PARAMETER-BASED VALUE FUNCTIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Traditional off-policy actor-critic Reinforcement Learning (RL) algorithms learn value functions of a single target policy. However, when value functions are updated to track the learned policy, they forget potentially useful information about old policies. We introduce a class of value functions called Parameter-based Value Functions (PVFs) whose inputs include the policy parameters. They can generalize across different policies. PVFs can evaluate the performance of any policy given a state, a state-action pair, or a distribution over the RL agent's initial states. First we show how PVFs yield novel off-policy policy gradient theorems. Then we derive off-policy actor-critic algorithms based on PVFs trained by Monte Carlo or Temporal Difference methods. We show how learned PVFs can zero-shot learn new policies that outperform any policy seen during training. Finally our algorithms are evaluated on a selection of discrete and continuous control tasks using shallow policies and deep neural networks. Their performance is comparable to the one of state-of-the-art methods.

1 INTRODUCTION

Value functions are central to Reinforcement Learning (RL). For a given policy, they estimate the value of being in a specific state (or of choosing a particular action in a given state). Many RL breakthroughs were achieved through improved estimates of such values, which can be used to find optimal policies (Tesauro, 1995; Mnih et al., 2015). However, learning value functions of arbitrary policies without observing their behavior in the environment is not trivial. Such off-policy learning requires to correct the mismatch between the distribution of updates induced by the behavioral policy and the one we want to learn. Common techniques include Importance Sampling (IS) (Hesterberg, 1988) and deterministic policy gradient methods (DPG) (Silver et al., 2014), which adopt the actor-critic architecture (Sutton, 1984; Konda & Tsitsiklis, 2001; Peters & Schaal, 2008a).

Unfortunately, these approaches have limitations. IS suffers from large variance (Cortes et al., 2010; Metelli et al., 2018) while DPG actor-critic methods introduce off-policy policy gradients that are only biased approximations to the true gradient since they do not estimate or compute the gradient of the action-value function with respect to the policy parameters $\nabla_\theta Q^{\pi_\theta}(s, a)$ (Degris et al., 2012; Silver et al., 2014). Furthermore, off-policy actor-critic algorithms learn value functions of a single target policy. When value functions are updated to track the learned policy, the information about old policies is lost.

We address the problem of generalization across many value functions in the off-policy setting by introducing a class of *parameter-based value functions* (PVFs) defined for any policy. PVFs are value functions whose inputs include the policy parameters, the PSSVF $V(\theta)$, PSVF $V(s, \theta)$, and PAVF $Q(s, a, \theta)$. PVFs can be learned using Monte Carlo (MC) (Metropolis & Ulam, 1949) or Temporal Difference (TD) (Sutton, 1988) methods. The PAVF $Q(s, a, \theta)$ leads to a novel stochastic and deterministic policy gradient theorem which are exact compared to previous approaches (Degris et al., 2012; Silver et al., 2014). Based on these results, we develop off-policy actor-critic methods and compare our algorithms to two strong baselines, ARS and DDPG (Mania et al., 2018; Lillicrap et al., 2015), outperforming them in some environments.

We make theoretical, algorithmic, and experimental contributions: Section 2 introduces the standard MDP setting; Section 3 formally presents PVFs and derive algorithms for $V(\theta)$, $V(s, \theta)$ and $Q(s, a, \theta)$; Section 4 describes the experimental evaluation using shallow and deep policies; Sections 5 and 6 discuss related and future work. Proofs and derivations can be found in Appendix A.2.

2 BACKGROUND

We consider a Markov Decision Process (MDP) (Stratonovich, 1960; Puterman, 2014) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma, \mu_0)$ where at each step an agent observes a state $s \in \mathcal{S}$, chooses action $a \in \mathcal{A}$, transitions into state s' with probability $P(s'|s,a)$ and receives a reward $R(s,a)$. The agent starts from an initial state, chosen with probability $\mu_0(s)$. It is represented by a parametrized stochastic policy $\pi_\theta : \mathcal{S} \rightarrow \Delta(\mathcal{A})$, which provides the probability of performing action a in state s. Θ is the space of policy parameters. The policy is deterministic if for each state s there exists an action a such that $\pi_\theta(a|s) = 1$. The return R_t is defined as the cumulative discounted reward from time step t: $R_t = \sum_{k=0}^{T-t-1} \gamma^k R(s_{t+k+1}, a_{t+k+1})$, where T denotes the time horizon and γ a real-valued discount factor. The performance of the agent is measured by the cumulative discounted expected reward (expected return), defined as $J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[R_0]$. Given a policy π_θ, the state-value function $V^{\pi_\theta}(s) = \mathbb{E}_{\pi_\theta}[R_t|s_t = s]$ is defined as the expected return for being in a state s and following policy π_θ. We define by $d^{\pi_\theta}(s')$ the discounted state distribution $d^{\pi_\theta}(s') = \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} \mu_0(s) P(s \rightarrow s', t, \pi_\theta) \, \mathrm{d}s$, where $P(s \rightarrow s', t, \pi_\theta)$ is the probability of transitioning to s' after t time steps, starting from s and following policy π_θ. By integrating over the state space $\mathcal{S}$, we can express the maximization of the expected cumulative reward in terms of the state-value function:

$$J(\pi_\theta) = \int_{\mathcal{S}} d^{\pi_\theta}(s) V^{\pi_\theta}(s) \, \mathrm{d}s. \tag{1}$$

The action-value function $Q^{\pi_\theta}(s,a)$, which is defined as the expected return for performing action a in state s, and following the policy π_θ, is $Q^{\pi_\theta}(s,a) = \mathbb{E}_{\pi_\theta}[R_t|s_t = s, a_t = a]$, and it is related to the state-value function by:

$$V^{\pi_\theta}(s) = \int_{\mathcal{A}} \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \, \mathrm{d}a. \tag{2}$$

In off-policy policy optimization, we seek to find the parameters of the policy maximizing the performance index $J(\pi_\theta)$ using data collected from a behavioral policy π_b. Formally, we want to find:

$$\theta^* = \arg\max_\theta \int_{\mathcal{S}} d^{\pi_b}(s) V^{\pi_\theta}(s) \, \mathrm{d}s = \arg\max_\theta \int_{\mathcal{S}} d^{\pi_b}(s) \int_{\mathcal{A}} \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \, \mathrm{d}a \, \mathrm{d}s, \tag{3}$$

where the value function is integrated over the distribution of the states under the behavioral policy.

3 PARAMETER-BASED VALUE FUNCTIONS

In this section, we introduce our parameter-based value functions, the PSSVF $V(\theta)$, PSVF $V(s,\theta)$, and PAVF $Q(s,a,\theta)$ and their corresponding learning algorithms. First, we augment the state and action-value functions, allowing them to receive as an input also the weights of a parametric policy. The parameter-based state-value function (PSVF) $V(s,\theta) = \mathbb{E}[R_t|s_t = s, \theta]$ is defined as the expected return for being in state s and following policy parameterized by θ. Similarly, the parameter-based action-value function (PAVF) $Q(s,a,\theta) = \mathbb{E}[R_t|s_t = s, a_t = a, \theta]$ is defined as the expected return for being in state s, taking action a and following policy parameterized by θ. In the off-policy setting, the objective to be maximized becomes:

$$J(\pi_{\theta^*}) = \max_\theta \int_{\mathcal{S}} d^{\pi_b}(s) V(s,\theta) \, \mathrm{d}s = \max_\theta \int_{\mathcal{S}} \int_{\mathcal{A}} d^{\pi_b}(s) \pi_\theta(a|s) Q(s,a,\theta) \, \mathrm{d}a \, \mathrm{d}s. \tag{4}$$

The performance index J can also be defined with respect to the start-value function (Peters & Schaal, 2008b):

$$J_{\mu_0}(\pi_\theta) = \int_{\mathcal{S}} \mu_0(s) V^{\pi}(s,\theta) \, \mathrm{d}s. \tag{5}$$

This notation allows us to omit the dependence on the state in the PSVF and learn a function $V(\theta)$, named parameter-based start-state-value function (PSSVF), which is an approximation of $J_{\mu_0}(\pi_\theta)$:

$$V(\theta) := \mathbb{E}_{s \sim \mu_0(s)}[V(s,\theta)] = \int_{\mathcal{S}} \mu_0(s) V(s,\theta) \, \mathrm{d}s = J_{\mu_0}(\pi_\theta). \tag{6}$$

By taking the gradient of the performance J with respect to the policy parameters θ we obtain novel policy gradient theorems. Since θ is continuous, we need to use function approximators $V_{\mathbf{w}}(\theta) \approx$

$V(\theta)$, $V_{\mathbf{w}}(s,\theta) \approx V(s,\theta)$ and $Q_{\mathbf{w}}(s,a,\theta) \approx Q(s,a,\theta)$. In all cases, the policy improvement step can be very expensive, due to the computation of the $\arg\max$ over a continuous space Θ. Actor-critic methods can be derived to solve this optimization problem, where the critic (PVFs) can be learned using TD or MC methods, while the actor is updated following the gradient with respect to the critic. Although our algorithms on PSSVF and PSVF can be used with stochastic policies, removing the stochasticity of the action-selection process can facilitate learning the value function. Therefore, we use deterministic policies. All our algorithms make use of a replay buffer.

3.1 Parameter-based Start-State-Value Function $V(\theta)$

We first derive the PSSVF $V(\theta)$. Given the performance index J_{μ_0}, and taking the gradient with respect to θ, we obtain:

$$\nabla_\theta J_{\mu_0}(\pi_\theta) = \int_{\mathcal{S}} \mu_0(s) \nabla_\theta V(s,\theta)\,\mathrm{d}s = \mathbb{E}_{s\sim\mu_0(s)}[\nabla_\theta V(s,\theta)] = \nabla_\theta V(\theta). \tag{7}$$

In Algorithm 1, the critic $V_{\mathbf{w}}(\theta)$ is learned using MC to estimate the value of any policy θ. The actor is then updated following the direction of improvement suggested by the critic. Since the PSSVF can be used only for episodic tasks, we optimize for the undiscounted objective.

Algorithm 1 Actor-critic with Monte Carlo prediction for $V(\theta)$

Input: Differentiable critic $V_{\mathbf{w}} : \Theta \to \mathcal{R}$ with parameters $\mathbf{w}$; deterministic actor π_θ with parameters θ; empty replay buffer D
Output : Learned $V_{\mathbf{w}} \approx V(\theta) \forall \theta$, learned $\pi_\theta \approx \pi_{\theta^*}$
Initialize critic and actor weights $\mathbf{w}, \theta$
repeat:
  Generate an episode $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$ with policy π_θ
  Compute return $r = \sum_{k=1}^{T} r_k$
  Store (θ, r) in the replay buffer D
  for many steps **do**:
    Sample a batch $B = \{(r,\theta)\}$ from D
    Update critic by stochastic gradient descent: $\nabla_{\mathbf{w}} \mathbb{E}_{(r,\theta)\in B}[r - V_{\mathbf{w}}(\theta)]^2$
  end for
  for many steps **do**:
    Update actor by gradient ascent: $\nabla_\theta V_{\mathbf{w}}(\theta)$
  end for
until convergence

3.2 Parameter-based State-Value Function $V(s,\theta)$

Learning the value function using MC approaches can be difficult due to the high variance of the estimate. Furthermore, episode-based algorithms like Algorithm 1 are unable to credit good actions in bad episodes. Gradient methods based on TD updates provide a biased estimate of $V(s,\theta)$ with much lower variance and can credit actions at each time step. Taking the gradient of $J(\theta)$ in the PSVF formulation[1], we obtain:

$$\nabla_\theta J(\pi_\theta) = \int_{\mathcal{S}} d^{\pi_b}(s) \nabla_\theta V(s,\theta)\,\mathrm{d}s = \mathbb{E}_{s\sim d^{\pi_b}(s)}[\nabla_\theta V(s,\theta)]. \tag{8}$$

Algorithm 2 (Appendix) uses the actor-critic architecture, where the critic is learned via TD[2].

