b; 2019)

1: posterior mean $\theta_{0,t}^\dagger \leftarrow$ prior mean $\bar{\theta}_t$, posterior covariance $\Sigma_{0,t}^\dagger \leftarrow$ prior covariance $\sigma_p^2 I, \forall t \in [T]$.
2: **for** episode $k = 0, 1, 2, \cdots$ **do**
3: **for** stage $t = 0, 1, 2, \cdots, T-1$ **do** ▷ Sampling
4: sample $\widetilde{\theta}_t \sim \mathcal{N}(\theta_{k,t}^\dagger, \Sigma_{k,t}^\dagger)$.
5: **end for**
6: **for** stage $t = 0, 1, 2, \cdots, T-1$ **do** ▷ Interaction
7: observe state s_t.
8: take the greedy action $a_t = \operatorname{argmax}_a \phi(s_t, a)^\top \widetilde{\theta}_t$.
9: receive reward $r(s_t, a_t)$.
10: **end for**
11: **for** stage $t = 0, 1, 2, \cdots, T-1$ **do** ▷ Update
12: $\theta_{k+1,t}^\dagger, \Sigma_{k+1,t}^\dagger \leftarrow$ update the posterior distribution (see (B.4)).
13: **end for**
14: **end for**

For Line 12 in Algorithm 1, RLSVI leverages the tool of Bayesian linear regression. Abstractly, given the feature matrix $X \in \mathbb{R}^{N \times d}$ (N is the number of samples) and the label vector $Y \in \mathbb{R}^N$, the posterior distribution over $\theta^\star$ can be computed in a closed-form way. More concretely, RLSVI forms

[10] Bayesian linear regression is reviewed in Appendix C.1.

X_t and Y_t with the state-action pairs collected at stage t up to episode k:

$$X_t = \begin{bmatrix} \phi_t(s_{0,t}, a_{0,t}) \\ \phi_t(s_{1,t}, a_{1,t}) \\ \cdots \\ \phi_t(s_{k,t}, a_{k,t}) \end{bmatrix} \in \mathbb{R}^{(k+1)\times d}, \quad Y_t = \begin{bmatrix} y_{0,t} \\ y_{1,t} \\ \cdots \\ y_{k,t} \end{bmatrix} \in \mathbb{R}^{k+1}, \tag{B.2}$$

where

$$y_{k,t} = \begin{cases} r_{k,t} + \max_a (\phi_{t+1}^\top \widetilde{\theta}_{k,t+1})(s_{k,t+1}, a) & \text{if } t < T-1 \\ r_{k,t} & \text{if } t = T-1 \end{cases} \tag{B.3}$$

With such defined X_t and Y_t, RLSVI can compute the posterior distribution $\mathcal{N}(\mathbb{E}[\theta_t^\star \mid \mathcal{D}], \mathrm{Cov}[\theta_t^\star \mid \mathcal{D}])$ over $\theta_t^\star$:

$$\begin{aligned} \mathbb{E}[\theta_t^\star \mid \mathcal{D}] &:= \theta_{k+1,t}^\dagger = \left(\frac{1}{\sigma_\omega^2} X_t^\top X_t + \frac{1}{\sigma_p^2} I\right)^{-1} \left(\frac{1}{\sigma_\omega^2} X_t^\top Y_t + \frac{1}{\sigma_p^2} \bar{\theta}_t\right), \\ \mathrm{Cov}[\theta_t^\star \mid \mathcal{D}] &:= \Sigma_{k+1,t}^\dagger = \left(\frac{1}{\sigma_\omega^2} X_t^\top X_t + \frac{1}{\sigma_p^2} I\right)^{-1}. \end{aligned} \tag{B.4}$$

Remark 2. As the feature is fixed in RLSVI, there exists an efficient implementation of RLSVI based on the incremental update. Specifically, let Φ be the feature covariance matrix, i.e.,

$$\Phi_{k+1,t} = \frac{1}{\sigma_\omega^2} X_t^\top X_t + \frac{1}{\sigma_p^2} I.$$

For easy presentation, let $\phi_{k,t}$ denote $\phi(s_{k,t}, a_{k,t})$. Then, we have the incremental update formula:

$$\Phi_{k+1,t} = \Phi_{k,t} + \frac{1}{\sigma_\omega^2} \phi_{k,t} \phi_{k,t}^\top, \quad \text{with } \Phi_{0,t} = \frac{1}{\sigma_p^2} I.$$

Furthermore, we have a more efficient update rule for the posterior covariance in (B.4) by the Sherman–Morrison formula (see Lemma 1). More concretely, we have that

$$\Sigma_{k+1,t}^\dagger := \Sigma_{k,t}^\dagger - \frac{(1/\sigma_\omega^2) \cdot \Sigma_{k,t}^\dagger \phi_{k,t} \phi_{k,t}^\top \Sigma_{k,t}^\dagger}{1 + (1/\sigma_\omega^2) \cdot \phi_{k,t}^\top \Sigma_{k,t}^\dagger \phi_{k,t}}, \quad \text{with } \Sigma_{0,t}^\dagger = \sigma_p^2 I.$$

As a side note, the term $X_t^\top Y_t$ in (B.4) can also be implemented in an incremental update way. In short, the posterior distribution is easy to compute when the feature is fixed.

Lemma 1 (Sherman–Morrison formula)**.** Suppose $A \in \mathbb{R}^{n\times n}$ is an invertible square matrix and $u, v \in \mathbb{R}^n$ are column vectors. Then $A + uv^\top$ is invertible if and only if $1 + v^\top A^{-1} u \neq 0$. In this case,

$$\left(A + uv^\top\right)^{-1} = A^{-1} - \frac{A^{-1} u v^\top A^{-1}}{1 + v^\top A^{-1} u}.$$

Here, $uv^\top$ is the outer product of two vectors u and v.

B.2 Implementation of HyperDQN

In this part, we provide more implementation details of HyperDQN.

Recall that we work with two value functions: the prior one and the differential one. The prior value function and the differential value function are independently constructed, meaning there are no shared parameters. The actual Q-value is the weighted sum of two terms:

$$Q(s, a, z) = \beta_{\text{prior}} \cdot Q_{\theta_{\text{prior}}, f_{\nu_{\text{prior}}}(z)}(s, a) + \beta_{\text{differential}} \cdot Q_{\theta_{\text{hidden}}, f_\nu(z)}(s, a), \tag{B.5}$$

where $\beta_{\text{prior}} > 0$ and $\beta_{\text{differential}} > 0$ are two positive scalars. The initial value of β_{prior} and $\beta_{\text{differential}}$ depend on the width of the last hidden layer; see Remark 5 in Appendix D. When $\beta_{\text{prior}} = \beta_{\text{differential}} = 1$, we recover the formulation in (4.2).

The feature extractor (i.e., the hidden layers) are task-specific, which will be discussed in Appendix D. For the hypermodel, the differential part is initialized based on the default initializer of PyTorch while the initialization of the prior hypermodel is a little tricky. Following (Dwaracherla et al., 2020), each row of ν_w is sampled from the unit hypersphere and ν_b is based on the mean of the desired prior distribution. The purpose of this initialization is to guarantee the $\theta_{\text{prior}} = f_{\nu_{\text{prior}}}(z)$ follows a desired Gaussian distribution. For example, if we want to obtain a prior distribution $\mathcal{N}(\mathbf{1}, I)$ for

Algorithm 2 HyperDQN

agent step $n \leftarrow 0$, train step $\ell \leftarrow 0$.
for episode $k = 0, 1, 2, \cdots$ **do**
  generate a random vector $z \sim \mathcal{N}(0, I)$. ▷ Sampling
  instantiate the Q-value function $Q_\theta(s, a, z)$.
  for stage $t = 0, 1, 2, \cdots, T-1$ **do** ▷ Interaction
    observe state s_t.
    take the greedy action $a \leftarrow \operatorname{argmax}_a Q_\theta(s_t, a, z)$.
    receive the next state s_{t+1} and reward $r(s_t, a_t)$.
    sample ξ uniformly from unit hypersphere.
    store $(s_t, a_t, r, \xi, s_{t+1})$ into the replay buffer $\mathcal{D}$.
    agent step $n \leftarrow n + 1$.
    if mod (agent step n, train frequency M) $== 0$ **then** ▷ Update
      sample a mini-batch $\widetilde{\mathcal{D}}$ of (s, a, r, ξ, s') from the replay buffer $\mathcal{D}$.
      sample N random vectors z_i': $\widetilde{\mathcal{Z}} = \{z_i'\}_{i=1}^N$.
      optimize ν and θ_{hidden} using the empirical loss function (B.6) with $\widetilde{\mathcal{D}}$ and $\widetilde{\mathcal{Z}}$.
      train step $\ell \leftarrow \ell + 1$.
    end if
    if mod (train step ℓ, target update frequency G) $== 0$ **then**
      update the target network.
    end if
  end for
end for

$\theta_{\text{prior}} := f_{\nu_{\text{prior}}}(z)$, it suffices to set $\nu_b = \mathbf{1}$ and to uniformly sample ν_w from the unit hypersphere, where $\mathbf{1}$ is a vector with elements of 1.

According to the *population* objective function in (4.2), the *empirical* objective function $\mathcal{L}(\nu, \theta_{\text{hidden}}; \widetilde{\mathcal{Z}}, \widetilde{\mathcal{D}})$ given the set $\widetilde{\mathcal{Z}}$ and mini-batch $\widetilde{\mathcal{D}}$ is

$$\frac{1}{|\widetilde{\mathcal{Z}}|}\left(\sum_{z\in\widetilde{\mathcal{Z}}}\frac{|\mathcal{D}|}{|\widetilde{\mathcal{D}}|}\sum_{(s,a,r,\xi,s')\in\widetilde{\mathcal{D}}}\left(Q_{\text{target}}(s',z)+\sigma_\omega z^\top\xi - Q_{\text{prediction}}(s,a,z)\right)^2 + \frac{\sigma_\omega^2}{\sigma_p^2}\left\|f_\nu(z)\right\|^2\right), \tag{B.6}$$

where

$$Q_{\text{prediction}}(s,a,z) = \beta_{\text{prior}}\cdot Q_{\theta_{\text{prior}}, f_{\nu_{\text{prior}}}(z)}(s,a) + \beta_{\text{differential}}\cdot Q_{\theta_{\text{hidden}}, f_\nu(z)}(s,a),$$

$$Q_{\text{target}}(s',z) = r + \gamma\max_{a'}\left[\beta_{\text{prior}}\cdot Q_{\theta_{\text{prior}}, f_{\nu_{\text{prior}}}(z)}(s',a') + \beta_{\text{differential}}\cdot Q_{\bar{\theta}_{\text{hidden}}, f_{\bar{\nu}}(z)}(s',a')\right].$$

In practice, (B.6) can be efficiently solved by Adam (Kingma & Ba, 2015). In particular, we sample multiple z and the mini-batch from the experience replay buffer to construct the empirical objective function. As such, Adam can be applied to optimize the hidden layers and the hypermodel as illustrated in Figure 1.

Remark 3 (Feature Representation for Prior Value Functions)**.** Different from supervised learning, the prior value function is crucial for online decision making (Osband et al., 2018). For the prior value functions, we not only need to consider the prior hypermodel but also the prior feature representation. Importantly, the feature extractor for the prior value function should be fixed rather than being shared with the differential part. In other words, we use two independent feature networks (i.e., the hidden layers) for the prior value function and the differential value function, respectively. Instead, the output of the prior value function would change if the prior value function and the differential value function share the same hidden layers, which violates the intuition that the prior value function should not be affected by the learning process.

B.3 Sampling From Unit Hypersphere

For completeness, we describe the method for uniform sampling over unit hypersphere. There are two steps:

(1) Generate $\mathbf{x} = (x_1, x_2, \ldots, x_d)$, using a zero mean and unit variance Gaussian distribution. Thus, the probability density of $\mathbf{x}$

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{d}{2}}} \exp\left(-\frac{x_1^2 + x_2^2 + \cdots + x_d^2}{2}\right)$$

is spherically symmetric.

(2) Normalize the vector $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ to a unit vector, namely $\mathbf{x}/\|\mathbf{x}\|$, which gives a sample uniformly over the unit hypersphere. Note that once the vector is normalized, its coordinates are no longer statistically independent.

C Connection Between Bayesian Linear Regression and Hypermodel

C.1 Bayesian Linear Regression

In this part, we review the Bayesian linear regression. Let $x \in \mathbb{R}^d$ be the feature and $y = \langle \theta^\star, x \rangle + \omega^\star$ be the label, where $\omega^\star \in \mathcal{N}(0, \sigma_\omega^2)$ is the observation noise. Here, we treat $\theta^\star$ as a random variable and pose a prior distribution $p_0 : \mathcal{N}(\bar{\theta}_p, \sigma_p^2 I)$ on $\theta^\star$. Given the dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, we can update the posterior probability density over $\theta^\star$ by the Bayes rule:

$$p(\theta^\star \mid \mathcal{D}) \propto p\left(\{(x_i, y_i)\}_{i=1}^N \mid \theta^\star\right) \cdot p_0(\theta^\star).$$

Thanks to the conjugate prior, the posterior distribution over $\theta^\star$ is also a Gaussian distribution $\mathcal{N}(\mathbb{E}[\theta^\star \mid \mathcal{D}], \mathrm{Cov}[\theta^\star \mid \mathcal{D}])$ with

$$\mathbb{E}[\theta^\star \mid \mathcal{D}] = \left(\frac{1}{\sigma_\omega^2} X^\top X + \frac{1}{\sigma_p^2} I\right)^{-1} \left(\frac{1}{\sigma_\omega^2} X^\top Y + \frac{1}{\sigma_p^2} \bar{\theta}_p\right), \tag{C.1}$$

$$\mathrm{Cov}[\theta^\star \mid \mathcal{D}] = \left(\frac{1}{\sigma_\omega^2} X^\top X + \frac{1}{\sigma_p^2} I\right)^{-1}, \tag{C.2}$$

where

$$X = \begin{bmatrix} x_1^\top \\ x_2^\top \\ \cdots \\ x_N^\top \end{bmatrix} \in \mathbb{R}^{N \times d}, \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_N \end{bmatrix} \in \mathbb{R}^N.$$

In summary, given the collected dataset (i.e., the features and labels), we can obtain posterior samples through *sampling* from $\mathcal{N}(\mathbb{E}[\theta^\star \mid \mathcal{D}], \mathrm{Cov}[\theta^\star \mid \mathcal{D}])$.

