Methods for Adding Explicit Uncertainty to Deep Q-Learning

Blair Bilodeau
University of Toronto
blair.bilodeau@mail.utoronto.ca

Anthony Coache
University of Toronto
anthony.coache@mail.utoronto.ca

Abstract

A successful reinforcement learning (RL) player must generalize its own experiences while considering long-term consequences as well as adequately explore the environment. While existing work has provided methods to capture the necessary uncertainty for exploration using Bayesian methods, there is not a principled method to do so with arbitrary prior mechanisms. In this work, we formalize a framework for adding explicit uncertainty into deep RL algorithms, highlight potential shortcomings of an existing ensemble algorithm (BootDQN+prior), and propose improvements inspired by both statistical optimization and the sequential decision making literature. We demonstrate that our proposed algorithms obtain smaller regret on classic benchmark RL problems.

1 Introduction

An essential problem in statistical learning is quantifying uncertainty for both the data-generating mechanism and the player. In reinforcement learning (RL), these two sources of uncertainty can have substantial effects on performance. First, the inherent randomness of the environment may mean that algorithms optimized for "on-average" performance can have prohibitively large variance, and consequently perform sub-optimally with significant probability. Second, it is well-known that the player must balance *exploiting* actions that they have already seen to be good with *exploring* actions whose quality is unknown. Recent literature [18, 25] has observed that *deep Q-network* approaches, which replace traditional tabular RL methods with neural network approximations, can be overly confident, fail to capture the player's uncertainty, and consequently under explore. To address this, Osband et al. [25] propose that introducing explicit sources of uncertainty can lead to improved performance, which they argue heuristically should resemble Bayesian statistical techniques and demonstrate empirically with additive "prior" functions.

In this work, we extend the ideas from Osband et al. [25] in two directions. First, we present alternatives to the additive prior structure that use more classical notions of uncertainty, inspired by classical Bayesian neural networks and regularized maximum likelihood estimators (MLEs). While these techniques suffer empirically from many of the same convergence issues as Bayesian neural networks, we provide a formalization of deep RL with explicit uncertainty, and provide a generic theoretical structure and empirical implementation for this approach. Second, we argue that the ensemble approach can be combined with traditional sequential decision making algorithms to tune the amount of explicit exploration, and demonstrate the benefits of these approaches empirically. We measure our success based on the performance of these modified algorithms on some classic benchmark RL problems in the literature, and also provide some novel extensions of these settings to have non-stationary behaviour.

We acknowledge that our novel contributions are not sufficient for publication, but only for a class project, as we rely heavily on the BootDQN+prior framework [25] and reuse some of their code. Section 2 introduces a unified notation for generic RL problems, while Section 3 uses this notation to formalize deep Q-learning and the explicit noise structure of Osband et al. [25]. In Section 4, we formalize how one can use traditional techniques for adding explicit uncertainty in deep Q-learning, and present our novel ideas for adding aggregation to ensemble techniques. Section 5 illustrates the performance of these modified algorithms on some benchmark stationary RL problems in the literature, as well as our novel non-stationary variants of them. Finally, we discuss related work in Section 6 and then our work's limitations and possible extensions in Section 7.

Final project for STA4273 at the University of Toronto, taught by Chris Maddison.

2 Notation and Problem Setup

In this section, we introduce the necessary theoretical background for deep RL methods. Suppose there exists an underlying probability space on which all random variables are defined. For any arbitrary measurable spaces $\mathcal{X}$ and $\mathcal{Y}$, let $\mathcal{K}(\mathcal{X}, \mathcal{Y})$ denote the set of all probability kernels; that is, all $\sigma(\mathcal{X})$-measurable maps to $\mathcal{P}(\mathcal{Y})$, the space of probability measures on $\sigma(\mathcal{Y})$. Let $[n] = \{1, \dots, n\}$ for any $n \in \mathbb{N}$.

Let $\mathcal{S}$ and $\mathcal{A}$ be arbitrary, finite state and action spaces respectively, and let $\mathcal{R} \subseteq \mathbb{R}$ be a reward space. A *(reinforcement learning) game* lasts for L *episodes*, each consisting of T_ℓ *rounds* for $\ell \in [L]$, where T_ℓ may be a random variable but is finite almost surely. At each round t during episode ℓ, the player begins in state $s_{t-1}^\ell \in \mathcal{S}$, takes an action a_t^ℓ, receives a reward $r_t^\ell \in \mathcal{R}$, and the next state s_t^ℓ is revealed.

Define $\mathcal{H} = \mathcal{A} \times \mathcal{R} \times \mathcal{S}$. The *episode history* up to round t in episode ℓ is the $\sigma(\mathcal{S} \times \mathcal{H}^{t-1})$-measurable random variable $h_t^\ell = (s_0^\ell) \cup (a_s^\ell, r_s^\ell, s_s^\ell)_{s \in [t-1]}$, while the *(game) history* is the $\sigma(\mathcal{H}^{\sum_{l=1}^{\ell-1} T_l})$-measurable random variable $h^{1:\ell} = \bigcup_{l \in [\ell-1]} h_{T_{l+1}}^l$. For a fixed $\hat{\Pi} \subseteq \prod_{t \in \mathbb{N}} \mathcal{K}(\mathcal{S} \times \mathcal{H}^{t-1}, \mathcal{A})$, an *(allowable) player strategy* is any sequence of probability kernels $\hat{\pi} = (\hat{\pi}_t^\ell)_{\ell \in [L], t \in \mathbb{N}} \in \hat{\Pi}$. Further, letting $\hat{\mathscr{P}} \subseteq \prod_{\ell \in [L]} \mathcal{K}(\mathcal{H}^{\sum_{l=1}^{\ell-1} T_l}, \hat{\Pi})$, an *(allowable) player algorithm* is any sequence of kernels $\hat{\mathfrak{a}} = (\hat{\mathfrak{a}}_\ell)_{\ell \in [L]} \in \hat{\mathscr{P}}$. Similarly, for $\Pi \subseteq \prod_{\ell \in [L]} \Big[\mathcal{K}(\mathcal{H}^{\sum_{l=1}^{\ell-1} T_l}, \mathcal{S}) \times \prod_{t \in \mathbb{N}} \mathcal{K}(\mathcal{S} \times \mathcal{H}^{t-1+\sum_{l=1}^{\ell-1} T_l} \times \mathcal{A}, \mathcal{R} \times \mathcal{S}) \Big]$, an *(allowable) data-generating mechanism* is any sequence of probability kernels $\pi = (\pi_t^\ell)_{\ell \in [L], t \in \mathbb{N} \cup \{0\}} \in \Pi$.