3.3 Parameter-based Action-Value Function $Q(s,a,\theta)$

The introduction of the PAVF $Q(s,a,\theta)$ allows us to derive new policy gradients methods (Sutton et al., 1999) when using a stochastic or deterministic policy.

[1]Compared to standard methods based on the state-value function, we can directly optimize the policy following the performance gradient of the PSVF, obtaining a policy improvement step in a model-free way.

[2]Note that the differentiability of the policy π_θ is never required in PSSVF and PSVF.

Stochastic policy gradients We want to use data collected from some stochastic behavioral policy π_b in order to learn the action-value of a target policy π_θ. Traditional off-policy actor-critic algorithms only approximate the true policy gradient, since they do not estimate the gradient of the action-value function with respect to the policy parameters $\nabla_\theta Q^{\pi_\theta}(s,a)$ (Degris et al., 2012; Silver et al., 2014). With PVFs, we can directly compute this contribution to the gradient. This yields an exact policy gradient theorem:

Theorem 3.1. *For any Markov Decision Process, the following holds:*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim d^{\pi_b}(s), a\sim\pi_b(.|s)}\left[\frac{\pi_\theta(a|s)}{\pi_b(a|s)}\left(Q(s,a,\theta)\nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q(s,a,\theta)\right)\right]. \quad (9)$$

Let us compare this to the standard off-policy policy gradient (Degris et al., 2012):

$$\nabla_\theta J(\pi_\theta) \approx \mathbb{E}_{s\sim d^{\pi_b}(s), a\sim\pi_b(.|s)}\left[\frac{\pi_\theta(a|s)}{\pi_b(a|s)}\left(Q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\right)\right]. \quad (10)$$

We notice that our expectation contains the derivative of the value function with respect to the policy. Algorithm 3 (Appendix) uses an actor-critic architecture and can be seen as an extension of Off-PAC (Degris et al., 2012) to PAVF.

Deterministic policy gradients Estimating $Q(s,a,\theta)$ is in general a difficult problem due to the stochasticity of the policy. Deterministic policies of the form $\pi : \mathcal{S} \rightarrow \mathcal{A}$ can help improving the efficiency in learning value functions, since the expectation over the action space is no longer required. Using PVFs, we can write the performance of a policy π_θ as:

$$J(\pi_\theta) = \int_{\mathcal{S}} d^{\pi_b}(s) V(s,\theta)\,\mathrm{d}s = \int_{\mathcal{S}} d^{\pi_b}(s) Q(s,\pi_\theta(s),\theta)\,\mathrm{d}s. \quad (11)$$

Taking the gradient with respect to θ we obtain a deterministic policy gradient theorem:

Theorem 3.2. *Under standard regularity assumptions (Silver et al., 2014), for any Markov Decision Process, the following holds:*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim d^{\pi_b}(s)}\left[\nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)}\nabla_\theta \pi_\theta(s) + \nabla_\theta Q(s,a,\theta)|_{a=\pi_\theta(s)}\right]. \quad (12)$$

Algorithm 4 (Appendix) uses an actor-critic architecture and and can be seen as an extension of DPG (Silver et al., 2014) to PAVF. Despite the novel formulation of algorithm 3, we decided to avoid the stochasticity of the policy and to implement and analyze only the deterministic PAVF.

4 EXPERIMENTS

Applying algorithms 1, 2 and 4 directly can lead to convergence to local optima, due to the lack of exploration. In practice, like in standard actor-critic algorithms, we use a noisy version of the current learned policy in order to act in the environment and collect data to encourage exploration. More precisely, at each episode we use $\pi_{\tilde{\theta}}$ with $\tilde{\theta} = \theta + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma^2 I)$ instead of π_θ and then store $\tilde{\theta}$ in the replay buffer. In our experiments, we report both for our methods as well as the baselines the performance of the policy without noise. Source code will be made available with the publication.

4.1 VISUALIZING PVFS USING LQRS

We start with an illustrative example that allows us to visualize how PVFs are learning to estimate the expected return over the parameter space. For this purpose, we use an instance of the 1D Linear Quadratic Regulator (LQR) problem and a linear policy with bias. In figure 1, we plot $J(\theta)$, the cumulative return that an agent would obtain by acting in the environment using policy π_θ and the cumulative return predicted by the PSSVF $V(\theta)$ for two different times during learning. At the beginning of the learning process, the PSSVF is able to provide just a local estimation of the performance of the agent, since only few data have been observed. However, after 500 episodes, it is able to provide a more accurate global estimate over the parameter space. We provide details of the environment and the hyperparameters used in Appendix A.4.1.

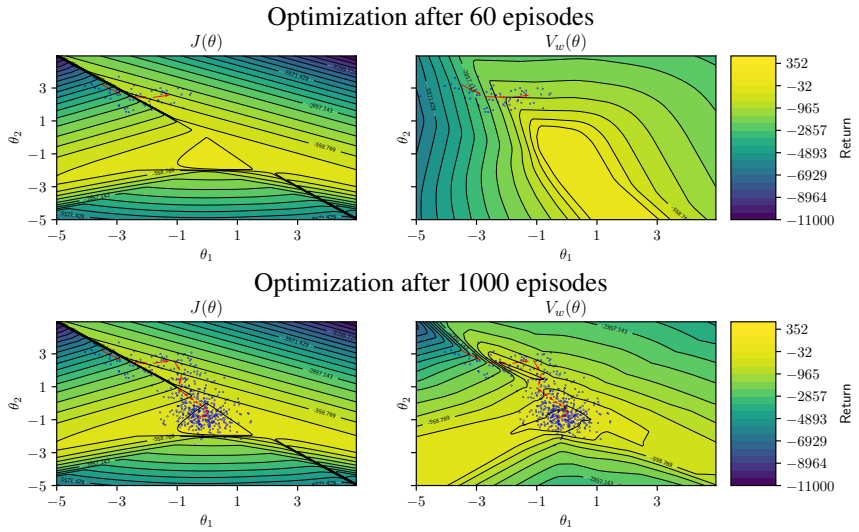

Figure 1: True cumulative return $J(\theta)$ and PSSVF estimation $V(\theta)$ as a function of the policy parameters at two different stages in training. The red arrows represent an optimization trajectory in parameter space. The blue dots represent the perturbed policy used to train $V(\theta)$.

4.2 Main results

Given the similarities between our PAVF and DPG, Deep Deterministic Policy Gradients (DDPG) is a natural choice for the baseline. Additionally, the PSSVF $V(\theta)$ resembles evolutionary methods as the critic can be interpreted as a global fitness function. Therefore, we decided to include in the comparison Augmented Random Search (ARS) which is known for its state-of-the-art performance using only linear policies in continuous control tasks. For the policy, we use a 2-layer MLP (64,64) with tanh activations and a linear policy followed by a tanh nonlinearity. Figure 2 shows results for both policy architectures. In all the tasks the PSSVF is able to achieve at least the same performance compared to ARS, often outperforming it. In the Inverted Pendulum environment, PSVF and PAVF with deep policy are very slow to converge, but they excel in the Swimmer task and MountainCarContinuous. In Reacher, all PVFs fail to learn the task, while DDPG converges quickly to the optimal policy. We conjecture that for this task it is difficult to perform a search in parameter space. On the other hand, in MountainCarContinuous, the reward is more sparse and DDPG only rarely observes positive reward when exploring in action space. In Appendix A.4 we include additional results with other policy sizes and hyperparameters. We analyze the sensitivity of the algorithms on the choice of hyperparameters in Appendix A.4.4.

4.3 Zero-shot learning

In order to test whether PVFs are generalizing across the policy space, we perform the following experiment with shallow policies: while learning using algorithm 1, we stop training and randomly initialize 5 policies. Then, without interacting with the environment, we train these policies offline, in a zero-shot manner, following only the direction of improvement suggested by $\nabla_\theta V_w(\theta)$, whose weights w remain frozen. We observe that shallow policies can be effectively trained from scratch. Results for PSSVFs in Swimmer-v3 are displayed in figure 3. In particular, we compare the performance of the policy learned, the best perturbed policy for exploration seen during training and five policies learned from scratch at three different stages in training. We note that after the PSSVF has been trained for 100,000 time steps interactions with the environment (first snapshot), these policies are already able to outperform both the current policy and any policy seen while training the PSSVF. They achieve an average return of 297, while the best observed return was 225. We observed similar performance for most of the environments and methods. When using deep policies, we obtained similar results only for the simplest environments. For this task, we used the same hyperparameters as in figure 2.

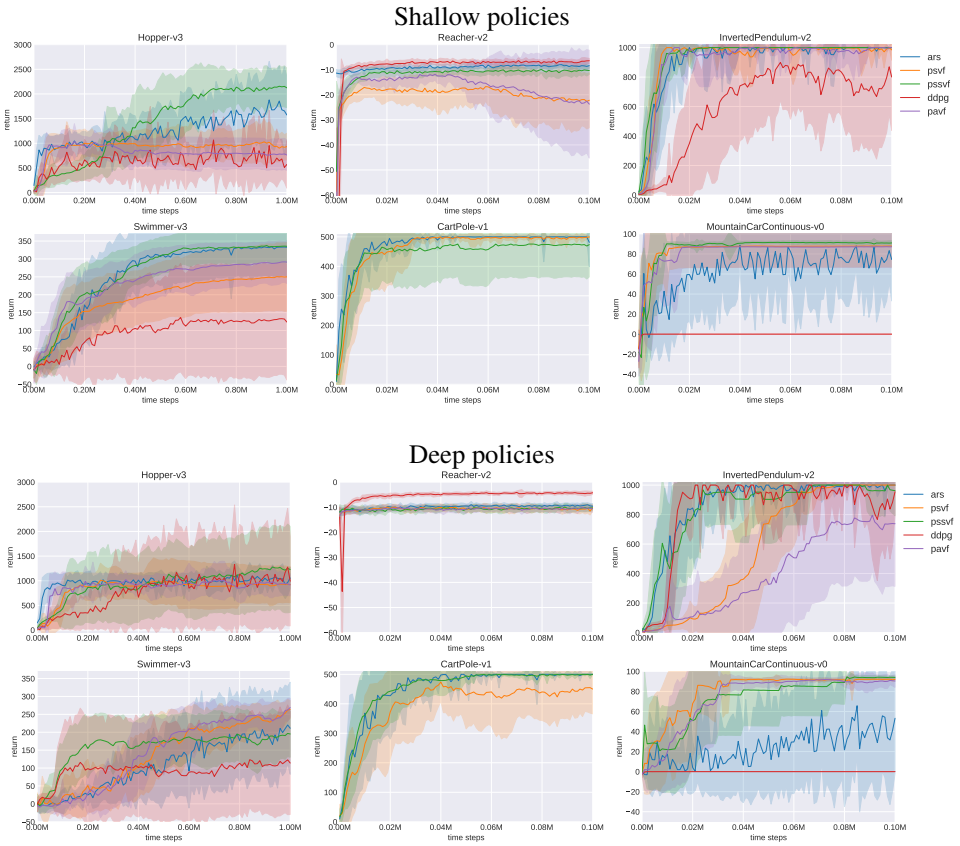

Figure 2: Average return of shallow and deep policies as a function of the number of time steps used for learning (across 20 runs, one standard deviation), for different environments and algorithms. We use the best hyperparameters found when maximizing the average return.