Recently, Osband et al. (2019) provide another way of obtaining posterior samples through *optimizing* a randomized least square problem.

Lemma 2 ((Osband et al., 2019))**.** Let x be the feature vector and the target y be generated by $y = \langle \theta^\star, x \rangle + \omega^\star$, where $\omega^\star \sim \mathcal{N}(0, \sigma_\omega^2)$ is the noise and the prior on $\theta^\star$ is $\mathcal{N}(\bar{\theta}_p, \sigma_p^2 I)$. Let ω be the algorithmic noise that follows the same distribution with $\omega^\star$ (i.e., $\omega \sim \mathcal{N}(0, \sigma_\omega^2)$) and $\widehat{\theta} \sim \mathcal{N}(\bar{\theta}_p, \sigma_p^2 I)$ be a prior sample. Then,

$$\operatorname*{argmin}_\theta \sum_{(x,y,\omega) \in \mathcal{D}} (y + \omega - \theta^\top x)^2 + \frac{\sigma_\omega^2}{\sigma_p^2} \left\|\theta - \widehat{\theta}\right\|_2^2, \tag{C.3}$$

or

$$\widehat{\theta} + \operatorname*{argmin}_\theta \sum_{(x,y,\omega) \in \mathcal{D}} (y + \omega - (\theta + \widehat{\theta})^\top x)^2 + \frac{\sigma_\omega^2}{\sigma_p^2} \|\theta\|_2^2, \tag{C.4}$$

could yield a sample from the posterior distribution $\mathcal{N}(\mathbb{E}[\theta^\star \mid \mathcal{D}], \mathrm{Cov}[\theta^\star \mid \mathcal{D}])$ defined by (C.1) and (C.2).

First, it is obvious that (C.3) and (C.4) are equivalent by the variable change trick. Let us focus on (C.4) to explain the main idea of Lemma 2. Notice that the posterior sample includes two parts: the prior $\widehat{\theta}$ and the differential θ (θ is a specific optimal solution to (C.4)). In particular, θ is a random variable, which depends on the randomness in ω. Based on this observation, we can leverage the optimality condition to obtain the optimal solution of (C.4). Subsequently, we can verify that the mean of $\widehat{\theta} + \theta$ and its covariance are identical with the posterior mean $\mathbb{E}[\theta^\star \mid \mathcal{D}]$ and posterior covariance $\mathrm{Cov}[\theta^\star \mid \mathcal{D}]$, respectively.

Remark 4. Importantly, Lemma 2 recasts a *sampling* problem to an *optimization* problem, which provides a purely computational cue to obtain a posterior sample. Consequently, it provides another way of implementing RLSVI by solving the least square problem in each iteration. However, there is no clear advantage for this implementation under the case where RLSVI can work (e.g., tabular or linear MDPs).

C.2 HYPERMODEL

In this part, we provide more explanation about the hypermodel. To proceed, note that the main issue of the procedure in Lemma 2 is its computational efficiency. The reason is that we have to solve multiple randomized least square problems if we want to obtain many posterior samples. One of the motivations of the hypermodel is to address this computational issue. Indeed, the original optimization problem in (Dwaracherla et al., 2020) is

$$\min_{\nu} \int_z p(z) \left[\sum_{(x,y,\xi)\in\mathcal{D}} \left(y + \sigma_\omega z^\top \xi - g_{f_\nu(z)}(x)\right)^2 + \frac{\sigma_\omega^2}{\sigma_p^2} \left\| f_\nu(z) - f_{\nu_{\text{prior}}}(z)(x) \right\|^2 \right] (\mathrm{d}z), \quad \text{(C.5)}$$

where $\nu_{\text{prior}} = \nu_0$ is the fixed prior hypermodel parameter. Consider the linear case where $g_{f_\nu(z)}(x) = f_\nu(z)^\top x$, then (C.5) becomes

$$\min_{\nu} \int_z p(z) \left[\sum_{(x,y,\xi)\in\mathcal{D}} \left(y + \sigma_\omega z^\top \xi - f_\nu(z)^\top x\right)^2 + \frac{\sigma_\omega^2}{\sigma_p^2} \left\| f_\nu(z) - f_{\nu_0}(z) \right\|^2 \right] (\mathrm{d}z),$$

which is very similar to (C.3) and the difference is discussed in Remark 1. Intuitively, the hypermodel wants to solve "infinite" randomized least-square problems simultaneously. Through this, the hypermodel can obtain multiple posterior samples by sampling z.

Note that (C.5) has a different form with the loss function in (4.1). However, as discussed in (Dwaracherla et al., 2020, Section 2.5), these two loss functions are equivalent. To derive (4.1) from (C.5), we take three steps: 1) replace $f_\nu(z)$ in (C.5) with $f_{\nu_{\text{prior}}}(z) + f_{\nu'}(z)$; 2) let $g_{f_\nu(z)}(x) = g_{f_{\nu_{\text{prior}}}(z)+f_{\nu'}(z)}(x) = g_{f_{\nu_{\text{prior}}}(z)}(x) + g_{f_{\nu'}(z)}(x)$; 3) change the optimization variable from ν to ν'.

C.3 PROOF OF THEOREM 1

Here, we consider the linear hypermodel, i.e., $f_\nu(z) = \nu_w^\top z + \nu_b$, where $\nu_w \in \mathbb{R}^{N_z \times d}$ and $\nu_b \in \mathbb{R}^d$. By the property of Gaussian distribution, we know that $f_\nu(z) \sim \mathcal{N}(\nu_b, \nu_w^\top \nu_w)$ as $z \sim \mathcal{N}(0, I)$. Therefore, ν_b and $\nu_w^\top \nu_w$ qualify the mean and covariance, respectively.

By the formulation, there are two models in a complete hypermodel: the prior one and the differential one. Specifically,

- The prior hypermodel is
$$f_{\widehat{\nu}}(z) = \widehat{\nu}_w^\top z + \widehat{\nu}_b,$$
where $\widehat{\nu} = (\widehat{\nu}_w, \widehat{\nu}_b)$ is fixed and a non-trainable variable. Here, we set the $\widehat{\nu}_w^\top = \sigma_p I$ and $\widehat{\nu}_b = \bar{\theta}_p$ as the correct prior parameters, then $f_{\widehat{\nu}}(z)$ is a sample from the prior distribution $\mathcal{N}(\bar{\theta}_p, \sigma_p^2 I)$.
- The differential hypermodel is
$$f_\nu(z) = \nu_w^\top z + \nu_b,$$
where $\nu = (\nu_w, \nu_b)$ is a trainable variable.

Let the sample (x_i, y_i) generated under Assumption 1. We augment each data sample $(x_i, y_i) \in \mathcal{D}$ with ξ_i independently sampled from unit hypersphere (refer to Appendix B.3). Then, the dataset becomes

$$\mathcal{D} = \{(x_i, y_i, \xi_i) : i = 1, 2, \ldots, |\mathcal{D}|\},$$

with the augmented noise realizations

$$\Xi = \{\xi_i : i = 1, \ldots, |\mathcal{D}|\}.$$

Theorem 1 (Formal statement)**.** Under Assumption 1, set $\nu_w^{\text{prior}} = \sigma_p I$ and $\nu_b^{\text{prior}} = \bar{\theta}_p$. Let $\nu^\star = (\nu_w^\star, \nu_b^\star)$ be the optimal solution of (4.1) conditioned on specific realizations of ξ , then $\theta := f_{\nu_{\text{prior}}}(z) + f_{\nu^\star}(z) \sim \mathcal{N}(\nu_b^{\text{prior}} + \nu_b^\star, (\nu_w^{\text{prior}} + \nu_w^\star)^\top (\nu_w^{\text{prior}} + \nu_w^\star))$ with

$$\nu_b^{\text{prior}} + \nu_b^\star = \mathbb{E}[\theta^\star \mid \mathcal{D}], \quad (\nu_w^{\text{prior}} + \nu_w^\star)^\top (\nu_w^{\text{prior}} + \nu_w^\star) = \mathrm{Cov}[\theta^\star \mid \mathcal{D}] + \mathtt{err}(\Xi),$$

where

$$\texttt{err}(\Xi) := \mathrm{Cov}[\theta^\star \mid \mathcal{D}] \left(\frac{1}{\sigma_\omega^2} \sum_{(x,\xi)\neq(x',\xi')\in\mathcal{D}} x\xi^\top \xi' x'^\top + \frac{1}{\sigma_\omega \sigma_p} \sum_{(x,\xi)\in\mathcal{D}} (x\xi^\top + \xi x^\top) \right) \mathrm{Cov}[\theta^\star \mid \mathcal{D}]$$

Furthermore, the error term satisfies $\mathbb{E}_\Xi[\texttt{err}(\Xi)] = 0$.

Proof of Theorem 1. For simplicity, we omit the subscript $\star$ for the optimal solution to (4.1) in the proof. That is, we use the shorthand notation $\nu = (\nu_\omega, \nu_b)$ for the optimal solution to (4.1).

The objective function (4.1) for learning hypermodel given the data $\mathcal{D}$ becomes:

$$\mathcal{L}(\nu; \mathcal{D}) = \int_z p(z) \Bigg(\sum_{(x,y,\xi)\in\mathcal{D}} \left(y + \sigma_\omega z^\top \xi - x^\top((\nu_w + \widehat{\nu}_w)^\top z + (\nu_b + \widehat{\nu}_b))\right)^2 + \frac{\sigma_\omega^2}{\sigma_p^2} \left\|\nu_w^\top z + \nu_b\right\|^2 \Bigg) (\mathrm{d}z)$$

According to the first-order optimality condition:

$$\frac{\partial \mathcal{L}}{\partial \nu_b} = \mathbb{E}_z \left[\sum_{(x,y,\xi)\in\mathcal{D}} (-x) \left(y + \sigma_\omega \xi^\top z - x^\top((\nu_w + \widehat{\nu}_w)^\top z + (\nu_b + \widehat{\nu}_b))\right) + \frac{\sigma_\omega^2}{\sigma_p^2} (\nu_w{}^\top z + \nu_b) \mid \mathcal{D} \right] \tag{C.6}$$

$$= \sum_{(x,y,\xi)\in\mathcal{D}} x(x^\top \nu_b + x^\top \widehat{\nu}_b - y) + \frac{\sigma_\omega^2}{\sigma_p^2} \nu_b = 0.$$

It is straightforward to obtain that

$$\sum_{(x,y,\xi)\in\mathcal{D}} x(x^\top \nu_b + x^\top \widehat{\nu}_b - y) + \frac{\sigma_\omega^2}{\sigma_p^2} (\nu_b + \widehat{\nu}_b) = \frac{\sigma_\omega^2}{\sigma_p^2} \widehat{\nu}_b.$$

Then, we can infer that

$$\begin{aligned} \nu_b + \widehat{\nu}_b &= \left(\frac{1}{\sigma_\omega^2} \sum_{x\in\mathcal{D}} xx^\top + \frac{1}{\sigma_p^2} I \right)^{-1} \left(\frac{1}{\sigma_\omega^2} \sum_{(x,y)\in\mathcal{D}} xy + \frac{1}{\sigma_p^2} \widehat{\nu}_b \right) \\ &= \left(\frac{1}{\sigma_\omega^2} X^\top X + \frac{1}{\sigma_p^2} I \right)^{-1} \left(\frac{1}{\sigma_\omega^2} X^\top Y + \frac{1}{\sigma_p^2} \bar{\theta}_p \right) \\ &= \mathbb{E}\left[\theta^\star \mid \mathcal{D}\right], \end{aligned}$$

where $X \in \mathbb{R}^{|\mathcal{D}|\times d}$, $Y \in \mathbb{R}^{|\mathcal{D}|}$, and $\mathbb{E}\left[\theta \mid \mathcal{D}\right]$ is defined in (C.1). This implies that the hypermodel can recover the posterior mean.

For the variable ν_w, we calculate its partial derivative $\partial\mathcal{L}/\partial\nu_w^\top$ as

$$\begin{aligned} &\mathbb{E}_z \left[\sum_{(x,y,\xi)\in\mathcal{D}} \left(y + \sigma_\omega z^\top \xi - x^\top((\nu_w + \widehat{\nu}_w)^\top z + (\nu_b + \widehat{\nu}_b))\right) (-xz^\top) + \frac{\sigma_\omega^2}{\sigma_p^2} (\nu_w^\top z + \nu_b) z^\top \mid \mathcal{D} \right] \\ &= \mathbb{E}_z \left[\sum_{(x,y,\xi)\in\mathcal{D}} \left(-\sigma_\omega z^\top \xi x z^\top + x^\top (\nu_w + \widehat{\nu}_w)^\top z x z^\top\right) + \frac{\sigma_\omega^2}{\sigma_p^2} \nu_w^\top z z^\top \mid \mathcal{D} \right]. \end{aligned} \tag{C.7}$$

To proceed, we utilize the following helpful lemma.