Together, a player algorithm $\hat{\mathfrak{a}}$ and a data-generating mechanism π induce a joint law over $h^{1:L+1}$, which we denote expectation under using $\mathbb{E}_{\hat{\mathfrak{a}}, \pi}$. The *value function* $V_{\hat{\mathfrak{a}}, \pi}^\ell(s) : \mathcal{S} \to \mathbb{R}$ on episode ℓ is

$$V_{\hat{\mathfrak{a}}, \pi}^\ell(s) = \mathbb{E}_{\hat{\mathfrak{a}}, \pi} \left[\sum_{t=1}^{T_\ell} \gamma^t r_t \;\middle|\; s_0 = s, h^{1:\ell} \right],$$

where $\gamma \in (0, 1]$ is a *discount factor* for future rewards. When clear, we will let $\hat{\pi}^\ell = \hat{\mathfrak{a}}(h^{1:\ell})$. Additionally, for a player algorithm with $\hat{\mathfrak{a}} \equiv \hat{\pi}$, we denote expectation by $\mathbb{E}_{\hat{\pi}, \pi}$ for simplicity. Then, the *regret* of the player algorithm is

$$\mathcal{R}_{\hat{\mathfrak{a}}, \pi}(L) = \mathbb{E}_{\hat{\mathfrak{a}}, \pi} \sum_{\ell=1}^{L} \sup_{\hat{\pi} \in \hat{\Pi}} \Big[V_{\hat{\pi}, \pi}^\ell(s_0^\ell) - V_{\hat{\mathfrak{a}}, \pi}^\ell(s_0^\ell) \Big].$$

We note that the supremum over player strategies in the definition of regret is *inside* both the summation over ℓ and the expectation. An alternative, easier definition often considered in the online learning literature [6] is *pseudo-regret*, defined as

$$\overline{\mathcal{R}}_{\hat{\mathfrak{a}}, \pi}(L) = \sup_{\hat{\pi} \in \hat{\Pi}} \mathbb{E}_{\hat{\mathfrak{a}}, \pi} \sum_{\ell=1}^{L} \Big[V_{\hat{\pi}, \pi}^\ell(s_0^\ell) - V_{\hat{\mathfrak{a}}, \pi}^\ell(s_0^\ell) \Big]. \tag{1}$$

Often, reinforcement learning is restricted to the set of *Markov data-generating mechanisms* $\mathcal{M} = \mathcal{P}(\mathcal{S}) \times \mathcal{K}(\mathcal{S} \times \mathcal{A}, \mathcal{R} \times \mathcal{S})$, although the definition of performance does not require this. While the majority of our experiments are on Markov data-generating mechanisms, we briefly discuss non-stationary data-generating mechanisms in Subsection 5.3. A crucial object for RL algorithms in the Markov setting is the *Q-function*, which for any $\pi \in \mathcal{M}$, $s \in \mathcal{S}$, and $a \in \mathcal{A}$ is

$$Q_\pi^*(s, a) = \mathbb{E}_\pi \left[r_1 + \gamma \sup_{\hat{\pi} \in \hat{\Pi}} V_{\hat{\pi}, \pi}(s_1) \;\middle|\; s_0 = s, a_1 = a \right]. \tag{2}$$

Note that, since π satisfies the Markov property, every round experiences the same transition probabilities, so there is no need to track the episode number and without loss of generality it can be supposed the current round is the first of the episode.

3 Deep Reinforcement Learning

We now express existing approaches to solve RL problems with off-policy algorithms in our notation. Instead of focusing on the simplest form of Q-learning, we consider advancements in deep Q-learning, which replaces point estimates with function approximators. We also present the existing techniques to improve on these approaches by explicitly modelling uncertainty to achieve enhanced exploration.

3.1 Deep Q-Learning

Q-learning, first introduced by Watkins and Dayan [32], seeks to find the best player strategy by evaluating the Q-function Q^*_π given in Eq. (2). It is well-known [19] that for any $\pi \in \mathcal{M}$, the optimal action for the player to take when they are in state s is $a^* = \arg\max_{a\in\mathcal{A}} Q^*_\pi(s,a)$. Since Q^*_π is unknown to the player, an approximation must be used. Classical RL uses the *Bellman* equation update rule

$$\hat{Q}_{t+1}(s,a) = \mathbb{E}_\pi\Big[r_t + \gamma \max_{a'\in\mathcal{A}} \hat{Q}_t(s_{t+1},a') \mid s_t = s, a_t = a\Big],$$

which will converge in many settings [see Chapter 4.1 of 29]. Unfortunately, this is impractical for complex settings, since it also requires an accurate estimate of π.