4.4 Offline learning with fragmented behaviors

In our last experiment we investigate how PSVFs are able to learn in a full offline setting. The goal is to learn a good policy in Swimmer-v3 given a fixed dataset containing 100,000 transitions, without additional environment interactions. Furthermore, the policy generating the data is perturbed every 200 time steps, for a total of 5 policies per episode. Observing only incomplete trajectories for each policy parameter makes TD bootstrapping harder: In order to learn, the PSVF needs to generalize across both the state and the parameter space. Given the fixed dataset, we first train the PSVF, minimizing the TD error. Then, at different stages during learning, we train 5 new shallow policies. Figure 4 describes this process. We note that at the beginning of training, when the PSVF $V(s, \theta)$ has a larger TD error, the policies learned from scratch have poor performance. However, after 7000 gradient updates, they are able to achieve a reward of 237, before eventually degrading to 167. They outperform the best policy in the dataset used to train the PSVF, whose return is only of 58.

5 Related work

There are two main classes of similar algorithms performing search in policy parameter space. Evolutionary algorithms (Wierstra et al., 2014; Salimans et al., 2017; Mania et al., 2018) iteratively estimate a fitness function evaluating the performance of a population of policies and then perform gradient ascent in parameter space, often estimating the gradient using finite difference approximation. By replacing the performance of a population through a likelihood estimation, evolutionary algorithms become a form of Parameter Exploring Policy Gradients (Sehnke et al., 2008; 2010). Our methods are similar to evolution since our value function can be seen as a fitness. Unlike evolution, however, our approach allows for obtaining the fitness gradient directly and is more

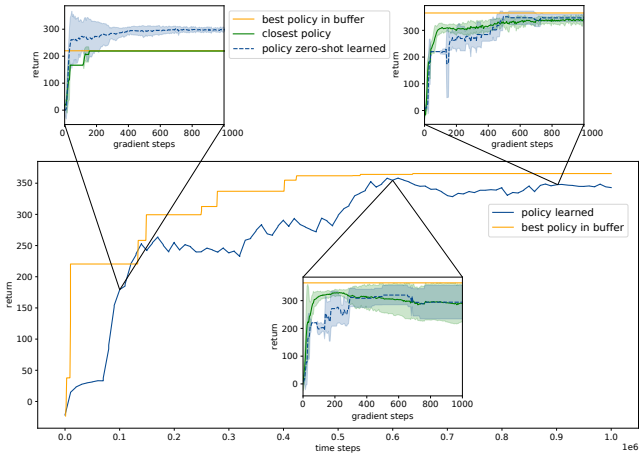

Figure 3: Policies learned from scratch during training. The plot in the center represents the return of the agent learning while interacting with the environment using Algorithm 1. We compare the best noisy policy $\pi_{\tilde{\theta}}$ used for exploration to the policy π_θ learned through the critic. The learning curves in the small plots represent the return obtained by policies trained from scratch following the fixed critic $V_{\mathbf{w}}(\theta)$ after different time steps of training. The return of the closest policy (L2 distance) in the replay buffer with respect to the policy learned from scratch is depicted in green.

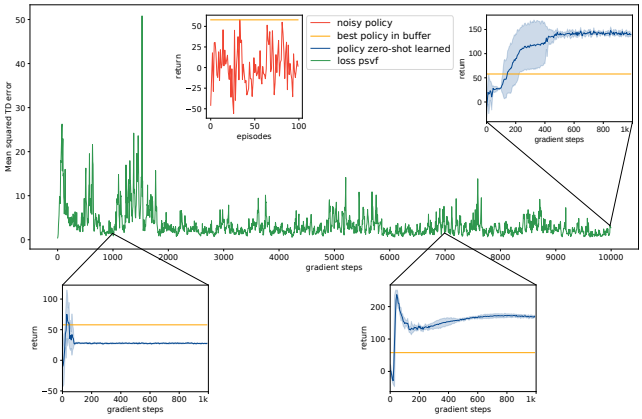

Figure 4: Offline learning of PSVF. We plot the mean squared TD error of a PSVF trained using data coming from a set of noisy policies. In the small plots, we compare the return obtained by policies trained from scratch following the fixed critic $V_{\mathbf{w}}(s, \theta)$ after different time steps of value function training and the return of the best noisy policy used to train V.

suitable for reusing past data. While direct $V(\theta)$ optimization is strongly related to evolution, our more informed algorithms optimize $V(s, \theta)$ and $Q(s, a, \theta)$. That is, ours both perform a search in policy parameter space AND train the value function and the policy online, without having to wait for the ends of trials or episodes.

The second related class of methods involves surrogate functions (Box & Wilson, 1951; Booker et al., 1998; Moore & Schneider, 1996). They often use local optimizers for generalizing across fitness functions. In particular, Bayesian Optimization (BO) (Snoek et al., 2012; 2015) uses a surrogate function to evaluate the performance of a model over a set of hyperparameters and follows the uncertainty on the surrogate to query the new data to sample. Unlike BO, we do not build a probabilistic model and we use the gradient of the value function instead of a sample from the posterior to decide which policy parameters to use next in the policy improvement step.

In 1990, adaptive critics trained by TD were used to predict the gradients of an RNN from

its activations (Schmidhuber, 1990b), avoiding backpropagation through time (BPTT) (Werbos, 1990). This idea was later used to update the weights of a neural network asynchronously (Jaderberg et al., 2017). In our work, the critic is predicting errors instead of gradients. If applied to POMDPs, or supervised learning tasks involving long time lags between relevant events, the PSSVF could avoid BPTT by viewing the parameters of an RNN as a static object and mapping them to their loss (negative reward). The possibility of augmenting the value functions with auxiliary parameters was already considered in work on General Value Functions (Sutton et al., 2011), where the return is defined with respect to an arbitrary reward function. Universal Value Function Approximators (Schaul et al., 2015) used the same approach to learn a single value function representing the value, given possible agent goals. However, General and Universal Value Functions have not been applied to learn a single value function for every possible policy.

Recent work (Unterthiner et al., 2020) shows how to map the weights of a trained Convolutional Neural Network to its accuracy. Experiments show how these predictions allow for performance rankings of neural networks on new unseen tasks. These maps are either learned by taking the flattened weights as input or using simple statistics. However, these predictions do not guide the training process of CNNs. Policy Evaluation Networks (PENs) (Harb et al., 2020) are closely related to our work and share the same motivation. PENs focus on the simplest PSSVF $V(\theta)$ trained without an actor-critic architecture. Like in some of our experiments, the authors show how following the direction of improvement suggested by $V(\theta)$ leads to an increase in policy performance. They also suggest to explore in future work a more complex setting where a PSVF $V(s, \theta)$ is learned using an actor-critic architecture. Our work directly introduces the PSVF $V(s, \theta)$ and PAVF $Q(s, a, \theta)$ and presents novel policy gradient theorems for PAVFs when stochastic or deterministic policies are used. We include a more detailed comparison in Appendix A.1.

Gradient Temporal Difference (Sutton et al., 2009a;b; Maei et al., 2009; 2010; Maei, 2011) and Emphatic Temporal Difference methods (Sutton et al., 2016) were developed to address convergence under on-policy and off-policy (Precup et al., 2001) learning with function approximation. The first attempt to obtain a stable off-policy actor-critic algorithm under linear function approximation was called Off-PAC (Degris et al., 2012), where the critic is updated using GTD(λ) (Maei, 2011) to estimate the state-value function. This algorithm has promising theoretical guarantees in the tabular case. However, in general, the actor does not follow the true gradient direction. A paper on DPG (Silver et al., 2014) extended the Off-PAC policy gradient theorem (Degris et al., 2012) to deterministic policies. This was coupled with a deep neural network to solve continuous control tasks through Deep Deterministic Policy Gradients (Lillicrap et al., 2015). The idea of using actor-critic methods with deterministic policies was proposed even earlier (Schmidhuber, 1990a). Here the critic was trained by TD to predict possibly multi-dimensional cumulative rewards, although no policy gradient theorem was formally derived.

All our algorithms are based on the off-policy actor-critic architecture. The two algorithms based on $Q(s, a, \theta)$ can be viewed as analogous to Off-PAC and DPG where the critic is defined for all policies and the actor is updated following the true gradient with respect to the critic.

6 Limitations and future work

We introduced PVFs, a novel class of value functions which receive as input the parameters of a policy and can be used for off-policy learning. We have shown that PVFs are competitive to ARS and DDPG (Mania et al., 2018; Lillicrap et al., 2015) while generalizing across policies and allowing for zero-shot training in an offline setting. Despite their positive results on shallow and deep policies, PVFs suffer the curse of dimensionality when the number of policy parameters is high. Embeddings similar to those used in PENs (Harb et al., 2020) may be useful not only for saving memory and computational time, but also for facilitating search in parameter space. We intend to evaluate the benefits of such embeddings and other dimensionality reduction techniques. We derived off-policy policy gradient theorems, showing how PVFs are optimizing for the correct unbiased off-policy objective. With these results, we plan to analyze the convergence of our algorithms using stochastic approximation techniques (Borkar, 2009) and test them on environments where traditional methods are known to diverge (Baird, 1995). Finally, we want to investigate how PVFs applied to supervised learning tasks or POMDPs, can avoid BPTT by mapping the weights of an RNN to its loss.

REFERENCES

Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning Proceedings 1995*, pp. 30–37. Elsevier, 1995.

Andrew J. Booker, J. E. Dennis, Paul D. Frank, David B. Serafini, and Virginia Torczon. *Optimization Using Surrogate Objectives on a Helicopter Test Example*, pp. 49–58. Birkhäuser Boston, Boston, MA, 1998. ISBN 978-1-4612-1780-0. doi: 10.1007/978-1-4612-1780-0_3. URL https://doi.org/10.1007/978-1-4612-1780-0_3.

Vivek S Borkar. *Stochastic approximation: a dynamical systems viewpoint*, volume 48. Springer, 2009.

G. E. P. Box and K. B. Wilson. On the experimental attainment of optimum conditions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 13(1):1–45, 1951. ISSN 00359246. URL http://www.jstor.org/stable/2983966.

Corinna Cortes, Yishay Mansour, and Mehryar Mohri. Learning bounds for importance weighting. In *Advances in neural information processing systems*, pp. 442–450, 2010.

Thomas Degris, Martha White, and Richard S. Sutton. Off-policy actor-critic. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML'12, pp. 179–186, USA, 2012. Omnipress. ISBN 978-1-4503-1285-1. URL http://dl.acm.org/citation.cfm?id=3042573.3042600.

Jean Harb, Tom Schaul, Doina Precup, and Pierre-Luc Bacon. Policy evaluation networks. *arXiv preprint arXiv:2002.11833*, 2020.

Timothy Classen Hesterberg. *Advances in importance sampling*. PhD thesis, Stanford University, 1988.

Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1627–1635. JMLR. org, 2017.

Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Society for Industrial and Applied Mathematics*, 42, 04 2001.

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. *arXiv preprint arXiv:1509.02971*, 2015.

Hamid R. Maei, Csaba Szepesvári, Shalabh Bhatnagar, Doina Precup, David Silver, and Richard S. Sutton. Convergent temporal-difference learning with arbitrary smooth function approximation. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, NIPS'09, pp. 1204–1212, Red Hook, NY, USA, 2009. Curran Associates Inc. ISBN 9781615679119.

Hamid Reza Maei. *Gradient temporal-difference learning algorithms*. PhD thesis, University of Alberta, 2011.

Hamid Reza Maei, Csaba Szepesvári, Shalabh Bhatnagar, and Richard S. Sutton. Toward off-policy learning control with function approximation. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pp. 719–726, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 1800–1809, 2018.

Alberto Maria Metelli, Matteo Papini, Francesco Faccio, and Marcello Restelli. Policy optimization via importance sampling. In *Advances in Neural Information Processing Systems*, pp. 5442–5454, 2018.