Lemma 3. *Let $z \sim \mathcal{N}(0, I_d)$. For any fixed vector $a \in \mathbb{R}^d$, we have*

$$\mathbb{E}_z \left[a^\top z x z^\top\right] = x a^\top.$$

Proof. The proof is easy if we look at each entry of the matrix,

$$\left[\mathbb{E}_z \left[z^\top a x z^\top\right]\right]_{ij} = \mathbb{E}_z \left[z^\top a [xz^\top]_{ij}\right] = \mathbb{E}_z \left[\left(\sum_{k=1}^d a_k z_k \right) x_i z_j \right] = a_j x_i = [xa^\top]_{ij}.$$

□

Then, with the above Lemma 3 and (C.7), we further calculate

$$\frac{\partial \mathcal{L}}{\partial \nu_w^\top} = \sum_{(x,y,\xi)\in\mathcal{D}} \left(-\sigma_\omega x\xi^\top + xx^\top(\nu_w + \widehat{\nu}_w)\right) + \frac{\sigma_\omega^2}{\sigma_p^2}\nu_w^\top. \tag{C.8}$$

By the first order optimality condition, we have

$$(\nu_w + \widehat{\nu}_w) = \left(\frac{1}{\sigma_\omega^2}X^\top X + \frac{1}{\sigma_p^2}I\right)^{-1}\left(\frac{1}{\sigma_\omega^2}\sum_{(x,\xi)\in\mathcal{D}}\sigma_\omega x\xi^\top + \frac{1}{\sigma_p^2}\widehat{\nu}_w\right).$$

With the posterior covariance in (C.2)

$$\Sigma^{-1} := \mathrm{Cov}\left(\theta^\star \mid \mathcal{D}\right) = \left(\frac{1}{\sigma_\omega^2}X^\top X + \frac{1}{\sigma_p^2}I\right)^{-1},$$

and define the set $\Xi = (\xi_1, \ldots, \xi_{|\mathcal{D}|})$ corresponding to the ξ_i in $(x_i, y_i, \xi_i) \in \mathcal{D}$, we obtain that

$$\begin{aligned}
&(\nu_w + \widehat{\nu}_w)^\top(\nu_w + \widehat{\nu}_w) \\
&= \Sigma^{-1}\left(\frac{1}{\sigma_\omega^2}\sum_{(x,\xi)}\sum_{(x',\xi')} x\xi^\top\xi' x'^\top + \frac{1}{\sigma_p^4}\widehat{\nu}_w^\top\widehat{\nu}_w + \frac{1}{\sigma_\omega^2\sigma_p^2}\sum_{(x,\xi)}\left(\sigma_\omega x\xi^\top\widehat{\nu}_w + \sigma_\omega\widehat{\nu}_w^\top\xi x^\top\right)\right)\Sigma^{-1} \\
&= \Sigma^{-1}\left(\frac{1}{\sigma_\omega^2}X^\top X + \frac{1}{\sigma_p^2}I + \frac{1}{\sigma_\omega^2}\sum_{(x,\xi)\neq(x',\xi')} x\xi^\top\xi' x'^\top + \frac{1}{\sigma_\omega\sigma_p}\sum_{(x,\xi)}(x\xi^\top + \xi x^\top)\right)\Sigma^{-1} \\
&= \Sigma^{-1}\Sigma\Sigma^{-1} + \mathtt{err}(\Xi),
\end{aligned}$$

where the second equality is due to the fact that $\|\xi\| = 1$ and we set $\widehat{\nu}_w = \sigma_p I$ at the beginning; and $\mathtt{err}(\Xi) = \Sigma^{-1}\left(\frac{1}{\sigma_\omega^2}\sum_{(x,\xi)\neq(x',\xi')} x\xi^\top\xi' x'^\top + \frac{1}{\sigma_\omega\sigma_p}\sum_{(x,\xi)}(x\xi^\top + \xi x^\top)\right)\Sigma^{-1}$. Taking the expectation over $\Xi = (\xi_1, \ldots, \xi_{|\mathcal{D}|})$, we have

$$\mathbb{E}_\Xi\left[\sum_{(x,\xi)\neq(x',\xi')} x\xi^\top\xi' x'^\top\right] = 0$$

$$\mathbb{E}_\Xi\left[\sum_{(x,\xi)}(x\xi^\top + \xi^\top x)\right] = 0.$$

Finally, we have

$$\mathbb{E}_\Xi\left[(\nu_w + \widehat{\nu}_w)^\top(\nu_w + \widehat{\nu}_w)\right] = \Sigma^{-1}\Sigma\Sigma^{-1} + \mathbb{E}_\Xi\left[\mathtt{err}(\Xi)\right] = \Sigma^{-1}.$$

This implies the hypermodel can also recover the posterior covariance in expectation. □

C.4 Why Independent Gaussian Noise Cannot Work For Hypermodel?

Following similar steps in Appendix C.3, we want to argue that the posterior approximation result cannot be achieved if we replace $z^\top\xi$ with an independent Gaussian noise ω in (4.1).

Similarly, we augment each sample $(x_i, y_i) \in \mathcal{D}$ with $\omega_i \sim \mathcal{N}(0, \sigma_\omega^2)$ and denote $\boldsymbol{\omega} = (\omega_1, \omega_2, \ldots, \omega_{|\mathcal{D}|})^\top$ as the noise vector. The dataset becomes

$$\mathcal{D} = (x_i, y_i, \omega_i : i = 1, 2, \ldots, |\mathcal{D}|).$$

Now, the objective function for learning hypermodel given the data $\mathcal{D}$ becomes:

$$\mathcal{L}^\omega(\nu; \mathcal{D}) = \int_z p(z)\left(\sum_{(x,y,\omega)\in\mathcal{D}}\left(y + \omega - x^\top((\nu_w + \widehat{\nu}_w)^\top z + (\nu_b + \widehat{\nu}_b))\right)^2 + \frac{\sigma_\omega^2}{\sigma_p^2}\left\|\nu_w^\top z + \nu_b\right\|^2\right)(\mathrm{d}z).$$

Similar to (C.6), we have the first order optimality condition

$$\frac{\partial \mathcal{L}^\omega}{\partial \nu_b} = \sum_{(x,y,\omega)} x(x^\top b + x^\top\widehat{\nu}_b - y + \omega) + \frac{\sigma_\omega^2}{\sigma_p^2}\nu_b = 0.$$

Then, we have

$$\nu_b + \widehat{\nu}_b = \left(\frac{1}{\sigma_\omega^2} X^\top X + \frac{1}{\sigma_p^2} I\right)^{-1} \left(\frac{1}{\sigma_\omega^2} X^\top (Y + \omega) + \frac{1}{\sigma_p^2} \bar{\theta}_p\right) \neq \mathbb{E}\left[\theta \mid \mathcal{D}\right].$$

For the variable ν_w, similar to (C.7), we calculate the partial derivative $\partial \mathcal{L} / \partial \nu_w^\top$ as

$$\begin{aligned}
&\mathbb{E}_z \left[\sum_{(x,y,\xi)\in\mathcal{D}} \left(y + \omega - x^\top ((\nu_w + \widehat{\nu}_w)^\top z + (\nu_b + \widehat{\nu}_b))\right)(-xz^\top) + \frac{\sigma_\omega^2}{\sigma_p^2}(\nu_w^\top z + \nu_b) z^\top \mid \mathcal{D} \right] \\
&= \mathbb{E}_z \left[\sum_{(x,y,\omega)\in\mathcal{D}} \left(-\omega x z^\top + x^\top (\nu_w + \widehat{\nu}_w)^\top z x z^\top\right) + \frac{\sigma_\omega^2}{\sigma_p^2} \nu_w^\top z z^\top \mid \mathcal{D} \right] \\
&= \mathbb{E}_z \left[\sum_{(x,y,\omega)\in\mathcal{D}} \left(x^\top (\nu_w + \widehat{\nu}_w)^\top z x z^\top\right) + \frac{\sigma_\omega^2}{\sigma_p^2} \nu_w^\top z z^\top \mid \mathcal{D} \right] \\
&= \sum_{(x,y,\omega)\in\mathcal{D}} \left(x x^\top (\nu_w + \widehat{\nu}_w)^\top\right) + \frac{\sigma_\omega^2}{\sigma_p^2} \nu_w^\top.
\end{aligned}$$

By the first order optimality condition, we have

$$(\nu_w + \widehat{\nu}_w)^\top = \left(\frac{1}{\sigma_\omega^2} X^\top X + \frac{1}{\sigma_p^2} I\right)^{-1} \left(\frac{1}{\sigma_p^2} \widehat{\nu}_w^\top\right).$$

Therefore, by the fact that $\widehat{\nu}_w^\top \widehat{\nu}_w = \sigma_p^2 I$,

$$(\nu_w + \widehat{\nu}_w)^\top (\nu_w + \widehat{\nu}_w) = \Sigma^{-1} \left(\frac{1}{\sigma_p^2} I\right) \Sigma^{-1} = \frac{1}{\sigma_p^2} \Sigma^{-2} \neq \mathrm{Cov}\left(\theta \mid \mathcal{D}\right).$$

This implies that an independent Gaussian noise ω cannot work and the z-dependent noise is indispensable for posterior approximation.

D EXPERIMENT DETAILS

D.1 ATARI 2600 SUITE

Environment Preprocessing. We follow the same preprocessing as in (Mnih et al., 2015) and (van Hasselt et al., 2016). In particular, there are random no-operation steps (up to 30) before the interaction. Each agent step corresponds to 4 environment steps by repeating the same action while each environment step corresponds to 4 frames of the simulator. The raw score is clipped to $\{-1, 0, +1\}$ for training but the evaluation performance is based on the raw score. Episodes are early stopped after 108K frames as in (van Hasselt et al., 2016). The observation for the agent is based on 4 stacked frames, which is reshaped to $(4, 84, 84)$.

Hyperparameters. All agents use the same network structure as in (Mnih et al., 2015):

$$\text{state} \to \mathbf{conv}(32, 8, 4) \to \mathbf{relu} \to \mathbf{conv}(64, 4, 2) \to \mathbf{relu} \to \mathbf{conv}(64, 3, 1)$$
$$\to \mathbf{mlp}(512) \to \mathbf{relu} \to \mathbf{mlp}(\text{number of actions}),$$

where $\mathbf{conv}(32, 8, 4)$ means a convolution layer with 64 filters of size 8 and stride of 4, and $\mathbf{mlp}(512)$ means a fully-connected layer with output size of 512, and $\mathbf{relu}$ stands for Rectified Linear Units.

The algorithmic parameters follow (Mnih et al., 2015, Table 1). For example, the replay buffer size is 1M; the batch size is 32; the discount factor is 0.99; the target network update frequency is 10K agent steps; the train frequency is 4 agent steps; and the replay starts after 50K agent steps. For algorithms with ϵ-greedy (e.g., Double DQN, BootDQN and OB2I), the exploration ϵ is annealed from 1.0 to 0.1 linearly (from 50K agent steps to 1M agent steps, respectively); the test ϵ is 0.05.

The training result of Double DQN (van Hasselt et al., 2016) is based on DQN Zoo (Quan & Ostrovski, 2020), which is a collection of DQN based algorithms. DQN Zoo is a best-effort replication of respective baselines and publishes the learning data[11]. The training data of OB2I is shared by authors in (Bai et al., 2021) (private communication). We implement Bootstrapped DQN (BootDQN) (Osband et al., 2016a) and HyperDQN with the framework of `tianshou`[12]. The number of ensembles in BootDQN is 10 and the prior scale is 1.0. BootDQN uses the same ϵ-greedy strategy with Double DQN as suggested by (Osband et al., 2016a). Note that we do not implement the "vote" mode since this evaluation strategy does not matter in practice. The hyperparameters of HyperDQN are shown in Table 2. Note that since each z corresponds to 32 mini-batch samples, the effective batch size of HyperDQN is $32 \times 10 = 320$.

We use three random seeds: 2021, 2022 and 2023 for HyperDQN and BootDQN. Based on tianshou, n-step return (Mnih et al., 2016) of 3 and double Q learning (van Hasselt et al., 2016) are implemented but dueling network (Wang et al., 2016) and prioritized replay (Schaul et al., 2016) are omitted. The optimizer for both methods is Adam (Kingma & Ba, 2015) with learning rate of 0.0001.

Table 2: Hyperparameters of HyperDQN for Atari and SuperMarioBros.

	Atari	SuperMarioBros	Deep Sea
z dimension N_z	32	32	32
prior scale β_{prior} in (B.5)	0.1	0.1	1.0
differential scale $\beta_{\text{differential}}$ in (B.5)	0.1	0.1	1.0
number of z for training in (B.6)	10	10	10
noise scale σ_w in (B.6)	0.01	0.01	0.01
prior regularization scale σ_p in (B.6)	0.1	0.1	1.0
learning rate	0.0001	0.0001	0.001

D.2 SUPERMARIOBROS

Environment Preprocessing. The observation and action preprocessing are the same with Atari 2600 suite. Two things are different: 1) there is no random no operation; 2) the training reward is based on $0.01 \times$ `raw score` rather than clipping. The maximum episode length is $1500 \times 4 = 6000$ frames.

[11] https://github.com/deepmind/dqn_zoo/blob/master/results.tar.gz
[12] https://github.com/thu-ml/tianshou

Hyperparameters. The neural network structure and algorithmic parameters are almost identical with Atari 2600 suite. There are two common parameters are suggested by tianshou to accelerate training without sacrificing the final performance: target network update frequency is 500 agent steps and the train frequency is 10 agent steps[13]. Unfortunately, OB2I is sensitive to this change so that we use the original parameters that the training frequency is 4 and the target network update frequency is 10000. All algorithms use three random seeds: 2021, 2022, and 2023.

D.3 Deep Sea

Deep sea task is modified from `bsuite`[14] (Osband et al., 2020). We use the version where the reward is deterministic and the state is in one-hot feature. The state space size is $50 \times 50/2 = 1250$ and the action space size is 2. The maximum episode length is 50.