To address this, Mnih et al. [21] introduced the *deep Q-learning* framework, which uses a neural network approximation of the Q-function rather than attempting to learn it exactly. More precisely, let θ^ℓ_t be the $\sigma(\mathcal{S}\times\mathcal{H}^{t-1+\sum_{l=1}^{\ell-1} T_l})$-measurable random variable taking values in Θ that denotes the parameters of a neural network. Then, $\hat{Q}(\cdot;\theta^\ell_t) : \mathcal{S}\times\mathcal{A}\to\mathbb{R}$ denotes the learned neural network approximation of Q^*_π after training[1] on the first $\ell-1$ episodes and the first $t-1$ rounds of episode ℓ. The player strategy is then to always play

$$a_t \sim \hat{\pi}^\ell_t(h^{1:\ell}, h^\ell_t) = \delta\left\{\arg\max_{a\in\mathcal{A}} \hat{Q}(s^\ell_t, a; \theta^\ell_t)\right\},$$

where δ denotes a point-mass measure.

For training purposes, we measure the quality of θ by the expected square loss

$$\mathcal{L}(\theta) = \mathbb{E}_{\hat{\pi},\pi}\Big(Q^*_\pi(s,a) - \hat{Q}(s,a;\theta)\Big)^2.$$

Ideally, we could learn θ using a supervised learning approach; for example, simple empirical risk minimization within a single episode would correspond to

$$\theta_t = \arg\min_{\theta\in\Theta} \sum_{s=1}^{t-1} \Big(Q^*_\pi(s_s,a_s) - \hat{Q}(s_s,a_s;\theta)\Big)^2.$$

Unfortunately we never actually observe Q^*_π, and consequently we instead aim to minimize

$$\hat{\mathcal{L}}_t(\theta) = \mathbb{E}_{\hat{\pi},\pi}\Big(\mathbb{E}_\pi\Big[r + \gamma\max_{a'\in\mathcal{A}} \hat{Q}(s',a';\theta_{t-1}) \;\Big|\; s,a\Big] - \hat{Q}(s,a;\theta)\Big)^2. \tag{3}$$

It is well-known [20] that Eq. (3) has the gradient

$$\nabla_\theta \hat{\mathcal{L}}_t(\theta) = \mathbb{E}_{\hat{\pi},\pi}\Big[\Big(r + \gamma\max_{a'\in\mathcal{A}} \hat{Q}(s',a';\theta_{t-1}) - \hat{Q}(s,a;\theta)\Big)\nabla_\theta \hat{Q}(s,a;\theta)\Big]. \tag{4}$$

This form makes it clear that by sampling from the current player strategy and data-generating mechanism we can obtain an unbiased gradient estimate, which we use to update θ. Finally, we note that due to stability issues θ_{t-1} is often replaced with $\tilde{\theta}^\ell$, which is a more slowly-updated parametrization of the neural net approximation, usually referred to as the *target network* [31].

3.2 Prior Mechanisms

While deep Q-learning provides a solution to the issue of Q^*_π being difficult to approximate, it still suffers from the fact that it plays a deterministic algorithm. Specifically, the player will always select the action that maximizes their Q-function estimate, which could lead to compounding bias [12]. The naive solution to this is to follow an ε-greedy strategy, sampling a_t uniformly with some small probability ε and otherwise playing the argmax. However, Osband et al. [25] demonstrate empirically that this still does not lead to sufficient exploration, and consequently more sophisticated techniques are required.

The idea is to introduce additional randomness into the player strategy in order to lead to more exploration and reduce overfitting to potentially misleading data. Osband et al. [25] do this using an *ensemble* method with so-called *additive priors*. Their algorithm, BootDQN+prior, is illustrated in Fig. 1. Rather than having a single estimate of the Q-function parametrized by θ^ℓ_t, they maintain $\mathcal{J}$ estimates $({}_{(j)}\theta^\ell_t)_{j\in[\mathcal{J}]}$. At each round (or at the start of each episode), they first sample $j^\ell_t \sim \mathrm{Unif}([\mathcal{J}])$, and then play $\arg\max_{a\in\mathcal{A}} \hat{Q}(s^\ell_t, a; {}_{(j^\ell_t)}\theta^\ell_t)$. We examine in Subsection 4.2 alternatives to uniformly sampling the ensemble index, but first we discuss how the ensembles are trained.

[1] The actual training frequency and batch size of data used for each training step may be chosen as hyperparameters.

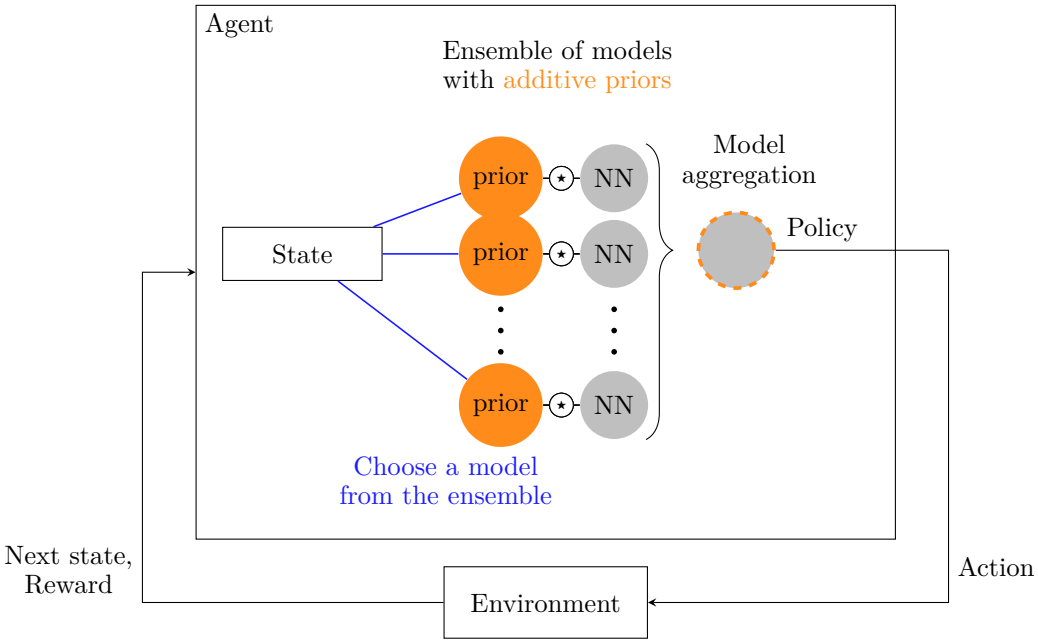

Figure 1: BootDQN+prior algorithm. We highlight the selection of models from the ensemble (blue) and the choice of prior mechanism (orange).