N. Metropolis and S. Ulam. The monte carlo method. *J. Am. Stat. Assoc.*, 44:335, 1949.

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.

Andrew W. Moore and Jeff G. Schneider. Memory-based stochastic optimization. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo (eds.), *Advances in Neural Information Processing Systems 8*, pp. 1066–1072. MIT Press, 1996. URL http://papers.nips.cc/paper/1124-memory-based-stochastic-optimization.pdf.

Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomput.*, 71(7-9):1180–1190, March 2008a. ISSN 0925-2312. doi: 10.1016/j.neucom.2007.11.026. URL http://dx.doi.org/10.1016/j.neucom.2007.11.026.

Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008b.

Doina Precup, Richard S. Sutton, and Sanjoy Dasgupta. Off-policy temporal difference learning with function approximation. In *ICML*, 2001.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Tom Schaul, Dan Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 1312–1320. JMLR.org, 2015. URL http://dl.acm.org/citation.cfm?id=3045118.3045258.

Jürgen Schmidhuber. Making the world differentiable: On using self-supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments. 1990a.

Jürgen Schmidhuber. Networks adjusting networks. In *Proceedings of" Distributed Adaptive Neural Information Processing"*, pp. 197–208, 1990b.

Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Policy gradients with parameter-based exploration for control. In Véra Kůrková, Roman Neruda, and Jan Koutník (eds.), *Artificial Neural Networks - ICANN 2008*, pp. 387–396, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87536-9.

Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, May 2010. ISSN 08936080. doi: 10.1016/j.neunet.2009.12.004. URL https://linkinghub.elsevier.com/retrieve/pii/S0893608009003220.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pp. I–387–I–395. JMLR.org, 2014. URL http://dl.acm.org/citation.cfm?id=3044805.3044850.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pp. 2951–2959, 2012.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat Prabhat, and Ryan P. Adams. Scalable bayesian optimization using deep neural networks. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 2171–2180. JMLR.org, 2015.

RL Stratonovich. Conditional Markov processes. *Theory of Probability And Its Applications*, 5(2): 156–178, 1960.

Richard S Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst, 1984.

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

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pp. 1057–1063, Cambridge, MA, USA, 1999. MIT Press. URL http://dl.acm.org/citation.cfm?id=3009657.3009806.

Richard S Sutton, Hamid R Maei, and Csaba Szepesvári. A convergent $o(n)$ temporal-difference algorithm for off-policy learning with linear function approximation. In *Advances in neural information processing systems*, pp. 1609–1616, 2009a.

Richard S. Sutton, Hamid Reza Maei, Doina Precup, Shalabh Bhatnagar, David Silver, Csaba Szepesvári, and Eric Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pp. 993–1000, New York, NY, USA, 2009b. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553501. URL https://doi.org/10.1145/1553374.1553501.

Richard S. Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M. Pilarski, Adam White, and Doina Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '11, pp. 761–768, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 0-9826571-6-1, 978-0-9826571-6-4. URL http://dl.acm.org/citation.cfm?id=2031678.2031726.

Richard S Sutton, A Rupam Mahmood, and Martha White. An emphatic approach to the problem of off-policy temporal-difference learning. *The Journal of Machine Learning Research*, 17(1): 2603–2631, 2016.

Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995. ISSN 0001-0782. doi: 10.1145/203330.203343. URL https://doi.org/10.1145/203330.203343.

Thomas Unterthiner, Daniel Keysers, Sylvain Gelly, Olivier Bousquet, and Ilya Tolstikhin. Predicting neural network accuracy from weights. *arXiv preprint arXiv:2002.11448*, 2020.

Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *The Journal of Machine Learning Research*, 15(1):949–980, 2014.

A APPENDIX

INDEX OF THE APPENDIX

In the following, we briefly recap the contents of the appendix.

- Appendix A.1 contains additional related works
- Appendix A.2 reports all proofs and derivations.
- Appendix A.3 illustrates implementation details and pseudocode.
- Appendix A.4 provides the hyperparameters used in the experiments and further results.

A.1 ADDITIONAL RELATED WORKS

Recently we learned about Policy Evaluation Networks (PENs) (Harb et al., 2020) which are closely related to our work and share the same motivation. There are many differences between our approach to learning $V(\theta)$ and theirs (Harb et al., 2020). For example, we do not use a fingerprint mechanism (Harb et al., 2020) for embedding the weights of complex policies. Instead, we simply parse all the policy weights as inputs to the value function, even in the nonlinear case. However, fingerprinting (Harb et al., 2020) may be important for representing nonlinear policies without losing information about their structure and for saving memory required to store the weights.

Other differences concern the optimization problem: we do not predict a bucket index for discretized reward, but perform a regression task. Therefore our loss is simply the mean squared error between the prediction of $V(\theta)$ and the reward obtained by π_θ, while their loss (Harb et al., 2020) is the KL divergence between the predicted and target distributions. Both approaches optimize the undiscounted objective when learning $V(\theta)$.

Harb et al. (2020) focus on the offline setting. They first collect a batch of randomly initialized policies and perform rollouts to collect reward from the environment. The PSSVF $V(\theta)$ is then trained using the data collected. Once V is trained, many gradient ascent steps through V yield new, unseen, randomly initialized policies in a zero-shot manner, exhibiting improved performance. In our offline experiment in section 4.4, we used a more difficult setting where each episode in the environment is composed by sub-trajectories generated using different policies. Episode-based approaches like PSSVF are not suitable for this task, while our PSVF can effectively learn how to improve the policy without waiting for the end of an episode.

Harb et al. (2020) train their value function using small nonlinear policies of one hidden layer and 30 neurons on Swimmer-v3. They evaluate 2000 deterministic policies on 500 episodes each (1 million policy evaluations), achieving a final expected return of ≈ 180 on new policies trained from scratch through V and a maximum observed return of 250. On the other hand, in our offline experiment using a linear PSSVF, after only 100 policy evaluations, we obtain a return of 297.

In our main experiments, we showed that a fingerprint mechanism is not necessary for the tasks we analyzed: even when using a much bigger 2-layers MLP policy, we are able to outperform the results in PEN. Although Harb et al. (2020) use Swimmer-v3 "to scale up their experiments", our results suggest that Swimmer-v3 does not conclusively demonstrate possible benefits of their policy embedding.

A.2 PROOFS AND DERIVATIONS

Theorem 3.1. *For any Markov Decision Process, the following holds:*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim d^{\pi_b}(s), a\sim\pi_b(.|s)} \left[\frac{\pi_\theta(a|s)}{\pi_b(a|s)} \left(Q(s,a,\theta)\nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q(s,a,\theta) \right) \right]. \quad (9)$$

Proof.

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \int_{\mathcal{S}} d^{\pi_b}(s) V(s,\theta)\,\mathrm{d}s \tag{13}$$

$$= \nabla_\theta \int_{\mathcal{S}} d^{\pi_b}(s) \int_{\mathcal{A}} \pi_\theta(a|s) Q(s,a,\theta)\,\mathrm{d}a\,\mathrm{d}s \tag{14}$$

$$= \int_{\mathcal{S}} d^{\pi_b}(s) \int_{\mathcal{A}} [Q(s,a,\theta)\nabla_\theta \pi_\theta(a|s) + \pi_\theta(a|s)\nabla_\theta Q(s,a,\theta)]\,\mathrm{d}a\,\mathrm{d}s \tag{15}$$

$$= \int_{\mathcal{S}} d^{\pi_b}(s) \int_{\mathcal{A}} \frac{\pi_b(a|s)}{\pi_b(a|s)} \pi_\theta(a|s)[Q(s,a,\theta)\nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q(s,a,\theta)]\,\mathrm{d}a\,\mathrm{d}s \tag{16}$$

$$= \mathbb{E}_{s\sim d^{\pi_b}(s), a\sim\pi_b(.|s)} \left[\frac{\pi_\theta(a|s)}{\pi_b(a|s)} \left(Q(s,a,\theta)\nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q(s,a,\theta)\right)\right] \tag{17}$$

□

Theorem 3.2. *Under standard regularity assumptions (Silver et al., 2014), for any Markov Decision Process, the following holds:*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim d^{\pi_b}(s)} \left[\nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s) + \nabla_\theta Q(s,a,\theta)|_{a=\pi_\theta(s)} \right]. \tag{12}$$

Proof.

$$\nabla_\theta J(\pi_\theta) = \int_{\mathcal{S}} d^{\pi_b}(s) \nabla_\theta Q(s,\pi_\theta(s),\theta)\,\mathrm{d}s \tag{18}$$

$$= \int_{\mathcal{S}} d^{\pi_b}(s) \left[\nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s) + \nabla_\theta Q(s,a,\theta)|_{a=\pi_\theta(s)} \right]\,\mathrm{d}s \tag{19}$$

$$= \mathbb{E}_{s\sim d^{\pi_b}(s)} \left[\nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s) + \nabla_\theta Q(s,a,\theta)|_{a=\pi_\theta(s)} \right] \tag{20}$$

□

A.3 Implementation details

A.3.1

In this appendix, we report the implementation details for PSSVF, PSVF, PAVF and the baselines. We specify for each hyperparameter, which algorithms and tasks are sharing them.

Shared hyperparameters:

- Policy architecture (continuous control tasks): We use three different deterministic policies: a linear mapping between states and actions; a single-layer MLP with 32 neurons and tanh activation; a 2-layers MLP (64,64) with tanh activations. All policies contain a bias term and are followed by a tanh nonlinearity in order to bound the action.
- Policy architecture (discrete control tasks): We use three different deterministic policies: a linear mapping between states and a probability distribution over actions; a single-layer MLP with 32 neurons and tanh activation; a 2-layers MLP (64,64) with tanh activations. The deterministic action a is obtained choosing $a = \arg\max \pi_\theta(a|s)$. All policies contain a bias term.
- Policy initialization: all weights and biases are initialized using the default Pytorch initialization for PVFs and DDPG and are set to zero for ARS.
- Critic architecture: 2-layers MLP (512,512) with bias and ReLU activation functions for PSVF, PAVF; 2-layers MLP (256,256) with bias and ReLU activation functions for DDPG.
- Critic initialization: all weights and biases are initialized using the default Pytorch initialization for PVFs and DDPG.
- Batch size: 128 for DDPG, PSVF, PAVF; 16 for PSSVF.
- Actor's frequency of updates: every episode for PSSVF; every batch of episodes for ARS; every 50 time steps for DDPG, PSVF, PAVF.

- Critic's frequency of updates: every episode for PSSVF; every 50 time steps for DDPG, PSVF, PAVF.
- Replay buffer: the size is 100k; data are sampled uniformly.
- Optimizer: Adam for PVFs and DDPG.

Tuned hyperparameters:

- Number of directions and elite directions for ARS ([directions, elite directions]): tuned with values in $[[1, 1], [4, 1], [4, 4], [16, 1], [16, 4], [16, 16]]$.
- Policy's learning rate: tuned with values in $[1e-2, 1e-3, 1e-4]$.
- Critic's learning rate: tuned with values in $[1e-2, 1e-3, 1e-4]$.
- Noise for exploration: the perturbation for the action (DDPG) or the parameter is sampled from $\mathcal{N}(0, \sigma I)$ with σ tuned with values in $[1, 1e-1]$ for PSSVF, PSVF, PAVF; $[1e-1, 1e-2]$ for DDPG; $[1, 1e-1, 1e-2, 1e-3]$ for ARS.

Environment hyperparameters:

- Environment interactions: 1M time steps for Swimmer-v3 and Hopper-v3; 100k time steps for all other environments.
- Discount factor for TD algorithms: 0.999 for Swimmer; 0.99 for all other environments.
- Survival reward in Hopper: True for DDPG, PSVF, PAVF; False for ARS, PSSVF.