The neural network architecture is :

$$\text{state} \to \mathbf{mlp}(64) \to \mathbf{relu} \to \mathbf{mlp}(64) \to \mathbf{relu} \to \mathbf{mlp}(\text{number of actions}).$$

Different from Atari, the train frequency is 10; the target network update frequency is 10; the experience replay buffer size is 20K. For ϵ-greedy, we diminish ϵ from 1.0 to 0.05 linearly (from 10K interactions to 100K interactions, respectively); in other words, the learning starts after 10K interactions for Double DQN. The test ϵ is 0.0 since any dithering may lead to a failure. Other hyperparameters are given in Table 2.

Remark 5 (Parameter choice of the prior scale and differential scale)**.** Since the hypermodel is used to model the parameter of the last control layer, it is helpful to obey the principle of neural network initialization (Glorot & Bengio, 2010). Specifically, it is beneficial to set β_{prior} and $\beta_{\text{differential}}$ proportional to $1/d$, where d the width of the last hidden layer (i.e., the dimension of the feature). Hence, for deep sea we use $\beta_{\text{prior}} = \beta_{\text{differential}} = 1.0$ while for Atari 2600 and SuperMarioBros we use $\beta_{\text{prior}} = \beta_{\text{differential}} = 0.1$.

D.4 Bayesian Linear Regression

The 2-dimensional Bayesian linear regression problem used in Section 4.3 is based on $y = \theta_w \cdot x + \theta_b + \epsilon$, where $x, y, \theta_w, \theta_b, \epsilon \in \mathbb{R}$. Specifically, ϵ is sampled from the Gaussian distribution. To generate the dataset, x is sampled from $[-4, 4]$ uniformly. The total number of training samples is 50. The prior distribution for (θ_w, θ_b) is $\mathcal{N}(0, I_2)$ while the actual value (θ_w, θ_b) is $(6.06, 0.47)$ for the dataset generation.

We obtain the exact posterior distribution by the Bayes update rule. For hypermodel, the dimension of z is 2. We use gradient descent with a learning rate of 0.005 and momentum of 0.9. The iteration number of the gradient descent is 10000. After the optimization, we visualize the 5000 samples from different distributions (e.g., the posterior distribution and the one transformed by the hypermodel) in Figure 2. Indeed, the KL-divergence between the exact posterior distribution and the counterparts by `hypermodel(`$z^\top \xi$`)` and `hypermodel(`ω`)` are 0.1 and 326.4, respectively.

D.5 Parameter Choice

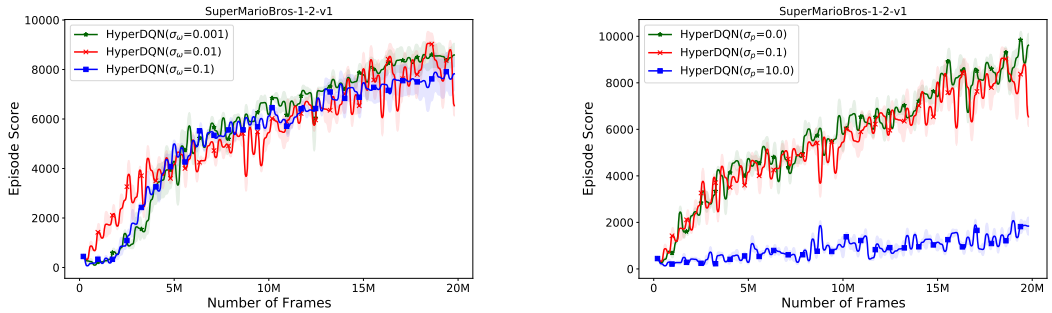
Ablation study about the noise scale σ_ω. In this part, we provide the ablation study about the noise scale σ_ω. The numerical result is displayed in Figure 6a. We observe the performance of HyperDQN is not very sensitive to σ_ω.

Ablation study about the prior scale σ_p. In this part, we provide the ablation study about the prior scale σ_p. The numerical result is shown in Figure 6b. We find that a large prior scale results in a poor performance. The reason is that the posterior update is slow under this case.

Ablation study about the non-linear hypermodel. In this paper, we mainly focus on the linear hypermodel. Here, we investigate the variant with a non-linear hypermodel (i.e., an MLP hypermodel). In particular, we consider the hypermodel is a two-layer neural network of width 64

[13] This parameter choice does not affect the performance too much; see Figure 9.

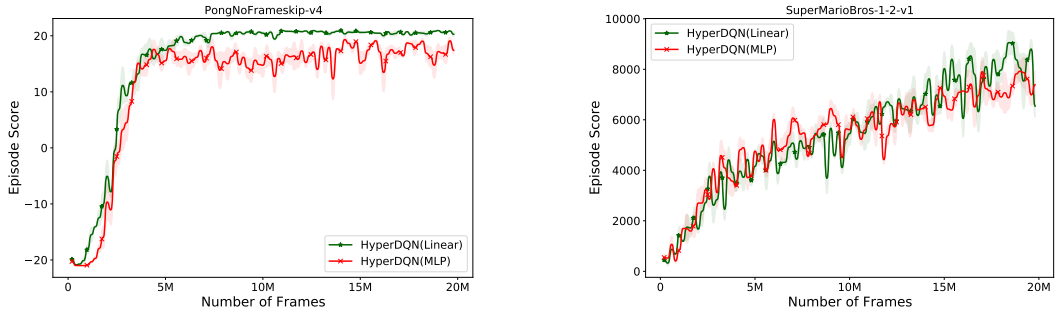
[14] https://github.com/deepmind/bsuite/blob/master/bsuite/environments/deep_sea.py



(a) Ablation study about the noise scale σ_ω. (b) Ablation study about the prior scale σ_p.

Figure 6: Ablation study about the noise scale σ_ω and prior scale σ_p.

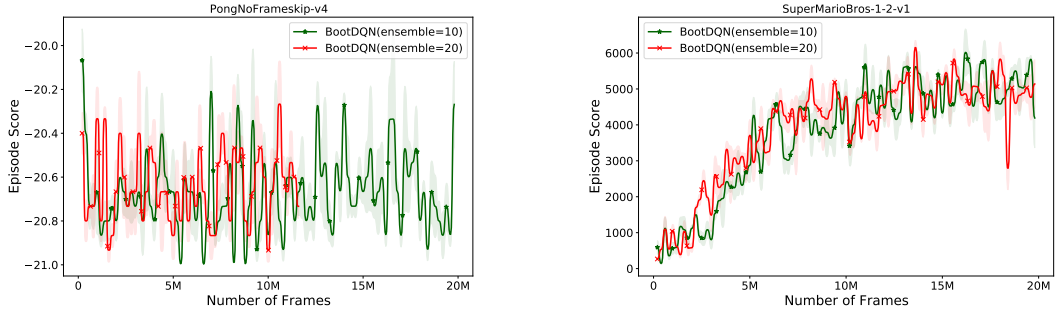
with ReLU activation. We call this variant as `HyperDQN(MLP)` and the original HyperDQN as `HyperDQN(Linear)`. It is intuitive that `HyperDQN(MLP)` has a more powerful posterior approximation ability but it may be hard to train `HyperDQN(MLP)` since the architecture is more complex. The empirical results on Atari and SuperMarioBros are shown in Figure 7. We see that HyperDQN does not obtain significant gains by an MLP hypermodel. We conjecture the reason is the trainability issue of the MLP hypermodel.



(a) Episode return on Pong. (b) Episode return on SuperMarioBros-1-2.

Figure 7: Ablation study about the non-linear hypermodel.

Ablation study about the number of ensembles in BootDQN. In this paper, we implement BootDQN with 10 ensembles, which follows the configuration in (Osband et al., 2016a, Section 6.1). However, in (Osband et al., 2018), BootDQN is implemented with 20 ensembles. We remark that the choice of 10 ensembles is commonly used in the previous literature (Rashid et al., 2020; Bai et al., 2021) since it is more computationally cheap. For completeness, we provide the ablation study about this parameter choice; see Figure 8 for the result.



(a) Episode return on Pong. (b) Episode return on SuperMarioBros-1-2.

Figure 8: Ablation study about the number of ensembles in BootDQN.

Ablation study about the training frequency. Here we provide the evidence that the performance does not change too much with a training frequency of 10 instead of 4. The results are shown in Figure 9.

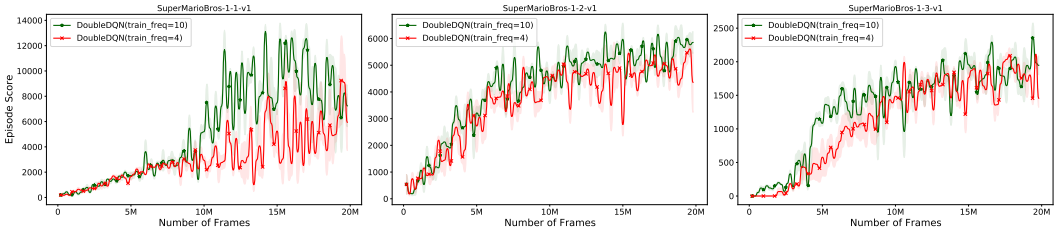


Figure 9: Ablation study about the training frequency.

E ADDITIONAL RESULTS

E.1 ATARI 2600 SUITE

The correlation between training and test performance. We provide the training curve and the test curve in Figure 16 and Figure 17, respectively. Note that the former is based on the clipped score while the latter is based on the raw score.

Remark 6. We highlight the mismatch between the training performance and the test performance due to reward clipping (i.e., clipping the raw score to $\{-1, 0, +1\}$ for training). By comparing HyperDQN and BootDQN, we classify 56 tasks into the following 4 categories:

- The training performance of HyperDQN and BootDQN are comparative but the test performance of HyperDQN is better: `amidar`, `choppercommand`, `doubledunk`, `mspacman`, `privateeye`, `zaxxon` (6).
- The training performance of HyperDQN and BootDQN are comparative but the test performance of HyperDQN is worse: `assault`, `asteroids`, `breakout`, `demonattack`, `phoenix`, `solaris`, `yarsrevenge` (7).
- The training performance of HyperDQN is better but its test performance of is worse: `atlantis`, `skiing` (2).
- The training and test performance of HyperDQN and BootDQN are much correlated: `alien`, `asterix`, `bankheist`, `battlezone`, `beamrider`, `berzerk`, `bowling`, `boxing`, `centipede`, `crazyclimber`, `defender`, `enduro`, `fishingderby`, `freeway`, `frostbite`, `gopher`, `gravitar`, `hero`, `icehockey`, `jamesbond`, `kangaroo`, `krull`, `kungfumaster`, `montezumarevenge`, `namethisgame`, `pitfall`, `pong`, `qbert`, `riverraid`, `roadrunner`, `robotank`, `seaquest`, `spaceinvaders`, `stargunner`, `tennis`, `timepilot`, `tutankham`, `upndown`, `venture`, `videopinball`, `wizardofwor` (41).

On the commitment of randomized exploration. Here we provide the evidence that using the ϵ-greedy strategy could destroy the commitment and leads to poor performance on several well-understood tasks; see the results in Figure 10. We observe that combing the randomized exploration strategy with ϵ-greedy results in poor performance.

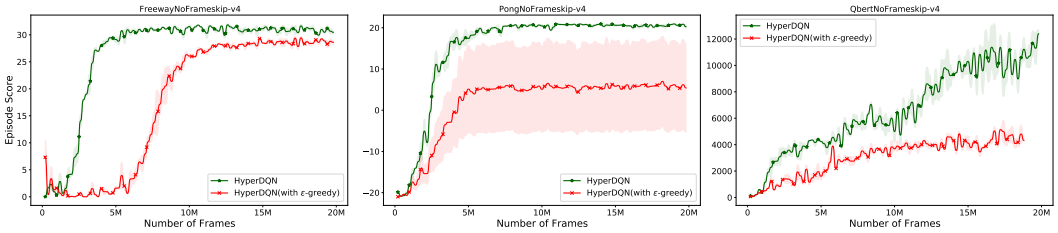


Figure 10: Ablation study about the commitment property (i.e., combination with ϵ-greedy) on Atari (HyperDQN).

Best performance in hindsight. In Section 5, we have evaluated the performance of algorithms in terms of the median score over 56 tasks in the Atari suite. Here, we provide the performance on each individual task; see Table 5. In particular, we save the best policy during the training and re-evaluate it over 200 episodes. Following (Bai et al., 2021), the performance is based on the maximal score over 200 evaluation episodes and is aggregated over 3 random seeds. Note that this metric is used to make a fair comparison with the results reported in (Bai et al., 2021). We observe that HyperDQN is the best on 31 out of 56 tasks.

Relative improvement on each environment. To better understand the improvement of HyperDQN, we visualize the relative score compared with baselines. Specifically, the relative score is calculated as $\frac{\text{proposed}-\text{baseline}}{\max(\text{human},\text{baseline})-\text{human}}$ (Wang et al., 2016). According to the taxonomy in (Bellemare et al., 2016),

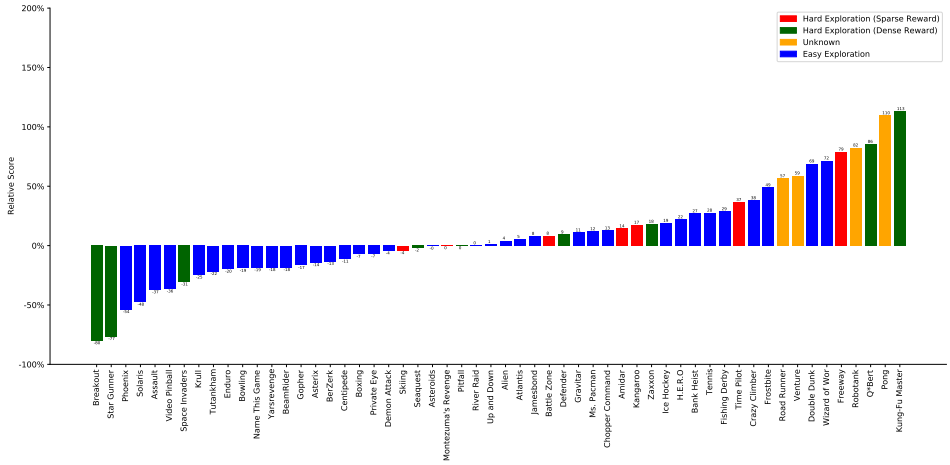


Figure 11: Relative improvement of HyperDQN compared with BootDQN (Osband et al., 2018) on 56 Atari environments. The relative performance is calculated as $\frac{\text{proposed}-\text{baseline}}{\max(\text{human},\text{baseline})-\text{human}}$ (Wang et al., 2016). Environments are grouped according to the taxonomy in (Bellemare et al., 2016, Table 1). "Unknown" indicates such environments are not considered in (Bellemare et al., 2016).