In order to avoid the ensemble estimates quickly converging to all being the same, which would defeat the purpose of an ensemble, each element is trained on a different, noisy version of the observed data. The noise is *structured*, which means that it changes based on the covariate information. Specifically, an ensemble is defined along with the *prior functions* $(P_j)_{j\in[\mathcal{J}]} \subseteq \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$. Then, each of the ensemble parameters are updated using a noisy version of the gradient in Eq. (4):

$$\left(r_t + \gamma \max_{a'\in\mathcal{A}} \left\{\hat{Q}(s_{t+1}, a'; \tilde{\theta}^\ell) + P_{\tilde{j}^\ell}(s_{t+1}, a')\right\} - \hat{Q}(s_t, a_t; {}_{(j)}\theta_t^\ell) - P_j(s_t, a_t)\right) \nabla_\theta \hat{Q}(s_t, a_t; {}_{(j)}\theta_t^\ell).$$

Osband et al. [25] implement these prior functions using a randomly initialized neural network that is not updated with the data.

4 Novel Uncertainty Structures

In this section, we describe the new types of uncertainty we consider to provide a more flexible prior mechanism. While these types of uncertainty have all been considered in some form by previous works, to the best of our knowledge they have not been combined in this way for deep RL problems.

4.1 Additional Bayesian Uncertainty

We use two additional techniques to introduce randomness into the Q-function estimation via a "prior". Both of these use the tools of *Bayesian RL*; the first induces a distribution over the parameters of the neural network, while the second induces a distribution over the Q-values themselves. Naturally, the use of neural nets makes it infeasible to find a closed-form for the posterior distributions. The first approach injects noise in neural network parameters and updates them by gradient descent, whereas the second uses regularized maximum likelihood estimation to efficiently obtain approximations of the distributions.

The first type of uncertainty considered is by introducing randomness directly in the neural network parameters instead of the data, as proposed for NoisyNets by Fortunato et al. [10], where they add parametric noise to the weights and biases of any neural network structure. It is worth noting that this approach differs from variational inference techniques for Bayesian neural networks [e.g., 4] since it does not maintain an explicit distribution over the neural network parameters.

Suppose we have a layer of a neural network with p inputs and q outputs, with input arguments $x \in \mathbb{R}^p$, the weight matrix $w \in \mathbb{R}^{q\times p}$, the bias $b \in \mathbb{R}^q$ and an activation function $a(\cdot)$. We can represent the noisy version of any layer by

$$y = a\left(\left(\mu^w + \sigma^w \odot \varepsilon^w\right) x + \left(\mu^b + \sigma^b \odot \varepsilon^b\right)\right), \tag{5}$$

where ε is a vector of white noise and $\odot$ represents the element-wise multiplication. The parametrization θ is then given by $\theta = \mu + \sigma \odot \varepsilon$ with trainable parameters $\mu = (\mu^w, \mu^b)$ and $\sigma = (\sigma^w, \sigma^b)$. The loss function has the same form as Eq. (3), but it is wrapped by an additional expectation over the noise, which makes NoisyNets easily adaptable to any deep RL algorithms. We focus on factorized Gaussian noise (see Section 3(b) of [10]) to characterize the noise distribution, which corresponds to independent noise per each input/output. Specifically, if $\varepsilon_i,\ i \in [p]$ and $\varepsilon_j,\ j \in [q]$ are independent standard Gaussian variables, then

$$\varepsilon^w_{i,j} = f(\varepsilon_i) f(\varepsilon_j),$$
$$\varepsilon^b_j = f(\varepsilon_j),$$

where f is any real-valued function.

The other type of uncertainty that we introduce is by explicitly modelling a conditional distribution over Q-values. Formally, let $\mathscr{D} \subseteq \mathcal{P}(\mathbb{R})$ be a family of real-valued distributions. For concreteness, we suppose each $D \in \mathscr{D}$ is uniquely defined by some parameter vector, and we consequently use D to also denote this parameter vector. While a deep Q-network approximation to the value function is a map $\hat{Q} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, we instead use the neural network to learn $\hat{K} : \mathcal{S} \times \mathcal{A} \to \mathscr{D}$, and then sample $\hat{Q}(s,a) \sim \hat{K}(s,a)$ as a random approximation to $Q^*_\pi(s,a)$.

To learn the parametrization θ of the neural net $\hat{K}$, it is no longer suitable to simply use square loss since we assume a probabilistic model over the Q-values; instead, we use log-loss since it is a proper scoring rule for distributions. Further, by defining a prior $P \in \mathcal{P}(\Theta)$, we can learn the full posterior over θ's. However, the limited observations and large state space can make it extremely unlikely that this posterior will have suitably converged to provide useful credible intervals. As a simplification, we instead choose θ to be the posterior mode, which for certain canonical priors is equivalent to using the regularized maximum likelihood loss function. Thus, we use gradient descent on the loss function

$$D^{s,a}_\theta = \hat{K}(s,a;\theta) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}, \theta \in \Theta;$$
$$\hat{Q}^*_t(s,a) = \mathbb{E}_\pi\Big[r + \gamma \max_{a' \in \mathcal{A}} \mathbb{E}[D^{s',a'}_{\theta_{t-1}}] \ \Big|\ s, a \Big] \quad \forall s \in \mathcal{S}, a \in \mathcal{A};$$
$$\hat{\mathcal{L}}_t(\theta) = \mathbb{E}_{\hat{\pi},\pi}\Big[-\log D^{s,a}_\theta\Big(\hat{Q}^*_t(s,a)\Big)\Big] - \log P(\theta).$$

In our experiments, $\mathscr{D}$ is the set of Gaussian distributions on Q-values. Due to the small size of the action space, it is feasible to learn a dense covariance matrix, rather than only a diagonal one as is usually considered in the literature. We also choose a Gaussian prior, which is equivalent to L_2 regularization.