Algorithm-specific hyperparameters:

- Critic's number of updates: 50 for DDPG, 5 for PSVF and PAVF; 10 for PSSVF.
- Actor's number of updates: 50 for DDPG, 1 for PSVF and PAVF; 10 for PSSVF.
- Observation normalization: False for DDPG; True for all other algorithms.
- Starting steps in DDPG (random actions and no training): first 1%.
- Polyak parameter in DDPG: 0.995.

ARS For ARS, we used the official implementation provided by the authors and we modified it in order to use nonlinear policies. More precisely, we used the implementation of ARSv2-t (Mania et al., 2018), which uses observation normalization, elite directions and an adaptive learning rate based on the standard deviation of the return collected. In order to avoid potential division by zero, which can potentially happened if all data sampled have the same return, we perform the standardization only in case the standard deviation is not zero. In the original implementation of ARS (Mania et al., 2018), the survival bonus for the reward in Hopper environment is removed in order to avoid local minima. Since we wanted our PSSVF to be close to their setting, we also applied this modification. We did not remove the survival bonus from all TD algorithms and we did not investigate how this could affect their performance. We provide a comparison of the performance of PSSVF with and without the bonus in figure 5.

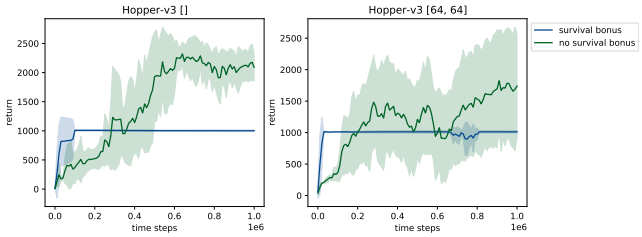

Figure 5: Performance of PSSVF with and without the survival bonus for the reward in Hopper-v3 when using the hyperparameters maximizing the average return. Learning curves are averaged over 5 seeds

DDPG For DDPG, we used the Spinning Up implementation provided by OpenAI (Achiam, 2018), which includes target networks for the actor and the critic and no learning for a fixed set of time steps, called starting steps. We did not include target networks and starting steps in our PVFs, although they could potentially help stabilizing training. The implementation of DDPG that we used (Achiam, 2018) does not use observation normalization. In preliminary experiments, we observed that DDPG did not obtain significant increase or decrease in performance when including it, hence we decided to not use it. Another difference between our TD algorithms and DDPG consists in the number of updated of the actor and the critic. Since DDPG's critic needs to keep track of the current policy, the critic and the actor are updated in a nested form, with the first's update depending on the latter and viceversa. Our PSVF and PAVF do not need to track the policy learned, hence, when it is time to update, we need only to train once the critic for many gradient steps and then train the actor for many gradient steps. This requires less compute. On the other hand, when using nonlinear policies, our PVFs suffer the curse of dimensionality. For this reason, we observed some benefits on using a bigger critic. In preliminary experiments, we observed that DDPG did not obtain significant change in performance when also equipped with a bigger critic. We show differences in performance for our methods when removing observation normalization and when using a smaller critic (MLP(256,256)) in figure 6. We observe that the performance is decreasing if observation normalization is removed. However, only for shallow policies in Swimmer and deep policies in Hopper there seems to be a significant benefit. We plan for future work to assess in which exact cases there is a benefit on having a bigger critic.

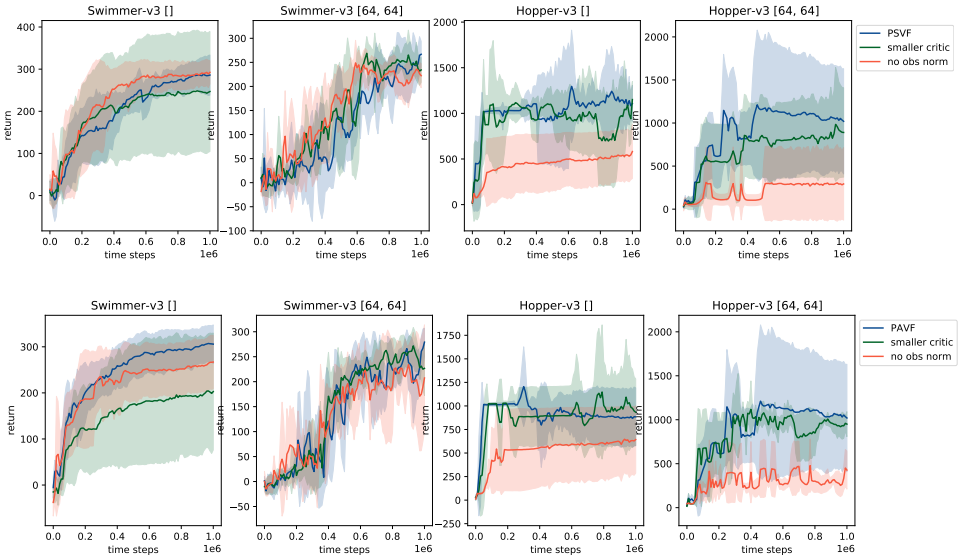

Figure 6: Learning curves for PSVF and PAVF for different environments and policies removing observation normalization and using a smaller critic. We use the hyperparameters maximizing the average return. Learning curves are averaged over 5 seeds.

Discounting in Swimmer For TD algorithms, we chose a fixed discount factor $\gamma = 0.99$ for all environments but Swimmer-v3. This environment is known to be challenging for TD based algorithms because discounting causes the agents to become too short sighted. We observed that, with the standard discounting, DDPG, PSVF and PAVF were not able to learn the task. However, making the algorithms more far sighted greatly improved their performance. In figure 7 we report the return obtained by DDPG, PSVF and PAVF for different values of the discount factor in Swimmer.

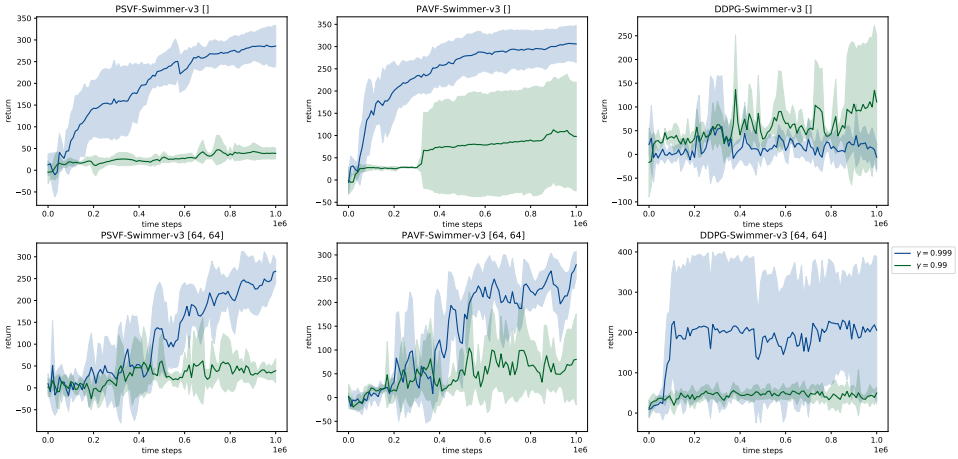

Figure 7: Effect of different choices of the discount factor in Swimmer-v3 for PSVF, PAVF and DDPG, with shallow and deep policies. We use the hyperparameters maximizing the average return. Learning curves are averaged over 5 seeds

A.3.2 PSEUDOCODE

Algorithm 2 Actor-critic with TD prediction for $V(s, \theta)$

Input: Differentiable critic $V_{\mathbf{w}} : \mathcal{S} \times \Theta \to \mathcal{R}$ with parameters $\mathbf{w}$; deterministic actor π_θ with parameters θ; empty replay buffer D

Output : Learned $V_{\mathbf{w}} \approx V(s, \theta)$, learned $\pi_\theta \approx \pi_{\theta^*}$

Initialize critic and actor weights $\mathbf{w}, \theta$
repeat:
  Observe state s, take action $a = \pi_\theta(s)$, observe reward r and next state s'
  Store (s, θ, r, s') in the replay buffer D
  if it's time to update **then**:
    for many steps **do**:
      Sample a batch $B_1 = \{(s, \tilde{\theta}, r, s')\}$ from D
      Update critic by stochastic gradient descent:
      $\nabla_{\mathbf{w}} \frac{1}{|B_1|} \mathbb{E}_{(s,\tilde{\theta},r,s') \in B_1} [V_{\mathbf{w}}(s, \tilde{\theta}) - (r + \gamma V_{\mathbf{w}}(s', \tilde{\theta}))]^2$
    end for
    for many steps **do**:
      Sample a batch $B_2 = \{(s)\}$ from D
      Update actor by gradient ascent: $\nabla_\theta \frac{1}{|B_2|} \mathbb{E}_{s \in B_2} [V_{\mathbf{w}}(s, \theta)]$
    end for
  end if
until convergence

Algorithm 3 Stochastic actor-critic with TD prediction for $Q(s, a, \theta)$

Input: Differentiable critic $Q_{\mathbf{w}} : \mathcal{S} \times \mathcal{A} \times \Theta \to \mathcal{R}$ with parameters $\mathbf{w}$; stochastic differentiable actor π_θ with parameters θ; empty replay buffer D

Output : Learned $Q_{\mathbf{w}} \approx Q(s, a, \theta)$, learned $\pi_\theta \approx \pi_{\theta^*}$

Initialize critic and actor weights $\mathbf{w}, \theta$

repeat:

  Observe state s, take action $a = \pi_\theta(s)$, observe reward r and next state s'

  Store (s, a, θ, r, s') in the replay buffer D

  if it's time to update **then**:

    for many steps **do**:

      Sample a batch $B_1 = \{(s, a, \tilde{\theta}, r, s')\}$ from D

      Update critic by stochastic gradient descent:

      $\nabla_{\mathbf{w}} \frac{1}{|B_1|} \mathbb{E}_{(s,a,\tilde{\theta},r,s') \in B_1} [Q_{\mathbf{w}}(s, a, \tilde{\theta}) - (r + \gamma Q_{\mathbf{w}}(s', a' \sim \pi_{\tilde{\theta}}(s'), \tilde{\theta}))]^2$

    end for

    for many steps **do**:

      Sample a batch $B_2 = \{(s, a, \tilde{\theta})\}$ from D

      Update actor by stochastic gradient ascent:

      $\frac{1}{|B_2|} \mathbb{E}_{(s,a,\tilde{\theta}) \in B_2} \left[\frac{\pi_\theta(a|s)}{\pi_{\tilde{\theta}}(a|s)} \left(Q(s, a, \theta) \nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q(s, a, \theta) \right) \right]$

    end for

  end if

until convergence

Algorithm 4 Deterministic actor-critic with TD prediction for $Q(s, a, \theta)$

Input: Differentiable critic $Q_{\mathbf{w}} : \mathcal{S} \times \mathcal{A} \times \Theta \to \mathcal{R}$ with parameters $\mathbf{w}$; differentiable deterministic actor π_θ with parameters θ; empty replay buffer D

Output : Learned $Q_{\mathbf{w}} \approx Q(s, a, \theta)$, learned $\pi_\theta \approx \pi_{\theta^*}$

Initialize critic and actor weights $\mathbf{w}, \theta$

repeat:

  Observe state s, take action $a = \pi_\theta(s)$, observe reward r and next state s'

  Store (s, a, θ, r, s') in the replay buffer D

  if it's time to update **then**:

    for many steps **do**:

      Sample a batch $B_1 = \{(s, a, \tilde{\theta}, r, s')\}$ from D

      Update critic by stochastic gradient descent:

      $\nabla_{\mathbf{w}} \frac{1}{|B_1|} \mathbb{E}_{(s,a,\tilde{\theta},r,s') \in B_1} [Q_{\mathbf{w}}(s, a, \tilde{\theta}) - (r + \gamma Q_{\mathbf{w}}(s', \pi_{\tilde{\theta}}(s'), \tilde{\theta}))]^2$

    end for

    for many steps **do**:

      Sample a batch $B_2 = \{(s)\}$ from D

      Update actor by stochastic gradient ascent:

      $\frac{1}{|B_2|} \mathbb{E}_{s \in B_2} [\nabla_\theta \pi_\theta(s) \nabla_a Q_{\mathbf{w}}(s, a, \theta)|_{a=\pi_\theta(s)} + \nabla_\theta Q_{\mathbf{w}}(s, a, \theta)|_{a=\pi_\theta(s)}]$

    end for

  end if

until convergence

A.4 Experimental details

A.4.1 LQR

For our visualization experiment, we employ an instance of the Linear Quadratic Regulator. Here, the agent observes a 1-D state, corresponding to its position and chooses a 1-D action. The transitions are $s' = s + a$ and there is a quadratic negative term for the reward: $R(s, a) = -s^2 - a^2$. The agent starts in state $s_0 = 1$ and acts in the environment for 50 time steps. The state space is bounded in [-2,2]. The goal of the agent is to reach and remain in the origin. The agent is expected to perform small steps towards the origin when it uses the optimal policy. For this task, we do not

use a tanh nonlinearity and we do not use observation normalization. We use a learning rate of $1e-3$ for the policy and $1e-2$ for the PSSVF. Weights are perturbed every episode using $\sigma = 0.5$. The policy is initialized with weight 3.2 and bias -3.5. All the other hyperparameters are set to their default. The true $J(\theta)$ is computed by running 10,000 policies in the environment with parameters in $[-5, -5] \times [-5, 5]$. $V_w(\theta)$ is computed by measuring the output of the PSSVF on the same set of policies. Each red arrow in figure 1 represents 200 update steps of the policy.

A.4.2 Offline experiments

Zero-shot learning In this task we used the same hyperparameters found in tables 3, 5 and 6. When new policies are trained offline, we use a learning rate of 0.02. We evaluate the performance of the policies learned from scratch evaluating them with 5 test trajectories every 5 gradient steps.

Offline learning with fragmented behaviors In this task, data are generate by perturbing a randomly initialized policy every 200 time steps and using it to act in the environment. We use $\sigma = 0.5$ for the perturbations. After the dataset is collected, the PSVF is trained using a learning rate of $1e-3$ with a batch size of 128. When the policy is learned, we use a learning rate of 0.02. All other hyperparameters are set to default values.

A.4.3 Full experimental results

Methodology In order to ensure a fair comparison of our methods and the baselines, we adopt the following procedure. For each hyperparameter configuration, for each environment and policy architecture, we run 5 instances of the learning algorithm using different seeds. We measure the learning progress by running 100 evaluations while learning the deterministic policy (without action or parameter noise) using 10 test trajectories. We use two metrics to determine the best hyperparameters: the average return over policy evaluations during the whole training process and the average return over policy evaluations during the last 20% time steps. For each algorithm, environment and policy architecture, we choose the two hyperparameter configurations maximizing the performance of the two metrics and test them on 20 new seeds, reporting average and final performance in table 1 and 2 respectively.

Figures 8 and 9 report all the learning curves from the main paper and for a small non linear policy with 32 hidden neurons.

A.4.4 Sensitivity analysis

In the following, we report the sensitivity plots for all algorithms, policies and environments. In particular, figure 10, 11, 12, 13 and 14 show the performance of each algorithm given different hyperparameters tried during training. We observe that in general deep policies are more sensitive and, apart for DDPG, achieve often a better performance than smaller policies. The higher sensitivity displayed by ARS is in part caused by the higher number of hyperparameters we tried when tuning the algorithm.

A.4.5 Table of best hyperparameters

We report for each algorithm, environment, and policy architecture the best hyperparameters found when optimizing for average return or final return in tables 3, 4, 5, 6 and 7.

Table 1: Average return with standard deviation (across 20 seeds) for hypermarameters optimizing the average return during training. Square brackets represent the number of neurons per layer of the policy. [] represents a linear policy.

Policy: []	MountainCar Continuous-v0	Inverted Pendulum-v2	Reacher -v2	Swimmer -v3	Hopper -v3
ARS	63 ± 6	886 ± 72	-9.2 ± 0.3	228 ± 89	1184 ± 345
PSSVF	85 ± 4	944 ± 33	-11.7 ± 0.9	259 ± 47	1392 ± 287
DDPG	0 ± 0	612 ± 169	-8.6 ± 0.9	95 ± 112	629 ± 145
PSVF	84 ± 20	926 ± 34	-19.7 ± 6.0	188 ± 71	917 ± 249
PAVF	82 ± 21	913 ± 40	-17.0 ± 7.7	231 ± 56	814 ± 223
Policy:[32]					
ARS	37 ± 11	851 ± 46	-9.6 ± 0.3	139 ± 78	1003 ± 66
PSSVF	60 ± 33	701 ± 138	10.4 ± 0.5	189 ± 35	707 ± 668
DDPG	0 ± 0	816 ± 36	-5.7 ± 0.3	61 ± 32	1384 ± 125
PSVF	71 ± 25	529 ± 281	-11.9 ± 1.2	226 ± 33	864 ± 272
PAVF	71 ± 27	563 ± 228	-10.9 ± 1.1	222 ± 28	793 ± 322
Policy: [64,64]					
ARS	28 ± 8	812 ± 239	-9.8 ± 0.3	129 ± 68	964 ± 47
PSSVF	72 ± 22	850 ± 93	-10.7 ± 0.2	158 ± 59	922 ± 568
DDPG	0 ± 0	834 ± 36	-5.5 ± 0.4	92 ± 117	767 ± 627
PSVF	80 ± 9	580 ± 107	-10.7 ± 0.6	137 ± 38	843 ± 282
PAVF	73 ± 10	399 ± 219	-10.7 ± 0.5	142 ± 26	875 ± 136

Policy: []	Acrobot-v1	CartPole-v1
ARS	-161 ± 23	476 ± 13
PSSVF	-137 ± 14	443 ± 105
PSVF	-148 ± 25	459 ± 28
Policy:[32]		
ARS	-296 ± 38	395 ± 141
PSSVF	-251 ± 80	463 ± 18
PSVF	-270 ± 113	413 ± 61
Policy: [64,64]		
ARS	-335 ± 35	416 ± 105
PSSVF	-281 ± 117	452 ± 34
PSVF	-397 ± 71	394 ± 71

Table 2: Final return with standard deviation (across 20 seeds) for hypermarameters optimizing the final return during training.

Policy: []	MountainCar Continuous-v0	Inverted Pendulum-v2	Reacher -v2	Swimmer -v3	Hopper -v3
ARS	73 ± 5	657 ± 477	-8.6 ± 0.5	334 ± 34	1443 ± 713
PSSVF	84 ± 28	970 ± 126	-10.0 ± 1.0	350 ± 8	1560 ± 911
DDPG	0 ± 1	777 ± 320	-7.3 ± 0.4	146 ± 152	704 ± 234
PSVF	76 ± 36	906 ± 289	-16.5 ± 1.6	238 ± 107	1067 ± 340
PAVF	68 ± 42	950 ± 223	-17.2 ± 15.4	298 ± 40	720 ± 281
Policy:[32]					
ARS	54 ± 20	936 ± 146	-9.2 ± 0.4	239 ± 117	1048 ± 68
PSSVF	89 ± 22	816 ± 234	-10.2 ± 1.0	294 ± 41	1204 ± 615
DDPG	0 ± 0	703 ± 283	-4.6 ± 0.6	179 ± 150	1290 ± 348
PSVF	84 ± 31	493 ± 462	-11.3 ± 0.8	290 ± 70	1003 ± 572
PAVF	92 ± 7	854 ± 295	-10.1 ± 0.9	307 ± 34	967 ± 411
Policy: [64,64]					
ARS	11 ± 30	976 ± 83	-9.4 ± 0.4	157 ± 54	1006 ± 47
PSSVF	91 ± 16	898 ± 227	-10.7 ± 0.6	224 ± 99	1412 ± 691
DDPG	0 ± 0	943 ± 73	-4.4 ± 0.4	196 ± 151	1437 ± 752
PSVF	93 ± 1	1000 ± 0	-10.6 ± 1.0	257 ± 26	1247 ± 344
PAVF	93 ± 2	827 ± 267	-10.6 ± 0.4	232 ± 42	1005 ± 155

Policy: []	Acrobot-v1	CartPole-v1
ARS	-126 ± 26	499 ± 2
PSSVF	-97 ± 6	482 ± 53
PSVF	-100 ± 18	500 ± 0
Policy:[32]		
ARS	-215 ± 97	471 ± 110
PSSVF	-116 ± 33	500 ± 0
PSVF	-244 ± 151	488 ± 36
Policy: [64,64]		
ARS	-182 ± 45	492 ± 18
PSSVF	-233 ± 139	500 ± 0
PSVF	-406 ± 51	499 ± 2

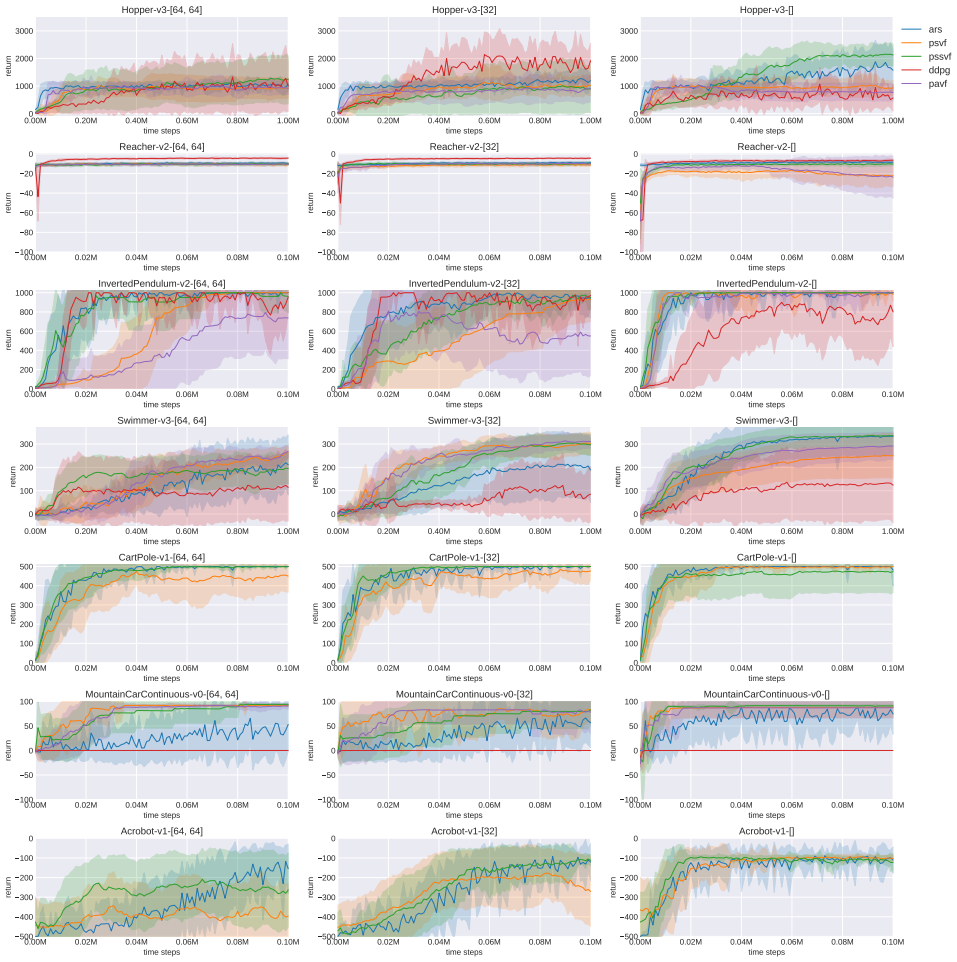

Figure 8: Learning curves representing the average return as a function of the number of time steps in the environment (across 20 runs) with different environments and policy architectures. We use the **best hyperparameters found while maximizing the average reward** for each task. For each subplot, the square brackets represent the number of neurons per policy layer. [] represents a linear policy.