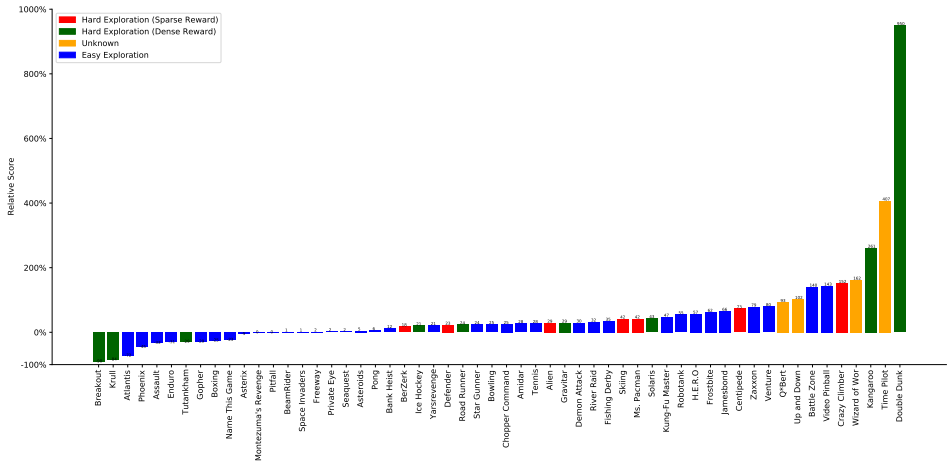


Figure 12: Relative improvement of HyperDQN compared with OB2I (Bai et al., 2021) on 56 Atari environments. The relative performance is calculated as $\frac{\text{proposed}-\text{baseline}}{\max(\text{human},\text{baseline})-\text{human}}$ (Wang et al., 2016). Environments are grouped according to the taxonomy in (Bellemare et al., 2016, Table 1). "Unknown" indicates such environments are not considered in (Bellemare et al., 2016).

we cluster environments by 4 groups: "hard exploration (dense reward)", "hard exploration (sparse reward)", "easy exploration", and "unknown". See the results in Figure 11 and Figure 12.

Remark 7. We observe that HyperDQN has improvements on both "easy exploration" environments (e.g., Battle Zone, Jamesbond, and Pong) and "hard exploration" environments (e.g., Frostbite, Gravitar, and Zaxxon). However, we notice that HyperDQN does not work well on very sparse reward tasks like Montezuma's Revenge. We explain the failure reason as follows. As we have discussed in the introduction, without a good feature, randomized exploration methods are rarely competent. On Montezuma's Revenge, the extremely sparse reward provides limited feedback for feature selection. As a result, it is expected that randomized exploration methods (including BootDQN and NoisyNet) do not work well for this task. In contrast, prediction error based methods (Pathak et al., 2017; Burda et al., 2019b; Rashid et al., 2020) could leverage specific architecture designs to provide auxiliary reward feedback to help feature selection and exploration. As a result, these methods perform well on Montezuma's Revenge. We kindly remind that these methods do not perform well on other tasks because specifically designed architectures in these methods do not generalize well as pointed out in (Taïga et al., 2020).

E.2 SUPERMARIOBROS

On the commitment of randomized exploration. Here we provide the evidence that using the ϵ-greedy strategy could destroy the commitment and leads to poor performance on SuperMarioBros; see the result in Figure 13.

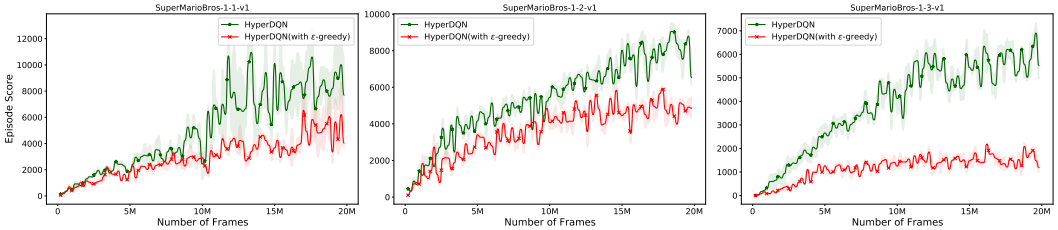


Figure 13: Ablation study about the commitment property (i.e., combination with ϵ-greedy) on SuperMarioBros (HyperDQN).

In addition, we provide the evidence that the randomized exploration approach BootDQN (Osband et al., 2016a) could obtain some gains if ϵ-greedy is not used; see the result in Figure 14. The result suggests that the original implementation of BootDQN does not satisfy the commitment property. Note that the variant without ϵ-greedy is still inferior compared with HyperDQN.

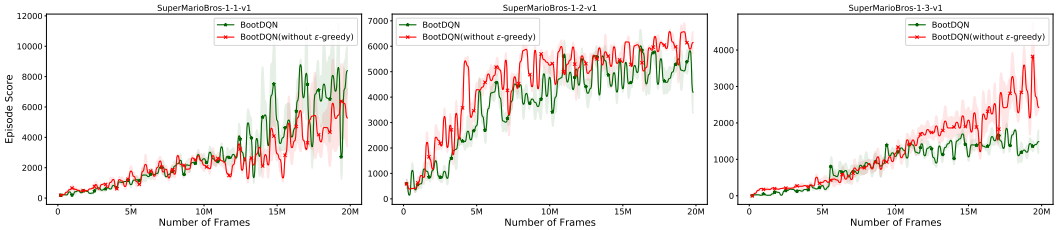


Figure 14: Ablation study about the commitment property (i.e., combination with ϵ-greedy) on SuperMarioBros (BootDQN).

Best performance in hindsight. Here, we provide the comparison with the metric that the best policy in hindsight after 20M frames. In particular, we save the best policy after the training procedure and re-evaluate this best policy with 200 episodes. The *mean* score over 200 episodes and 3 random seeds is given in Table 3. Different from the metric in the Atari suite, we do not use the maximal score metric, which is very sensitive and provides limited insights. We observe that HyperDQN outperforms baselines on 7 out of 9 games.

Table 3: The mean score over 200 evaluation episodes for the best policy in hindsight (after 20M frames) for SuperMarioBros games.

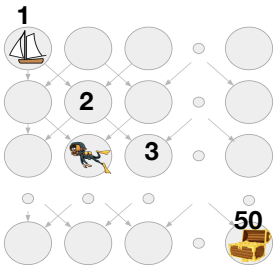
	DoubleDQN	BootDQN	OB2I	HyperDQN
SuperMarioBros-1-1-v1	**8,698**	7,008	4,457	7,924
SuperMarioBros-1-2-v1	5,903	5,665	4,695	**8,266**
SuperMarioBros-1-3-v1	1,989	1,609	1,583	**6,046**
SuperMarioBros-2-1-v1	**31,247**	26,415	14,225	23,046
SuperMarioBros-2-2-v1	1,622	1,092	1,587	**1,983**
SuperMarioBros-2-3-v1	5,515	5,107	4,401	**5,980**
SuperMarioBros-3-1-v1	4,463	3,861	3,251	**48,384**
SuperMarioBros-3-2-v1	20,511	20,954	26,508	**41,139**
SuperMarioBros-3-3-v1	3,416	2,650	3,009	**5,568**

E.3 DEEP SEA

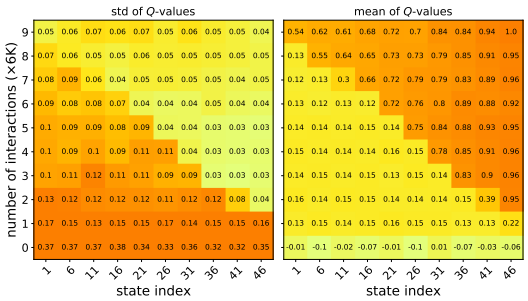
On the multi-step uncertainty. To better understand exploration behaviors by HyperDQN, we also hope to visualize the learned epistemic uncertainty. It is intuitive to expect the epistemic uncertainty

is uniformly high for all states before the interaction. Through learning from the collected data, we hope the epistemic uncertainty becomes low for the "leaf" states (i.e., the bottom states in Figure 15a). In contrast, the epistemic uncertainty could be relatively high for the "root" states (i.e., the top states in Figure 15a) since the transitions from these states contain more possibilities than the "leaf" states. Once the agent could explore to find the treasure and obtain the positive reward, the Q-value becomes high for these valuable "leaf" states. At the same time, the epistemic uncertainty starts to diminish for these states. Correspondingly, the temporal difference learning objective will push the "root" states to increase their Q-values and to reduce their epistemic uncertainty.

Considering states on the diagonal path in Figure 15a, we visualize the standard deviation (as a proxy for the epistemic uncertainty) and the mean of Q-values in Figure 15b. In particular, the x-axis corresponds to states along the optimal path and the y-axis corresponds to the number of interactions. From Figure 15b, we observe that the evolution of the standard deviation and the mean values could verify our intuition about the multi-step uncertainty. In addition, the evolution is also consistent with the learning curve in Figure 5b.



(a) Illustration for deep sea. Digit numbers correspond to the state indices in the right figure.



(b) Standard deviation and mean of Q-values generated by HyperDQN .

Figure 15: Visualization of the multi-step uncertainty on deep sea.

Computation Complexity Comparision. Here we provide the evidence that HyperDQN could be computationally efficient than Bootstrapped DQN (BootDQN) (Osband et al., 2016a). Here, we work on the hard version of the deep sea, which is slightly different from the version we used in Section 5. In particular, actions are randomly masked so that the index 0 may correspond to "left" or "right" at different states (Osband et al., 2020).

Following (Dwaracherla et al., 2020), we measure the performance by the computation complexity[15]:

$$\text{computation complexity} = n_{\text{sgd}} \times n_z \times K, \tag{E.1}$$

where n_{sgd} is the number of SGD steps per iteration, n_z is the number of ensemble (index) samples, and K is the number of episode that the episode return is 0.99. For BootDQN, it uses 10 ensembles (i.e., $n_{\text{sgd}} = n_z = 10$) while HyperDQN samples z from a two dimensional distribution (i.e., $n_z = 2$ and $n_{\text{sgd}} = 20$). For a fair comparison, the BootDQN considered here is identical with HyperDQN except that z is sampled from the set of 10-dimensional one-hot features. The (averaged) empirical result with 10 random seeds (from 2021 to 2030) is shown in Table 4. We see that HyperDQN is clearly more computationally efficient than BootDQN.

Table 4: Ablation study about the computation complexity on the deep sea. The metric is defined in (E.1). A smaller number indicates a better performance. "deep-sea-10" means the width and height of the deep sea are 10; refer to Figure 5a.

	deep-sea-10	deep-sea-15	deep-sea-20	deep-sea-25	deep-sea-30
BootDQN	130K	250K	490K	870K	1,640K
HyperDQN	48K	104K	196K	304K	1,120K

[15] This metric is reasonable since without the restriction of the computation complexity, methods can perfect approximate RLSVI and enjoy the same sample complexity.

Table 5: The maximal score over 200 evaluation episodes for the best policy in hindsight (after 20M frames) for Atari games. The performance of the random policy and the human expert is from https://github.com/deepmind/dqn_zoo/blob/master/dqn_zoo/atari_data.py#L41-L101. The performance of OB2I is from (Bai et al., 2021).