4.2 Model Aggregation

As discussed in Subsection 3.2, one way to increase exploration is to consider an ensemble of Q-function estimates. Current implementations sample uniformly from this ensemble on each episode to decide how to estimate the Q-function and inform the next action choice. However, while the ensemble is useful for extra exploration during early episodes and avoids potentially disastrous initializations, one would hope to be able to exploit whichever ensemble member has been the best approximation after observing data. Towards this, we introduce two novel approaches for aggregating over the ensembles to make a data-informed choice at each episode. Both methods correspond to maintaining a discrete distribution (*weights*) over the ensemble, which we denote with $w = (w_1, \ldots, w_{\mathcal{J}}) \in \Delta_{\mathcal{J}}$. The reward of playing the jth element of the ensemble for episode ℓ is denoted by $r^\ell(j)$, and its estimate (to account for unobserved values) is $\hat{r}^\ell(j)$. Note that while the use of ensembles was motivated for the additive prior type of uncertainty, we are also able to implement either of our two Bayesian uncertainty methods in ensemble form and aggregate over these ensembles.

For the first type of aggregation, we apply more traditional RL techniques by using the simplest form of *policy gradient* [30]. When choosing which element of the ensemble w to use, we are interested in prioritizing elements that led to the best regret over episodes, as opposed to using the Q-value to measure performance. Therefore, the gradient of the objective is of the form

$$\nabla_w W(w^\ell) = \mathbb{E}_{\pi, j \sim w^{\ell-1}}\Big[r(j) \nabla_w \log w^\ell(j) \Big].$$

While taking a stochastic gradient step for the policy gradient method is one way around partially observed information, we also introduce a method to use techniques from multi-armed bandits. Specifically, we determine the ensemble weights using the Exp3 algorithm [1], although in principle any bandit algorithm could be swapped in to achieve the same type of behaviour. In particular, we use the importance-weighted estimates of the reward for an ensemble member in episode ℓ defined by

$$\hat{r}^\ell(j) = \frac{r^\ell}{w^\ell(j)} \mathbb{I}\{j^\ell = j\},$$

where j^ℓ is the actual ensemble member played during the episode. The weights are then updated according to the entropy-maximizing update rule

$$w^{\ell+1}(j) = \frac{\exp\Big\{ -\lambda^{\ell+1} \sum_{l\in[\ell]} \hat{r}^l(j) \Big\}}{\sum_{j'\in[\mathcal{J}]} \exp\Big\{ -\lambda^{\ell+1} \sum_{l\in[\ell]} \hat{r}^l(j') \Big\}},$$

where $\lambda^\ell > 0$ is the learning rate on the ℓth episode. We also implement a lower-variance ε-greedy version of Exp3, and suggest that an interesting alternative to test is the Tsallis-INF algorithm [34].

5 Experiments

In this section, we run a twofold comparison study on two benchmark RL tasks, Deepsea and Cartpole. We first compare the different prior mechanisms discussed in Subsection 4.1: the original approach from [25] (*NeuralNet*), *NoisyNet* [10], and the regularized MLE over Gaussian distributions (*GaussianNet*). We also evaluate these modified algorithms based on the aggregation methods discussed in Subsection 4.2: uniform sampling (*Unif*), policy gradient (*PG*), and the multi-armed bandit approach (*Exp3*). We measure the regret $\mathcal{R}_{\hat{\mathfrak{a}},\pi}(L)$ to assess how well the player performs. Unless otherwise stated, we assume that any neural network structure is a multilayer perceptron, with two hidden layers of 20 nodes and ReLU activation functions. The output layer consists of $|\mathcal{A}|$ nodes with an identity activation function. Additional details on the implementation and hyperparameter initialization are given in Appendix A.

5.1 Deepsea

The first set of experiments is performed on the Deepsea task as described by Osband et al. [26]. The player starts each episode in the top left corner of a $T \times T$ grid, and on each round the player must move down one row and choose to either move one column to the left or to the right. Every episode terminates after T rounds, when the player has reached the last row. The difficulty is that while the action space $\mathcal{A} = \{0, 1\}$ is known, the meaning of these actions is encoded; that is, from one state the action "0" may mean left while from another state the action "0" may mean right. The player incurs a small cost $r_t = -0.01/T$ for moving right, $r_t = 0$ for moving left, and is rewarded with an additional $r_T = 1$ if they reach the bottom right corner. Clearly, the optimal reward is 0.99, which is achieved when the player moves right at each round.

Fig. 2 shows the cumulative regret of all procedures when playing the deep sea game for $L = 7,000$ episodes with $T = 20$. We observe that the only algorithms finding an optimal player strategy use a BootDQN+prior approach. Both NoisyNets and GaussianNets appear to perform poorly on complex large-scale tasks such as this sparse reward problem. However, we have observed that these novel methods are able to find an optimal player strategy on smaller grids (e.g. $T \leq 10$) where a naive deep Q-network will still fail, which means that our algorithms still achieve some form of deep exploration.