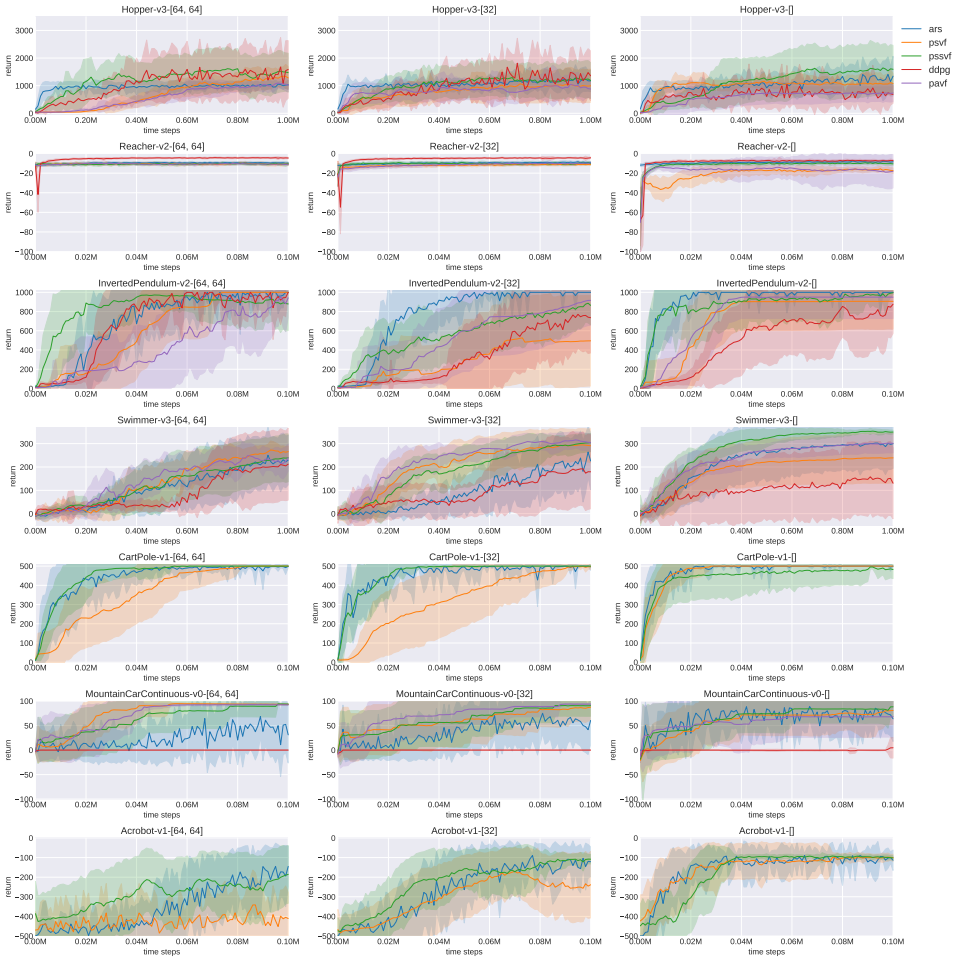

Figure 9: Learning curves representing the average return as a function of the number of time steps in the environment (across 20 runs) with different environments and policy architectures. We use the **best hyperparameters found while maximizing the final reward for each task**. For each subplot, the square brackets represent the number of neurons per policy layer. [] represents a linear policy.

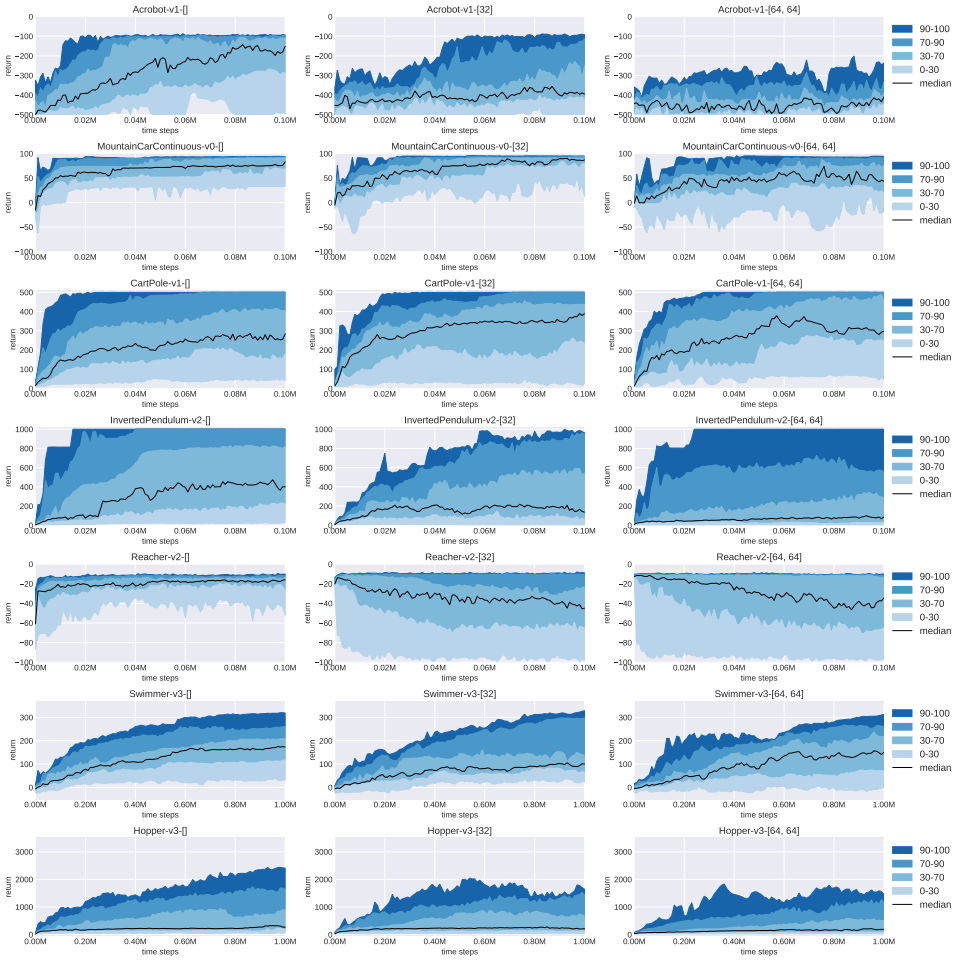

Figure 10: **Sensitivity of PSSVFs** to the choice of the hyperparameter. Performance is shown by percentile using all the learning curves obtained during hyperparameter tuning. The median performance is depicted as a dark line. For each subplot, the numbers in the square brackets represent the number of neurons per layer of the policy. [] represents a linear policy.

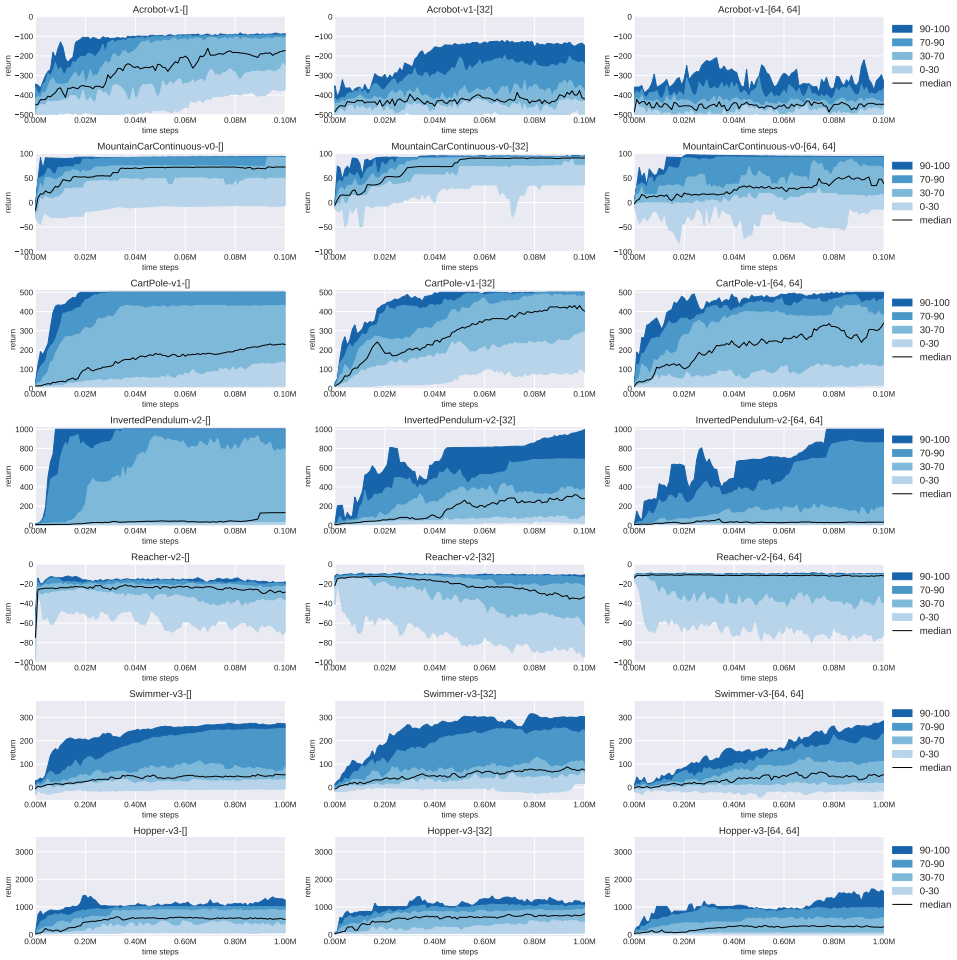

Figure 11: **Sensitivity of PSVFs** to the choice of the hyperparameter. Performance is shown by percentile using all the learning curves obtained during hyperparameter tuning. The median performance is depicted as a dark line. For each subplot, the numbers in the square brackets represent the number of neurons per layer of the policy. [] represents a linear policy.