	Random	Human	OB2I	BootDQN	HyperDQN
Alien	227.8	7,127.7	916.9	2,623.3	**2,910.0**
Amidar	5.8	1,719.5	94.0	319.0	**565.7**
Assault	222.4	742.0	2,996.2	**3,182.7**	2,083.0
Asterix	210.0	8,503.3	2,719.0	**3,466.7**	2,283.3
Asteroids	719.1	47,388.7	959.9	**3,353.3**	3,190.0
Atlantis	12,850.0	29,028.1	**3,146,300.0**	835,166.7	880,233.3
Bank Heist	14.2	753.1	378.6	270.0	**470.0**
Battle Zone	2,360.0	13,454.5	8,756.5	22,666.7	**24,333.3**
BeamRider	363.9	16,926.5	3,736.7	**7,002.0**	3,955.3
BerZerk	123.7	2,630.4	493.2	**1,270.0**	936.7
Bowling	23.1	160.7	30.0	**90.3**	64.0
Boxing	0.1	12.1	**75.1**	60.7	56.3
Breakout	1.7	30.5	**423.1**	212.0	43.7
Centipede	2,090.9	12,017.0	2,661.8	**11,051.7**	9,923.0
Chopper Command	811.0	7,387.8	1,100.3	1,900.0	**2,733.3**
Crazy Climber	10,780.5	35,829.4	53,346.7	88,400.0	**117,966.7**
Defender	2,874.5	18,688.9	9,350.0	11,450.0	**12,916.7**
Demon Attack	152.1	1,971.0	6,794.6	**9,148.3**	8,755.0
Double Dunk	-18.6	-16.4	-18.2	-6.0	**2.7**
Enduro	0.0	860.5	**719.0**	626.3	456.0
Fishing Derby	-91.7	5.5	-60.1	-54.3	**-26.3**
Freeway	0.0	29.6	32.1	9.3	**32.7**
Frostbite	65.2	4,334.7	1,277.3	1,853.3	**3,943.3**
Gopher	257.6	2,412.5	**6,359.5**	5,466.7	4,600.0
Gravitar	173.0	3,351.4	393.6	966.7	**1,316.7**
H.E.R.O	1,027.0	30,826.4	3,302.5	13,590.0	**20,156.7**
Ice Hockey	-11.2	0.9	-4.2	-4.0	**-1.7**
Jamesbond	29.0	302.8	434.3	650.0	**700.0**
Kangaroo	52.0	3,035.0	2,387.0	8,666.7	**10,166.7**
Krull	1,598.0	2,665.5	**45,388.8**	10,643.3	8,413.3
Kung-Fu Master	258.5	22,736.3	16,272.2	1,500.0	**26,933.3**
Montezuma's Revenge	0.0	4,753.3	**0.0**	**0.0**	**0.0**
Ms. Pacman	307.3	6,951.6	1,794.9	3,793.3	**4,590.0**
Name This Game	2,292.3	4,076.0	**8,576.8**	8,200.0	7,106.7
Pong	-20.7	14.6	18.7	-17.7	**21.0**
Phoenix	761.4	7,242.6	9,880.0	**11,516.7**	5,733.3
Pitfall	-229.4	6,463.7	**0.0**	**0.0**	**0.0**
Private Eye	24.9	69,571.3	1,174.1	**7,641.0**	2,810.0
Q*Bert	163.9	13,455.0	4,275.0	5,250.0	**16,616.7**
River Raid	1,338.5	17,118.0	2,926.5	8,003.3	**8,020.0**
Road Runner	11.5	7,845.0	21,831.4	17,266.7	**27,033.3**
Robotank	2.2	11.9	13.5	11.7	**19.7**
Seaquest	68.4	42,054.7	332.1	**2,180.0**	1,360.0
Skiing	-17,098.1	-4,336.9	-13,670.4	**-7,810.0**	-8,340.0
Solaris	1,236.3	12,326.7	2,322.4	**12,446.7**	7,113.3
Space Invaders	148.0	1,668.7	904.9	1,390.0	925.0
Star Gunner	664.0	10,250.0	1,290.2	**13,300.0**	3,633.3
Tennis	-23.8	-8.3	-1.0	-1.0	**5.3**
Time Pilot	3,568.0	5,229.2	3,404.5	8,400.0	**10,166.7**
Tutankham	11.4	167.6	**297.0**	269.3	213.3
Up and Down	533.4	11,693.2	5,100.8	16,333.3	**16,520.0**
Venture	0.0	1,187.5	16.1	266.7	**966.7**
Video Pinball	16,256.9	17,667.9	80,607.0	**262,718.0**	172,896.3
Wizard of Wor	563.5	4,756.5	480.7	4,266.7	**7,266.7**
Yarsrevenge	3,092.9	54,576.9	7,367.4	**27,835.0**	18,361.7
Zaxxon	32.5	9,173.3	2,842.0	8,400.0	**10,066.7**

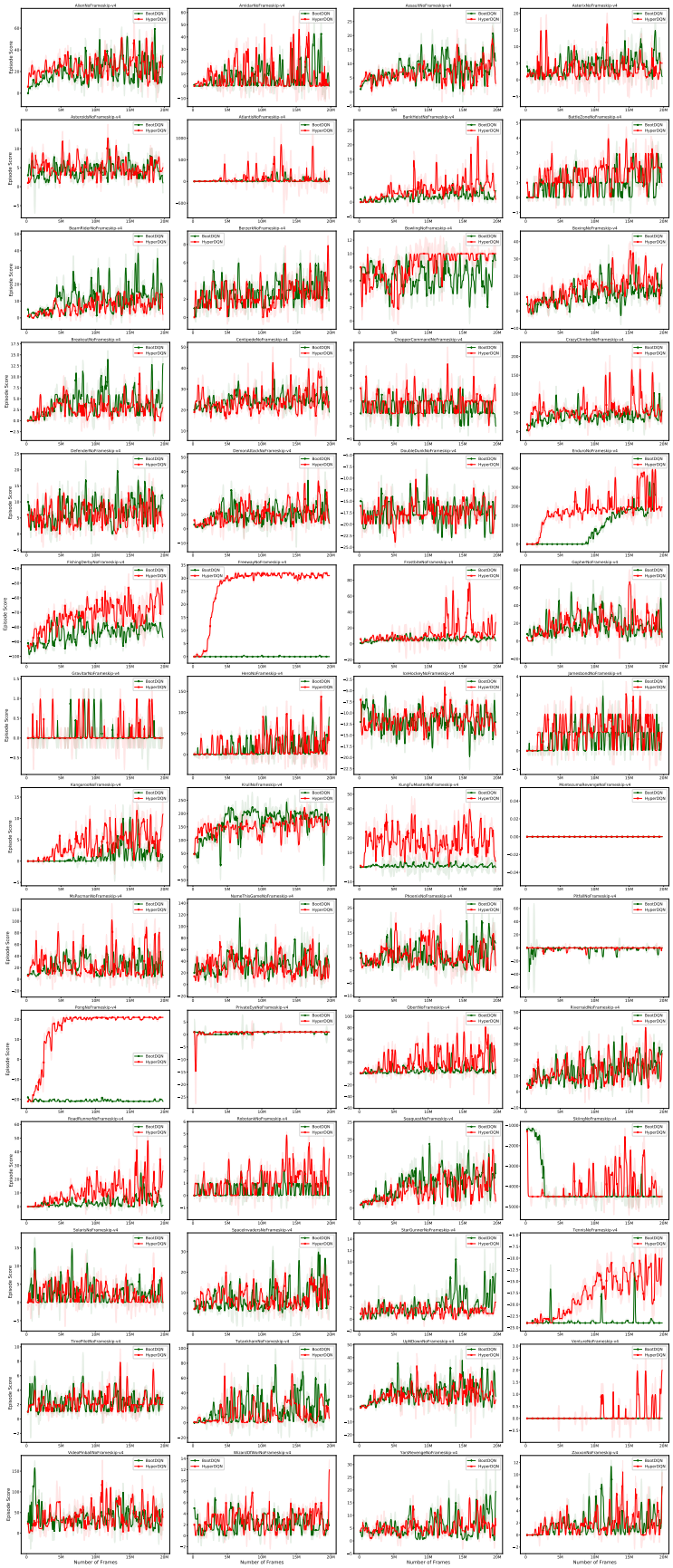


Figure 16: Training curves on Atari 2600 suite. Solid lines correspond the mean over 3 random seeds. Refer to Remark 6 for the discussion.

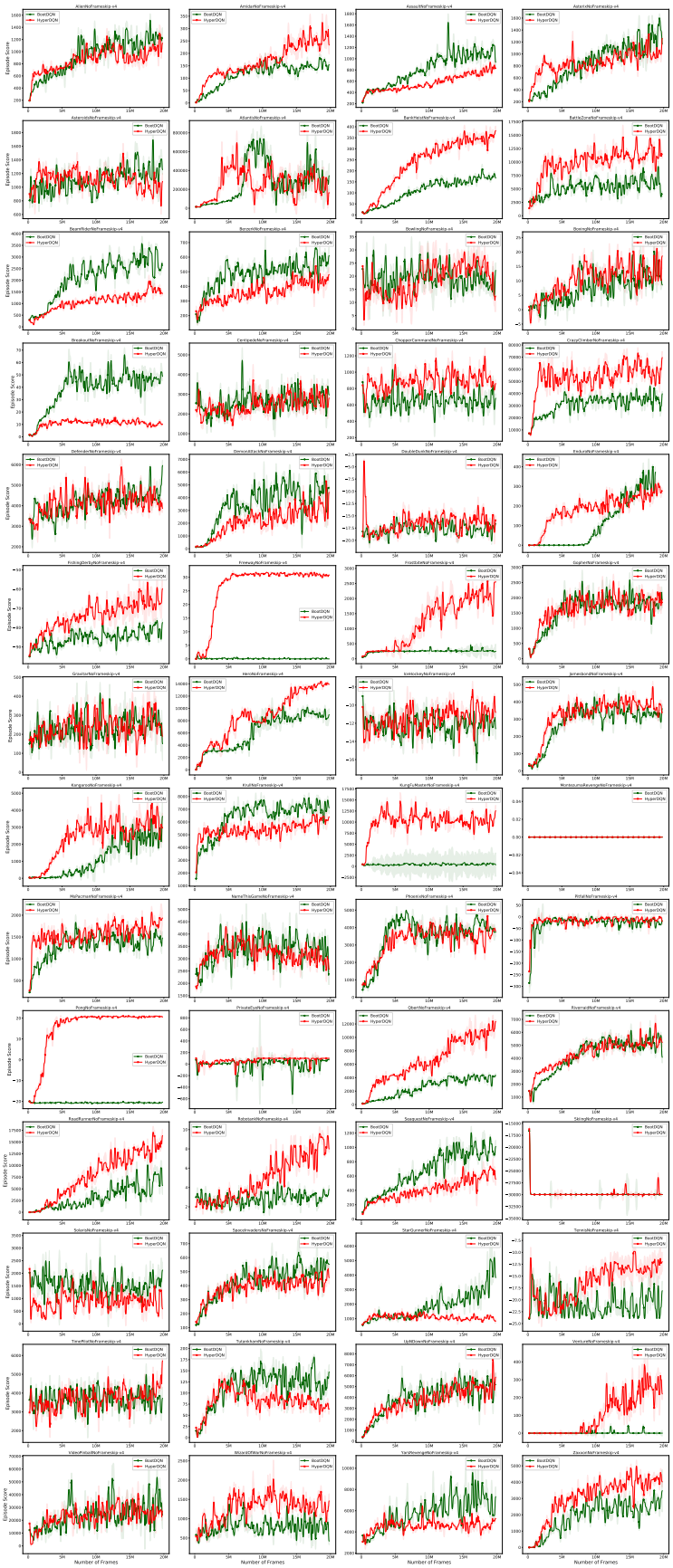


Figure 17: Test curves on Atari 2600 suite. Solid lines correspond the mean over 3 random seeds. Refer to Remark 6 for the discussion.

F DISCUSSION

F.1 WHY BAYESIAN DQN AND DIRECT EXTENSION OF RLSVI COULD FAIL?

Based on the algorithm description of RLSVI in Appendix B.1, one may ask whether we can directly extend RLSVI for deep RL. In fact, Bayesian DQN (Azizzadenesheli et al., 2018) is a "deep" version of RLSVI (Osband et al., 2016b; 2019). In particular, Bayesian DQN leverage the raw feature outputted by a randomly initialized neural network to implement RLSVI. We hope the empirical results presented in this part could help explain why RLSVI could not be directly applied in the deep RL scenario and why the direct extension in (Azizzadenesheli et al., 2018) is not satisfying.

Like RLSVI, Bayesian DQN repeatedly takes two steps for exploration. The first step is to solve a Bayesian linear regression problem by fixing the feature representation $\phi_{\theta_{\text{hidden}}}$ (i.e., the hidden layers of the neural network). The other step is to update the hidden layers by a temporal difference learning objective; see (F.1). Specifically, it randomly samples a θ_{predict} from the posterior distribution to substantiate the prediction layer. And, the target network uses the posterior mean $\bar{\theta}_{\text{predict}}$ to substantiate the prediction layer. Finally, these two steps are performed alternatively to train the feature representation and the posterior samples.

$$\min_{\theta_{\text{hidden}}} \sum_{(s,a,r,s')\in\mathcal{D}} \left(r + \gamma Q_{\text{target}}(s') - Q_{\text{predict}}(s,a)\right)^2, \tag{F.1}$$

where

$$Q_{\text{predict}}(s,a) = \phi_{\theta_{\text{hidden}}}(s)^\top \theta_{\text{predict}}[a],$$
$$Q_{\text{target}}(s') = \max_a \left(\bar{\phi}_{\theta_{\text{hidden}}}(s')\right)^\top \bar{\theta}_{\text{predict}}[a],$$

where $x[i]$ means the i-th element of the vector x and $\bar{\phi}_{\theta_{\text{hidden}}}$ is a delayed copy of $\phi_{\theta_{\text{hidden}}}$.

Note that Bayesian DQN is not evaluated in all Atari tasks. Despite the empirical results on several tasks shown in (Azizzadenesheli et al., 2018), we find two bugs in the official code[16], which makes the conclusions in (Azizzadenesheli et al., 2018) less reliable. Firstly, the authors wrongly implement the posterior sampling procedure; see Listing 1. With the authors' implementation, the code re-samples a posterior sample (i.e., a Q-value function) for each stage/period. Clearly, this is against the algorithm description in (Azizzadenesheli et al., 2018) and violates the principle of commitment discussed in Section 4.4. The second issue is the implementation of the Bayesian linear regression update. The authors seem to additionally multiply the prior noise scale "sigma" in line 9 of Listing 2, which is different from the closed-form solution in Appendix C.1.

```
1 if frame_counter % opt.f_sampling:
2     E_W_ = sample_W(E_W,Cov_W_decom)
```

Listing 1: Bug 1 of Bayesian DQN Code.

```
1 def BayesReg(phiphiT,phiY,alpha,batch_size):
2     """
3     Some missing code to update the feature covariance matrix phiphiT and
       the inner product between the feature phi and the label Y.
4     """
5     for i in range(num_action):
6         inv = np.linalg.inv((phiphiT[i]/sigma_n + 1/sigma*eye).asnumpy())
7         E_W[i] = nd.array(np.dot(inv,phiY[i].asnumpy())/sigma_n, ctx =
     opt.ctx)
8         Cov_W[i] = sigma * inv
9     return phiphiT,phiY,E_W,Cov_W
```

Listing 2: Bug 2 of Bayesian DQN Code.