Empirically, we observe that including knowledge from the previous episodes to aggregate over the ensemble achieves lower regret than simply choosing uniformly. Our two proposed types of aggregation are better than the BoootDQN+prior approach with uniform sampling, and the Exp3 technique outperforms the PG technique in expectation (although both have high variance), as illustrated in Fig. 6. The evolution of the weights using the policy gradient and multi-armed bandit approaches with the original BootDQN+prior are given respectively in Figs. 4 and 5, and we notice that the emphasis is put towards only a subset of models that led to better results in earlier episodes.

5.2 Cartpole

The second set of experiments is performed on the well-known RL problem Cartpole [5]. In this setting, a player can move a cart to the left or the right, and its goal is to keep the pole attached to the cart upright. Each episode ℓ ends when either the pole is not upright anymore[2], or the number of rounds T_ℓ reaches 200. For every round t that the pole remains upright, the player is rewarded with a constant $r_t = 1$, which implies that the optimal player strategy consists of keeping it upright for 200 rounds.

In the RL literature, the Cartpole problem is not considered as an overly complicated learning task [15], so we expect all algorithms to have similar performance. Indeed, all approaches behave similarly on this problem when measuring the regret, as shown in Fig. 3, although the ensemble methods perform marginally better than the single network methods.

[2]The pole needs to stay balanced (i.e. the pole is less than 15 degrees from vertical) and centered (i.e. the cart deviates less than 2.4 units from the center).

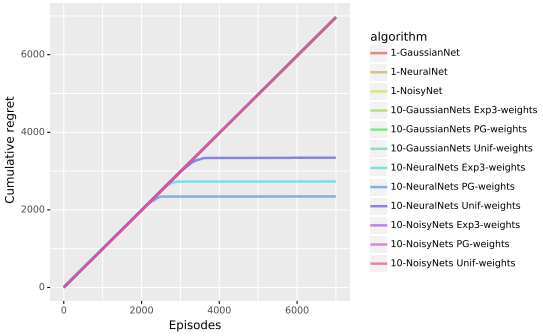

Figure 2: Cumulative regret for Deepsea.

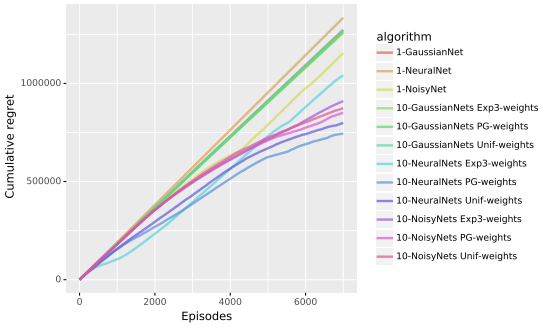

Figure 3: Cumulative regret for Cartpole.

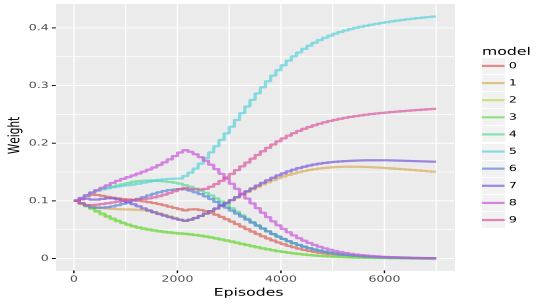

Figure 4: PG ensemble weights for Deepsea.

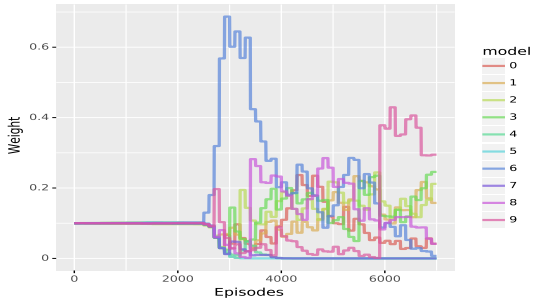

Figure 5: Exp3 ensemble weights for Deepsea.

5.3 Non-Stationary Data-Generating Mechanisms

Testing algorithms in non-stationary settings illustrates their robustness and performance on more complex real-life applications where a Markov assumption may be unrealistic. A non-stationary data-generating mechanism [28] allows both the reward and state transition distributions to change over time. We focus on our two previously described toy environments, Deepsea and Cartpole, and show how one can modify them in order to obtain simple non-stationary benchmarks. We introduce non-stationarity in the Deepsea environment by allowing the mapping between the action space and left / right transitions to change over time. At each episode, we incorporate a small probability of randomly selecting a column of the grid and reversing its action mapping. For Cartpole, we add non-stationarity by modifying the physics parameters. Instead of having the parameters fixed during the full training phase, we allow them to slowly evolve over each episode, where slowly may be characterized by random processes such as an autoregressive model.

Unfortunately, all tested approaches perform poorly according to the regret in non-stationary settings. This suggests the the regret is not an appropriate way to evaluate the performance of learning algorithms in non-stationary data-generating mechanisms, since it assumes that a player can find the optimal player strategy at each episode even with environment perturbations. Instead, the pseudo-regret from Eq. (1) may be a more reasonable measure of performance. Regardless, it remains open to modify the BootDQN+prior and ensemble aggregation framework to perform well in the face of such non-stationarity.

6 Related Work

Osband et al. [25] are not the only authors to explicitly inject noise in order to better model uncertainty and increase exploration in RL. Plappert et al. [27] also use noisy perturbations of the data to force exploration by applying diagonal Gaussian noise directly to the parameters of the deep Q-network, although they demonstrate empirically that this only improves performance in certain settings. Burda et al. [7] demonstrate that downweighting the observed rewards during training in favour of a realization from a "prior" (in the same sense as [25]) function is beneficial in some benchmark settings.