In addition to the above two bugs, Bayesian DQN updates the posterior distribution with the batch size of 100K, which is very computationally inefficient. More concretely, the procedure requires at least 20 days to run even though the above two bugs are fixed. This computation burden is unbearable like RLSVI as discussed in Section 1.

[16] https://github.com/kazizzad/BDQN-MxNet-Gluon

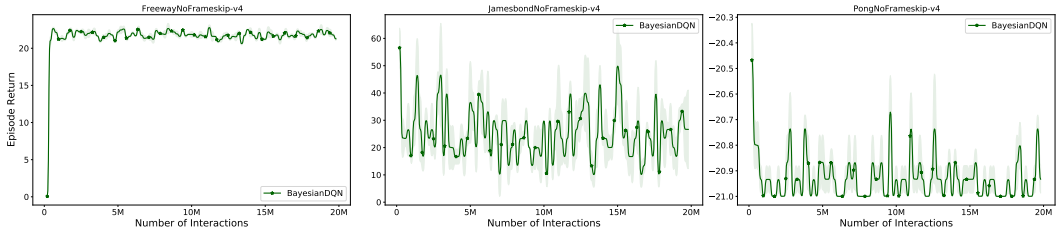
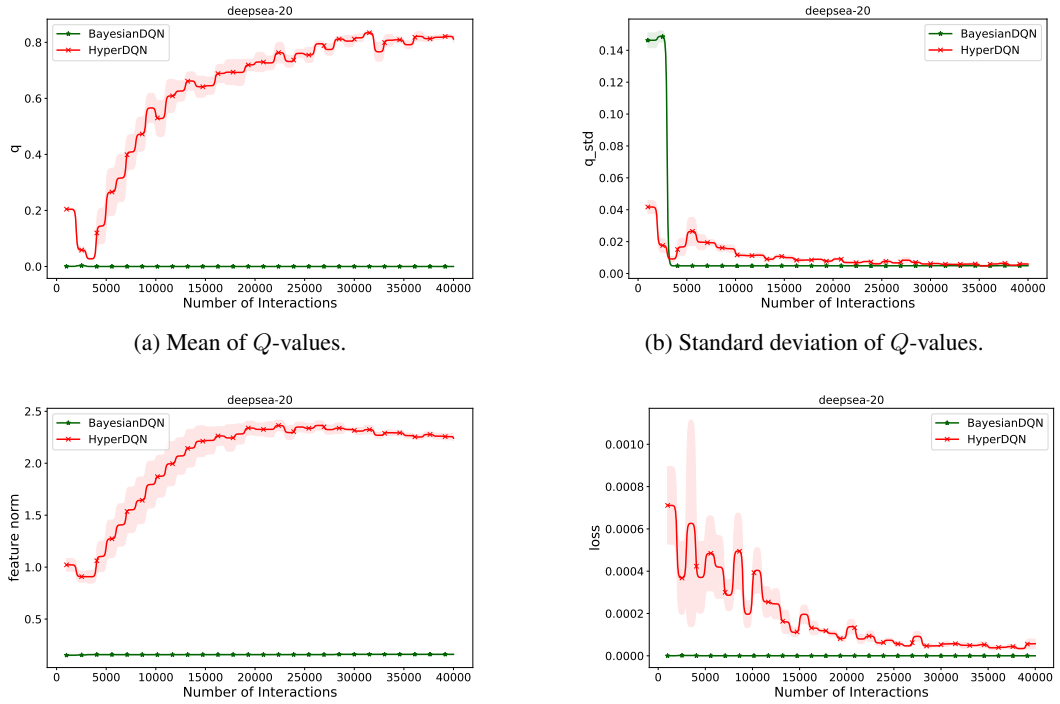


Figure 18: The performance of Bayesian DQN on three Atari tasks.

To investigate whether the extension Bayesian DQN is applicable, we fix the above two bugs and use a batch size of 2560 to perform the Bayesian linear regression update. The performance of Bayesian DQN on Atari is displayed in Figure 18. It turns out Bayesian DQN does not work except for FreewayNoFrameskip-v4. Furthermore, the performance of FreewayNoFrameskip-v4 is very poor (refer to the performance of baselines in Table 5).

To analyze why Bayesian DQN fails, we investigate the training logs on the deep sea task. We choose this toy task because we can fully understand the algorithmic behaviors on the deep sea task. The experimental results are shown in Figure 19, where we use our approach HyperDQN as a reference. The analysis is given in the following.



(a) Mean of Q-values.

(b) Standard deviation of Q-values.

(c) Feature norm.

(d) Training loss.

Figure 19: Empirical study of Bayesian DQN on deep sea.

- First, from Figure 19a, we observe that Bayesian DQN could *not* find the optimal Q-value, indicating its performance is poor though this task is not very hard (i.e., the height and width of the deep sea is 20, refer to Figure 5a).
- Second, Figure 19b suggests after the initialization, the standard deviation of the Q-value functions generated by Bayesian DQN is large (the standard deviation is a proxy for the epistemic uncertainty). However, it suddenly shrinks to a very small value. The time slot of this event is about 2560, which is exactly the first update time of Bayesian DQN (recall that the batch size of Bayesian DQN is 2560). In contrast, HyperDQN could update its

confidence during the interaction; that is, the standard deviation diminishes as more data is collected. The analysis for the sudden change in Bayesian DQN is analyzed below.

- Third, we find that the feature norm (≈ 0.1603) does not change too much over iterations. This implies the feature representation remains as a raw feature outputted by a randomly initialized neural network even though more data is gathered. On the other hand, HyperDQN could learn a task-relevant feature from the interaction.
- Finally, we can explain the failure reason of Bayesian DQN by the empirical loss shown in Figure 19d. After the initialization, the feature norm of the Bayesian DQN is small as displayed in Figure 19c. As a result, the posterior covariance is small (which could be verified by the result in Figure 19b). Accordingly, the TD loss is small and the feature representation does not change too much. This implies the Bayesian DQN goes to a "dead lock", in which its epistemic uncertainty measure (by the posterior distribution) is poor and the feature cannot be updated.

F.2 Trainability Issues of HyperDQN

In this part, we continue the discussion of the architecture design in Section 4.1. In particular, we illustrate why the original architecture in (Dwaracherla et al., 2020) is not suitable for RL tasks.

As we have argued in Section 4.1, the main challenge of applying the architecture in (Dwaracherla et al., 2020) is parameter initialization. To illustrate this issue, recall that we use a two-layer MLP neural network with the width of 64 for the deep sea task. Suppose we use the default PyTorch initialization[17] (i.e., sampling from the uniform distribution $U[-1/\sqrt{d_{i-1}}, 1/\sqrt{d_{i-1}}]$), which is also known as "$1/\sqrt{f_{\text{in}}}$" with f_{in} being the input dimension of the i-th layer. We list the parameter information in Table 6 (recall that the state dimension is 2500 and the action dimension is 2 for the deep sea task). We see that the expected magnitude of the parameter differs over layers. In particular, the desired magnitude is quite small compared with the magnitude of the output of the hypermodel (≈ 1). As a result, if we directly apply the hypermodel for all layers of the base model, the input signal over layers explodes and the gradient is quite large. Thus, training fails when we use the architecture in (Dwaracherla et al., 2020). Furthermore, this parameter initialization issue becomes more severe when we use deep convolution neural networks for Atari and SuperMarioBros tasks.

Table 6: Information about the base model used for deep sea.

	Shape ($f_{\text{in}}, f_{\text{out}}$)	Expected Magnitude of Initialization ($1/\sqrt{f_{\text{in}}}$)
Layer 1	(2500, 64)	1/50
Layer 2	(64, 64)	1/8
Layer 3	(64, 2)	1/8

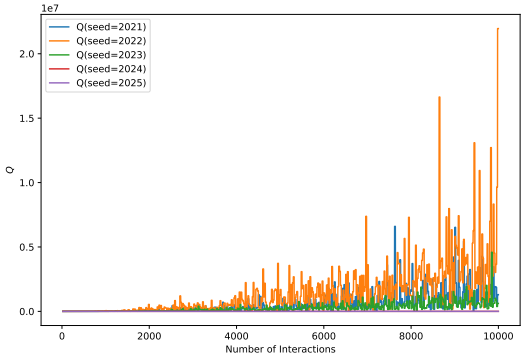


Figure 20: Q-values of the direct extension of (Dwaracherla et al., 2020) on deep sea.

Readers may argue that we can normalize the output of the hypermodel by dividing proper normalization constants for each layer. Unfortunately, we find this variant does not work, either. We provide the numerical evidence in Figure 20. We observe that Q-values are regular at the initial stage by

[17] https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

this trick but they still explode after some iterations (i.e., the magnitude could be 10^7). That is, even though this trick alleviates this issue but the landscape of the normalized objective function is still ill-condition. Combined with the changing target (due to temporal difference learning) in RL rather than the fixed target (in bandit), the instability occurs as shown in Figure 20.

In summary, the architecture in (Dwaracherla et al., 2020) is hard to be applied for deep RL due to the parameter initialization and training instability issues.

F.3 Why ϵ-greedy Is Commonly Used in Many Efficient Algorithms?

We realize many efficient algorithms (Osband et al., 2016a; Rashid et al., 2020; Bai et al., 2021) still use ϵ-greedy in practice. However, we argue that ϵ-greedy is not efficient since it cannot write off sub-optimal actions after experimentation. In addition, combining with ϵ-greedy violates the principle of commitment as discussed in Section 4.4. We conjecture existing practical algorithms combine their exploration strategies with ϵ-greedy mainly due to imperfect imitation of theoretical algorithms, which is analyzed below.

On the one hand, we believe BootDQN (Osband et al., 2016a) uses ϵ-greedy is to improve the diversity since finite ensembles could offer limited diverse action sequences. For instance, if the number of actions is larger than the number of ensembles, BootDQN cannot produce all possible action sequences to explore at the initial stage. In this case, ϵ-greedy is somehow helpful. Another reason is based on historical development. That is, BootDQN just follows the default setting of DQN (Mnih et al., 2015). In Figure 14, we see that the variant that drops the ϵ-greedy could be better than the original implementation of BootDQN in some tasks.

On the other hand, there are many practical issues to implement OFU-based algorithms in real applications, as discussed in Section 3. In fact, directly adding an exploration bonus could yield unexpected outcomes. Let us (re-)illustrate this using the example of the deep sea (shown in Figure 5a). Imagine the agent visits a certain path to collect the data; for instance, the red path in Figure 21. With the experience replay buffer, the agent can only update the Q-value function with state-action pairs from the visited path. By exploration bonus, such state-action sequences would have larger Q-values. During the next episode, the agent would repeat the same action sequences by following the greedy policy. It turns out that theoretical algorithms like (Jin et al., 2018) do not have such an issue. This is because theoretically, the algorithm can use an optimistic initialization. With the optimistic initialization, the algorithm instead would select other actions rather than visited actions. In practice, however, it is not easy to obtain an optimistic initialization with deep neural networks (Rashid et al., 2020). Again, ϵ-greedy could somehow avoid this issue.

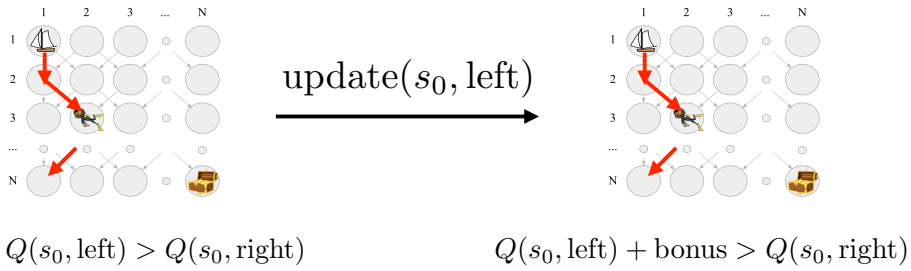


Figure 21: Illustration for the initialization issue of OFU-based algorithms. After the initialization, action "left" dominates at state s_0. After the experience replay buffer, action "left" dominates again at state s_0. This issue is caused by the pessimistic initialization by neural network (Rashid et al., 2020).

In addition to the initialization issue, the uncertainty should propagate over stages/periods with exploration bonus in a backward manner. In (Jin et al., 2018), this operation can be achieved by dynamic programming. In contrast, practical algorithms randomly draw mini-batch from the experience buffer to update. This indeed may not properly propagate the uncertainty. As a result, the Q-values are not always optimistic, and induced action sequences may not be diverse. This phenomenon is pointed out in (Bai et al., 2021). Again, ϵ-greedy may be helpful to address this issue.

F.4 COMPARISON WITH NOISYNET

In this part, we explain the differences between HyperDQN and NoisyNet in detail. The following discussion aims to provide insights about *what HyperDQN can achieve while NoisyNet cannot*. Note that we are by no means criticizing NoisyNet. Instead, NoisyNet is simple and has strong empirical performance. We hope the above discussion could provide intuitions (or explanations) why NoisyNet successes or fails under different cases.