Modelling uncertainty using noisy posterior samples is also common in the *maximum a posteriori* (MAP) literature, of which our regularized MLE is an example. In this setting, an approach is to use *perturbation models*, which maximize over a random distribution to induce higher-order dependency structures for modelling purposes [11]. Some examples of these techniques are A* sampling [17] and GumbelMIP [14], which exploit

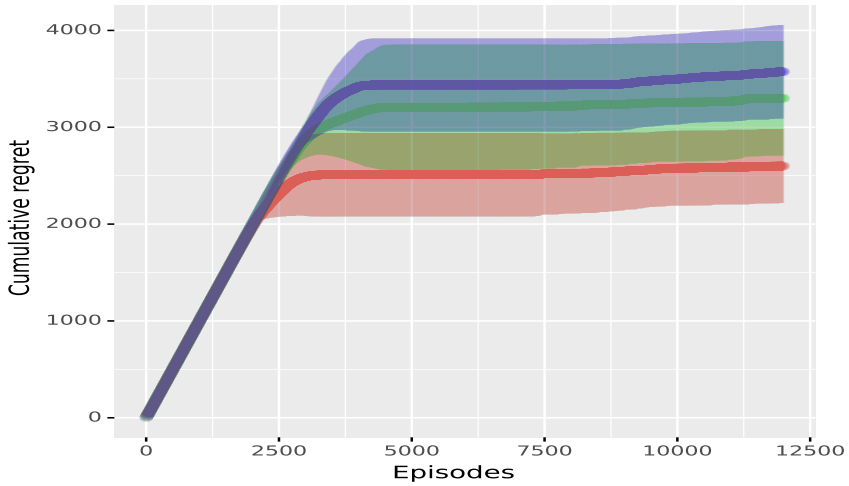

Figure 6: Confidence bands for cumulative regret on Deepsea using the BootDQN+prior algorithm. The three types of aggregation used are uniform sampling (blue), policy gradient (green) and Exp3 (red).

properties of the Gumbel distribution to obtain useful random maximizations. Implementing these algorithms in our regularized MLE framework for sampling Q-values may lead to more useful dependency structures than the dense Gaussian we consider, while still preserving the "structured noise" intuition.

Prioritizing an element from an ensemble of choices is an instance of sequential decision making with limited feedback, on which the literature is vast [for an overview, see 16]. The difference from the original RL problem we are solving is that there is no reason to assume the models in the ensemble have performance governed by an MDP like the rewards, since they are heavily influenced by the training process. We implement Exp3, the canonical example of the entropy-maximizing algorithms that have seen great success for RL exploration [13]. In particular, the combination of explicitly Bayesian methods with entropy-maximization has been used to obtain improved performance in RL via *K-learning* [22, 23]. Further, a wide range of algorithms can be implemented by replacing entropy with another regularizer in the generic follow-the-regularized-leader (FTRL) framework [24, 2]. This insight combined with existing modifications of FTRL better suited for use with deep networks, such as *follow-the-moving-leader* [33], may lead to more sophisticated ensemble aggregation approaches that perform better in complex environments. For example, improved robustness to environment dynamics has been demonstrated via approximations of posterior transition probabilities in an algorithm similar to the upper confidence bound algorithms [9].

7 Discussion

In this work, we provided novel extensions to existing methods for adding explicit uncertainty in deep Q-learning for RL. Our extensions use both Bayesian neural network and classical sequential decision making techniques to obtain empirical improvements in benchmark settings. While our proposed types of uncertainty achieve worse results on large-scale sparse reward tasks when actions are very unlikely to realize a reward, our novel approaches to aggregate ensembles using observed data outperform a naive uniform sampling.

Since the main goal of the BootDQN+prior algorithm is to enhance exploration, our methods should preserve this characteristic. One limitation of our aggregation techniques is that they may converge too quickly, and consequently ignore portions of the ensemble. We implement the naive solution of an ε-greedy approach that uniformly selects an model from the ensemble with probability ε and otherwise exploits the weight distribution. It remains open to examine more closely the performance of these algorithms in a non-stationary setting when adjusting this parameter; we propose that it should be data-dependent, in order to adapt to the situation where a good model gets worse as the environment is evolving.

In both the original and modified algorithms, the player focuses on multiple models of the ensemble, whether the whole ensemble or a smaller subset. Applying distillation [8] on the trained player would obviously provide model compression, as we transfer knowledge from the entire ensemble to a single smaller model, but it would also be interesting to investigate if distillation could achieve stronger performance on the long run.

It is still unclear what kind of prior mechanism is the most appropriate for reinforcement learning games. Although our proposed approaches give more flexibility to the induced prior mechanism, they seem to perform poorly on tasks requiring a thorough exploration. One direction pointed out by the authors would be to meta-learn the prior distributions.

Author Contributions

In this project, Blair implemented the Exp3 method, implemented the GaussianNet objects, created a wrapper function to easily use any algorithm, wrote Sections 2 and 3, and assisted with writing the remaining sections. Also, Anthony implemented the policy gradient method, implemented the NoisyNet objects, cleaned the code in Python files, ran the final experiments and wrote Section 5, and assisted with writing the remaining sections.

Acknowledgments and Disclosure of Funding

BB thanks Jeffrey Negrea and Daniel Roy for many discussions about precise notation for online prediction problems. BB and AC thank Chris Maddison for providing useful references about exploration via random neural networks.

References

[1] P. Auer, N. Cesa-Bianchi, and R. E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

[2] C.-A. Badr-Eddine, P. Alquier, and M. E. Khan. Generalization bounds for variational inference. In *Proceedings of the 11th Asian Conference on Machine Learning*, 2019.

[3] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[4] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight uncertainty in neural networks. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulan, J. Tang, and W. Zaremba. OpenAI Gym, 2016. arXiv:1606.01540.

[6] S. Bubeck and N. Cesa-Bianchi. *Regret Analysis of Stochastic and Nonstochastic Multi-Armed Bandit Problems*. Now Foundations and Trends, 2012.