- First, as remarked in (Fortunato et al., 2018), NoisyNet is not ensured to approximate the posterior distribution of parameters. Therefore, NoisyNet is not a typical Thompson sampling based algorithm. The implication is that we cannot use the well-known theory (Osband et al., 2013; 2019; Zanette et al., 2020) to analyze NoisyNet.
- Second, NoisyNet re-samples a new policy every time step (Line 5 of Algorithm 1 in (Fortunato et al., 2018)). Consequently, it does not implement deep exploration like RLSVI (see (Osband, 2016b, Section 4.1)). This explains the empirical result that NoisyNet can not solve the deep sea task when the problem size is large than 20 (see Figure (Osband et al., 2018, Figure 9)). Instead, BootDQN and HyperDQN can solve the deep sea task even if the problem size is large than 20 as shown in Appendix E.3.
- Third, NoisyNet does not have the mechanism of "prior" (Osband et al., 2018) even though it is randomized. As a consequence, NoisyNet cannot leverage an informative prior to accelerate exploration when such information is available. In contrast, HyperDQN can achieve this goal as discussed in Appendix F.6.

Next, we conduct experiments to empirically compare NoisyNet and HyperDQN. Since there is no public implementation of NoisyNet, we reproduce NoisyNet according to the description in (Fortunato et al., 2018, Algorithm 1) and the hyper-parameters in (Fortunato et al., 2018, Section 3.2). The neural network architecture and learning parameters are the same as the ones described in Appendix D. The empirical results on Atari and SuperMarioBros are shown in Figure 22 and Figure 23, respectively.

For tasks in the Atari suite, the reproduced results of Noisy basically match the results reported in (Fortunato et al., 2018, Figure 6). In particular, both methods do not work for Montezuma's Revenge. On the one hand, NoisyNet is better than HyperDQN on Beam Rider and Seaquest. On the other hand, HyperDQN outperforms NoisyNet on Pong, Venture, and Zaxxon.

For tasks in the SuperMarioBros suite, NoisyNet is better than HyperDQN on SuperMarioBros-1-1. Both methods perform well on SuperMarioBros-2-3. On the hard exploration task SuperMarioBros-1-3 and SuperMarioBros-2-2, HyperDQN performs better than NoisyNet since these tasks require deep exploration while NoisyNet is hard to generate "far-sighted" behaviors as we have discussed.

In summary, HyperDQN is slightly better than NoisyNet. We are still running more experiments and we will provide full results in the later revision.

F.5 EXTENSION TO CONTINUOUS CONTROL

In this part, we briefly discuss how to extend the idea in this paper for continuous control tasks. We also present some preliminary results to support this direction.

Following the idea presented in Section 4.3, we should leverage the hypermodel to capture the posterior distribution of the Q-value function. Considering the standard actor-critic methods, we can replace the vanilla critic with the one that is built by the hypermodel. In particular, we replace the last layer of the critic with a hypermodel. In this way, each z corresponds to a specific critic from the posterior distribution. To perform policy optimization, we notice that each actor (i.e., a policy network) should be greedy to a specific critic with the same index z. To this end, the last layer of the actor network is also built by a hypermodel. The architecture design is illustrated in Figure 24.

Let $(\theta^{\text{critic}}_{\text{hidden}}, \nu^{\text{critic}})$ be the parameter for the critic network and $(\theta^{\text{actor}}_{\text{hidden}}, \nu^{\text{actor}})$ be the parameter for the actor network. Following the optimization framework in (Lillicrap et al., 2016; Haarnoja et al., 2018), the training objective for the actor network is:

$$\max_{\theta^{\text{actor}}_{\text{hidden}}, \nu^{\text{actor}}} \sum_{z \in \widetilde{\mathcal{Z}}} \sum_{s \in \widetilde{\mathcal{D}}} Q_{\theta^{\text{critic}}_{\text{hidden}}, \nu^{\text{critic}}}(s, a_z, z), \quad \text{with } a_z \sim \pi_{\theta^{\text{actor}}_{\text{hidden}}, \nu^{\text{actor}}}(s; z), \tag{F.2}$$

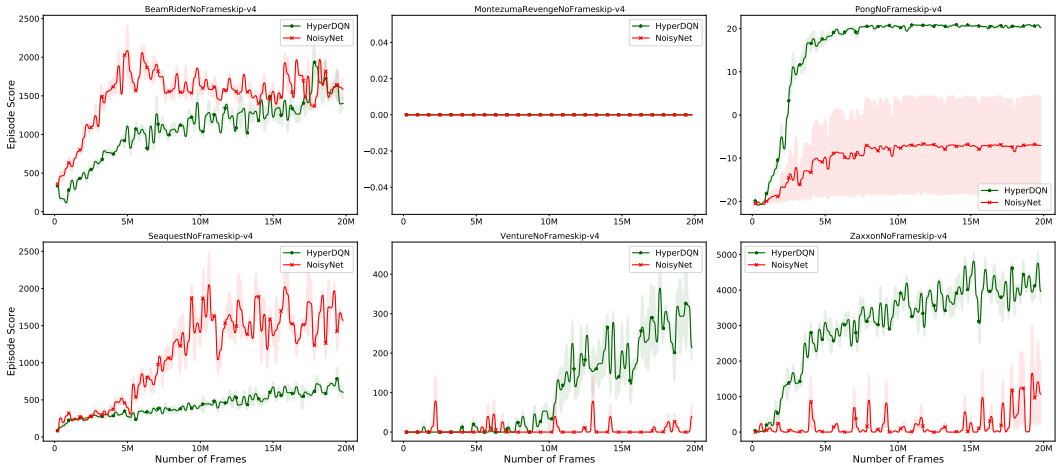


Figure 22: Comparison between NoisyNet and HyperDQN on Atari.

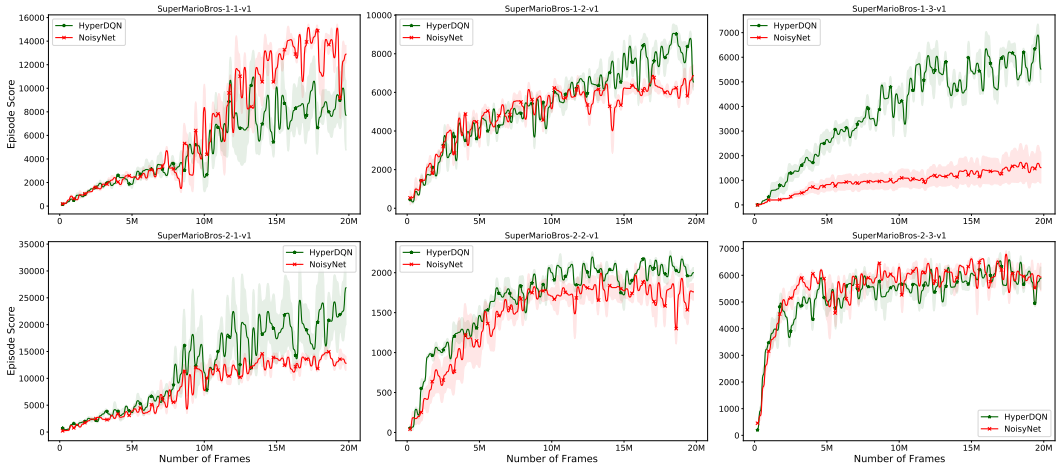


Figure 23: Comparison between NoisyNet and HyperDQN on SuperMarioBros.

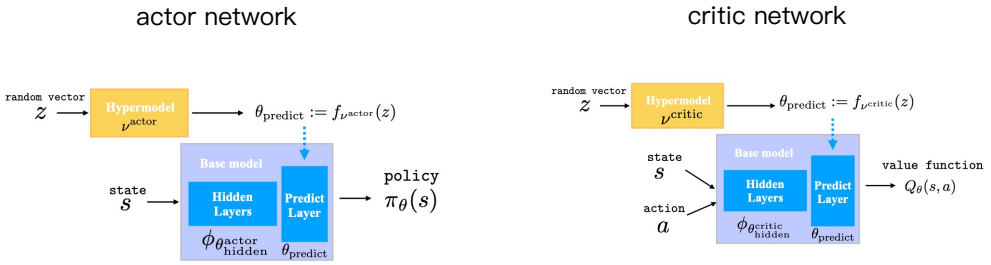


Figure 24: Illustration for HAC.

where $Q_{\theta^{\text{critic}}_{\text{hidden}},\nu^{\text{critic}}}$ is the critic network and $\pi_{\theta^{\text{actor}}_{\text{hidden}},\nu^{\text{actor}}}$ is the actor network. In particular, objective (F.2) states that we should sample actions from the actor network to maximize the Q-value function. The difference with the traditional actor-critic methods is that we sample many actor networks (indexed by z) to optimize.

Similarly, the training objective for the critic network is:

$$\min_{\theta^{\text{critic}}_{\text{hidden}},\nu^{\text{critic}}} \sum_{z\in\widetilde{\mathcal{Z}}} \sum_{(s,a,r,s',\xi)\in\widetilde{\mathcal{D}}} \left(Q_{\theta^{\text{critic}}_{\text{hidden}},\nu^{\text{critic}}}(s,a,z) - \sigma_w \xi^\top z - \left(r + \gamma Q_{\bar{\theta}^{\text{critic}}_{\text{hidden}},\bar{\nu}^{\text{critic}}}(s',a'_z,z)\right)\right)^2, \tag{F.3}$$

with $a'_z \sim \pi_{\theta^{\text{actor}}_{\text{hidden}},\nu^{\text{actor}}}(s',z)$. In (F.3), $Q_{\bar{\theta}^{\text{critic}}_{\text{hidden}},\bar{\nu}^{\text{critic}}}$ is the target network. The goal of (F.3) is to perform the temporal difference learning to optimize the critic. Note that the prior regularization term is not appeared in (F.3) for easy presentation.

We call the above method as HyperActorCritic(HAC). Its implementation is outlined in Algorithm 3, which shares many features with Algorithm 2.

Algorithm 3 HyperActorCritic(HAC)

```
for episode k = 0, 1, 2, ··· do
    generate a random vector z ~ N(0, I).                                  ▷ Sampling
    instantiate an actor π_θ(·; z).
    for stage t = 0, 1, 2, ··· , T − 1 do                                  ▷ Interaction
        observe state s_t.
        sample the action a_t ~ π_θ(s_t; z).
        receive the next state s_{t+1} and reward r(s_t, a_t).
        sample ξ uniformly from unit hypersphere.
        store (s_t, a_t, r, ξ, s_{t+1}) into the replay buffer D.
        agent step n ← n + 1.
        if mod (agent step n, train frequency M) == 0 then                 ▷ Update
            sample a mini-batch D̃ of (s, a, r, ξ, s') from the replay buffer D.
            sample N random vectors z'_i: Z̃ = {z'_i}_{i=1}^N.
            optimize the critic network using the loss function (F.3) with D̃ and Z̃.
            optimize the actor network using the loss function (F.2) with D̃ and Z̃.
            update the critic target network with exponential moving average.
        end if
    end for
end for
```

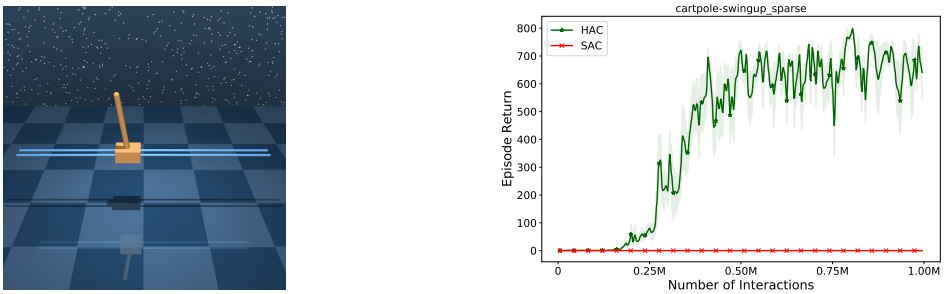
Now, we consider the Cart Pole task from `dm_control`[18] (Tunyasuvunakool et al., 2020) as a testbed; see Figure 25a . Detailed environment information is provided as follows: the dimension of the state space is 5, the dimension of the action space is 1, the planning horizon is 1000 and the reward is between 0 and 1. This environment is a standard platform to test the exploration efficiency for continuous control algorithms because the agent can only obtain $+1$ reward when it succeeds and obtain 0 reward otherwise.

We compare our extension (HAC) with the strong baseline SAC (Haarnoja et al., 2018). Even though the policy entropy reward is used to guide exploration, SAC does not have the epistemic uncertainty qualification like HAC. As a result, we expect SAC does not perform well for Cart Pole. The numerical result is displayed in Figure 25b. We see that HAC outperforms SAC on this hard exploration task.

F.6 When an Informative Prior is Available

In this part, we briefly discuss the role of prior value functions in our formulation. In particular, we argue that if an informative prior value function is available, the exploration efficiency can be significantly improved.

[18] https://github.com/deepmind/dm_control



(a) Illustration for Cart Pole.

(b) Episode return.

Figure 25: Empirical study on the hard exploration Cart Pole (Tunyasuvunakool et al., 2020).

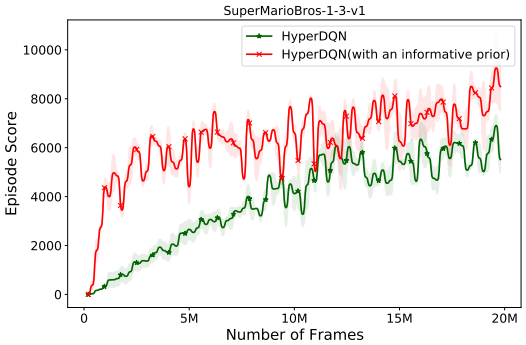


Figure 26: Ablation study about the prior Q-value function on SuperMarioBros.

Here we show the learning curve with an informative prior in Figure 26. In particular, this informative prior value function is obtained from the pre-trained model after 20M frames. We see that this informative prior value function improves efficiency a lot. Note that the main goal of this experiment is to illustrate that an informative prior could accelerate exploration. We will consider how to automatically acquire an informative prior with human demonstrations (Hester et al., 2018; Sun et al., 2018) in the future.