[7] Y. Burda, H. Edwards, A. Storkey, and O. Klimov. Exploration by random network distillation. In *Proceedings of the 7th International Conference on Learning Representations*, 2019.

[8] W. M. Czarnecki, R. Pascanu, S. Osindero, S. Jayakumar, G. Swirszcz, and M. Jaderberg. Distilling policy distillation. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, 2019.

[9] E. Derman, D. Mankowitz, T. Mann, and S. Mannor. A Bayesian approach to robust reinforcement learning. In *Proceedings of the 35th Uncertainty in Artificial Intelligence Conference*, 2020.

[10] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, et al. Noisy networks for exploration, 2017. arXiv:1706.10295.

[11] A. Gane, T. Hazan, and T. Jaakkola. Learning with maximum a-posteriori perturbation models. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*, 2014.

[12] H. Hasselt. Double Q-learning. In *Advances in Neural Information Processing Systems 23*, 2010.

[13] E. Hazan, S. M. Kakade, K. Singh, and A. Van Soest. Provably efficient maximum entropy exploration. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.

[14] C. Kim, A. Sabharwal, and S. Ermon. Exact sampling with integer linear programs and random perturbations. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, 2016.

[15] S. Kumar. Balancing a CartPole system with reinforcement learning – a tutorial, 2020. arXiv:2006.04938.

[16] T. Lattimore and C. Szepesvári. *Bandit Algorithms*. Cambridge University Press, 2020.

[17] C. J. Maddison, D. Tarlow, and T. Minka. A* sampling. In *Advances in Neural Information Processing Systems 27*, 2014.

[18] H. Mania, A. Guy, and B. Recht. Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems 31*, 2018.

[19] F. S. Melo. Convergence of Q-learning: A simple proof. *Institute Of Systems and Robotics, Technical Report*, pages 1–4, 2001.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with deep reinforcement learning. *arXiv:1312.5602*, 2013.

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

[22] B. O'Donoghue. Variational bayesian reinforcement learning with regret bounds, 2018. arXiv:1807.09647.

[23] B. O'Donoghue, I. Osband, and C. Ionescu. Making sense of reinforcement learning and probabilistic inference. In *Proceedings of the 8th International Conference on Learning Representations*, 2020.

[24] F. Orabona. A modern introduction to online learning, 2019. arXiv:1912.13213.

[25] I. Osband, J. Aslanides, and A. Cassirer. Randomized prior functions for deep reinforcement learning. In *Advances in Neural Information Processing Systems 32*, 2018.

[26] I. Osband, Y. Doron, M. Hessel, J. Aslanides, E. Sezener, A. Saraiva, K. McKinney, T. Lattimore, C. Szepesvari, S. Singh, et al. Behaviour suite for reinforcement learning. *arXiv:1908.03568*, 2019.

[27] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz. Parameter space noise for exploration. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

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

[29] R. S. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[30] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, 1999.

[31] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double Q-learning. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, 2016.

[32] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[33] S. Zheng and J. T. Kwok. Follow the moving leader in deep learning. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[34] J. Zimmert and Y. Seldin. An optimal algorithm for stochastic and adversarial bandits. In *Proceedings of the 22nd International Conference on Artifical Intelligence and Statistics*, 2019.

A Implementation

In this section, we expand on the experimental setup in Section 5 with additional details on the implementation and hyperparameters used for the empirical study.

All code is written in Python and available at `https://github.com/acoache/sta4273-final-project`. The `envs.py` file contains environment classes for the Deepsea and Cartpole problems, as well as functions to make them non-stationary. Models are regrouped under the `models.py` file with classes to build ensembles of neural networks, NoisyNets and GaussianNets among others. The `losses.py` file has both the loss functions to train neural network parameters and the update rule for the weights of the ensemble. There is also a `utils.py` file which contains a replay buffer completely copied from the Osband et al. [25] colab notebook.

The whole training process is wrapped into a single function named `PlayRL`, where input arguments specify which algorithm it should use. The skeleton of the function is given in Algorithm 1. The user needs to give an environment class and a corresponding function that is called at every episode and adds non-stationarity in the data-generating mechanism. The function also requires an update rule for the weights of the ensemble, even though the ensemble could be a single model. Finally, specifications of the ensemble of models are given as inputs with three different arguments; a Python class to build the neural network structure, a method for sampling given the parametrization of Q-values, and a loss function to update the ensemble parameters. This function returns tables with performance results and the evolution of ensemble weights throughout the training phase, along with the trained agent.

We use the TensorFlow 2.0 library to build any neural network structure, whether an ordinary neural network, NoisyNet or GaussianNet. The algorithm updates parameters from the target network $\tilde{\theta}$ and performs a gradient step using the Adam optimizer (with learning rate 0.001 and batch size 64) for θ every episode. The weights of the ensemble w are modified only every 100 episodes, but this parameter can be tuned to use knowledge more frequently as it is made available. The discount factor is $\gamma = 0.99$ (as used by Osband et al. [25]) and we keep in memory the last $100,000$ rounds in an experience replay (which is all of them in our experiments). Some other implemented features include exploration rates of actions for both the policy and the neural network from the ensemble, although we set these to zero in our experiments for simplicity.

Algorithm 1: PlayRL function

Input: Environment, aggregation method, ensemble classes, other hyperparameters ;

Initialize optimizers, instances of ensembles and result dataframes ;
for *episode* **from** 1 **to** L **do**
    Sample from the ensemble one of the $\mathcal{J}$ models ;
    Update the target network ;
    while *game is not terminated* **do**
        Follow the greedy policy induced by the selected model ;
        Keep track of round experiences in a replay buffer ;
    Train the main network ;
    Update the ensemble weights according to the aggregation technique ;
    Perturb the environment ;
    Store results ;