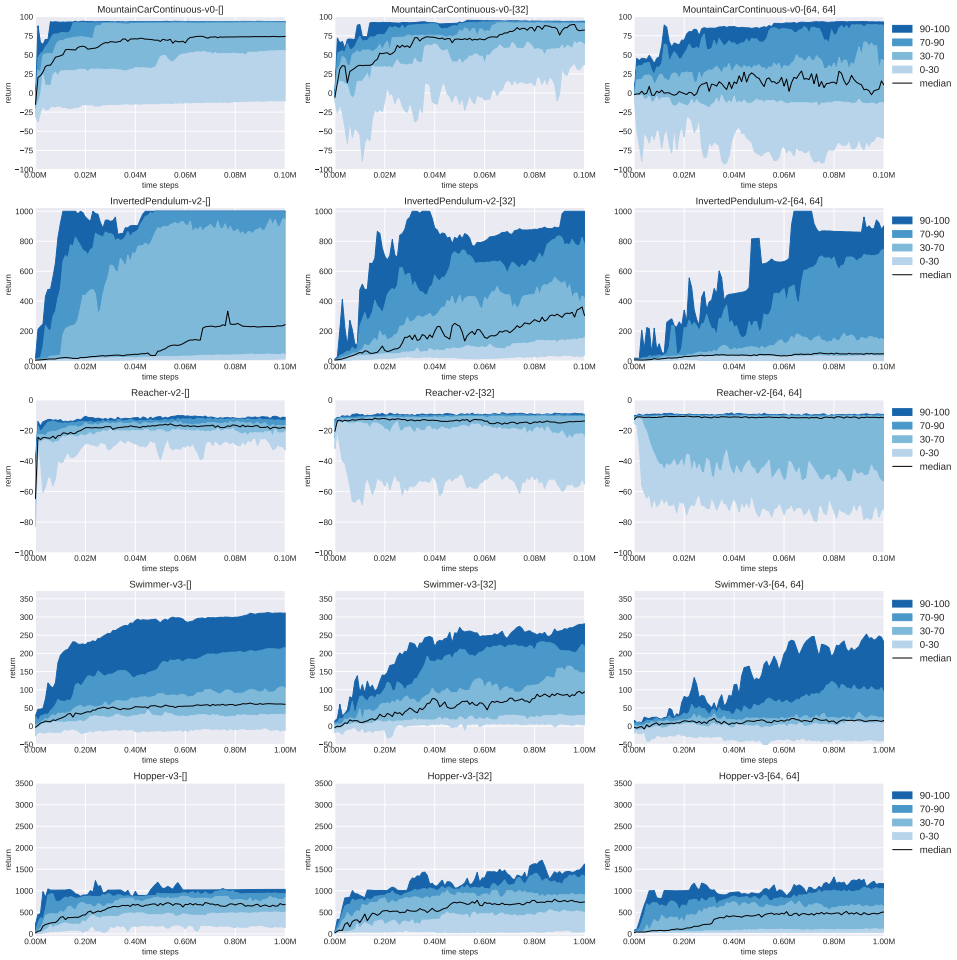

Figure 12: **Sensitivity of PAVFs** to the choice of the hyperparameter. Performance is shown by percentile using all the learning curves obtained during hyperparameter tuning. The median performance is depicted as a dark line. For each subplot, the numbers in the square brackets represent the number of neurons per layer of the policy.

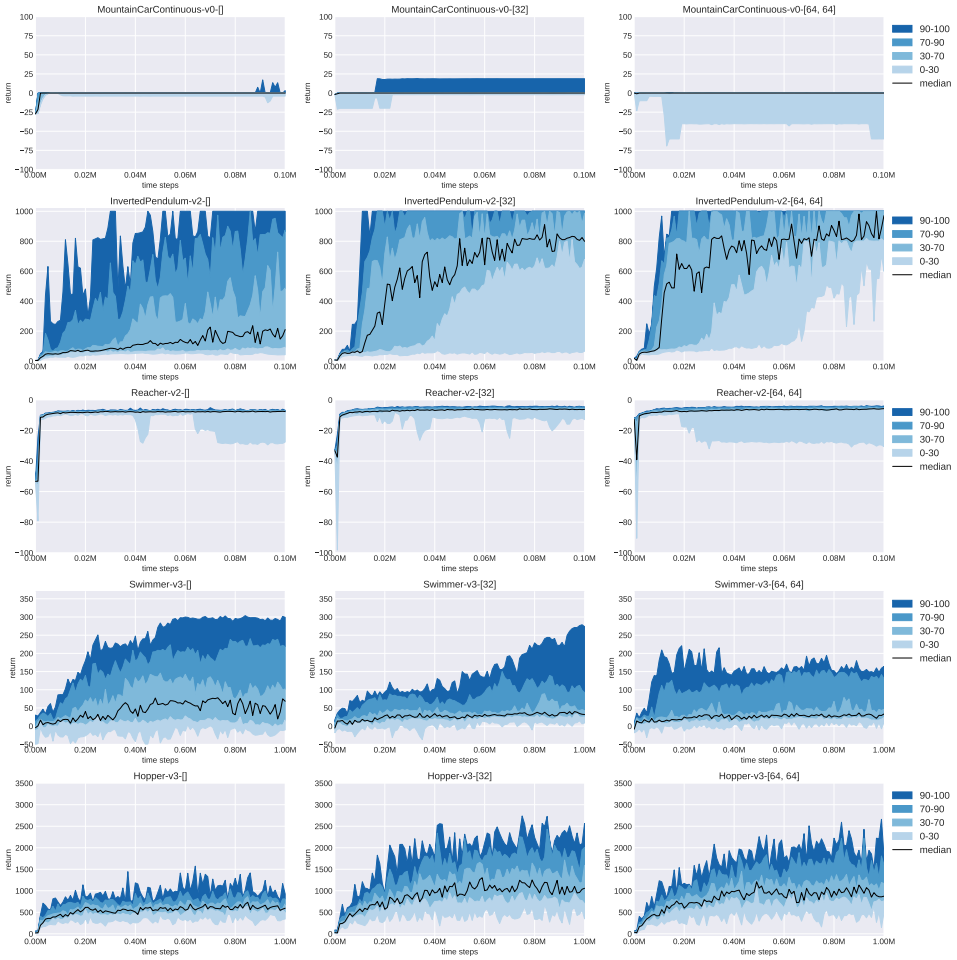

Figure 13: **Sensitivity of DDPG** to the choice of the hyperparameter. Performance is shown by percentile using all the learning curves obtained during hyperparameter tuning. The median performance is depicted as a dark line. For each subplot, the numbers in the square brackets represent the number of neurons per layer of the policy.

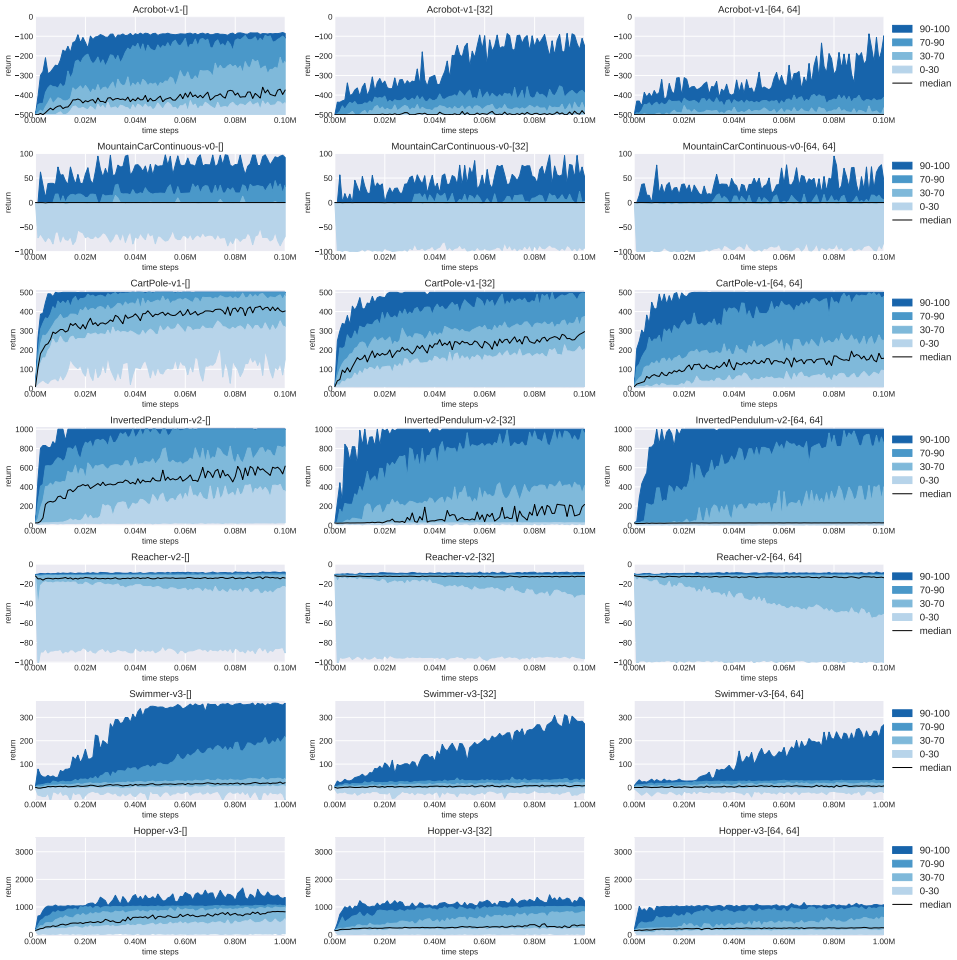

Figure 14: **Sensitivity of ARS** to the choice of the hyperparameter. Performance is shown by percentile using all the learning curves obtained during hyperparameter tuning. The median performance is depicted as a dark line. For each subplot, the numbers in the square brackets represent the number of neurons per layer of the policy.

Table 3: Table of best hyperparameters for PSSVFs

Learning rate policy	Policy:	[]		[32]		[64,64]	
	Metric:	avg	last	avg	last	avg	last
Acrobot-v1		1e-2	1e-3	1e-4	1e-4	1e-4	1e-4
MountainCarContinuous-v0		1e-2	1e-3	1e-4	1e-4	1e-4	1e-4
CartPole-v1		1e-3	1e-3	1e-3	1e-3	1e-4	1e-4
Swimmer-v3		1e-3	1e-3	1e-3	1e-3	1e-2	1e-4
InvertedPendulum-v2		1e-3	1e-3	1e-3	1e-3	1e-4	1e-4
Reacher-v2		1e-4	1e-4	1e-4	1e-4	1e-4	1e-4
Hopper-v3		1e-4	1e-4	1e-4	1e-3	1e-4	1e-4
Learning rate critic							
Acrobot-v1		1e-2	1e-3	1e-2	1e-2	1e-2	1e-2
MountainCarContinuous-v0		1e-3	1e-2	1e-3	1e-2	1e-2	1e-2
CartPole-v1		1e-2	1e-2	1e-3	1e-3	1e-2	1e-2
Swimmer-v3		1e-3	1e-3	1e-2	1e-2	1e-3	1e-2
InvertedPendulum-v2		1e-2	1e-2	1e-3	1e-2	1e-3	1e-3
Reacher-v2		1e-3	1e-3	1e-3	1e-3	1e-4	1e-4
Hopper-v3		1e-3	1e-3	1e-2	1e-2	1e-2	1e-2
Noise for exploration							
Acrobot-v1		1.0	1.0	1e-1	1e-1	1e-1	1e-1
MountainCarContinuous-v0		1.0	1.0	1e-1	1e-1	1e-1	1e-1
CartPole-v1		1.0	1.0	1.0	1.0	1e-1	1e-1
Swimmer-v3		1.0	1.0	1.0	1.0	1.0	1e-1
InvertedPendulum-v2		1.0	1.0	1.0	1.0	1e-1	1e-1
Reacher-v2		1e-1	1e-1	1e-1	1e-1	1e-1	1e-1
Hopper-v3		1.0	1.0	1e-1	1.0	1e-1	1e-1

Table 4: Table of best hyperparameters for ARS

Learning rate policy	Policy:	[]		[32]		[64,64]	
	Metric:	avg	last	avg	last	avg	last
Acrobot-v1		1e-2	1e-3	1e-2	1e-2	1e-2	1e-2
MountainCarContinuous-v0		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
CartPole-v1		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
Swimmer-v3		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
InvertedPendulum-v2		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
Reacher-v2		1e-2	1e-2	1e-3	1e-2	1e-3	1e-3
Hopper-v3		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
Number of directions and elite directions							
Acrobot-v1		(4,4)	(4,4)	(1,1)	(1,1)	(1,1)	(1,1)
MountainCarContinuous-v0		(1,1)	(1,1)	(1,1)	(16,4)	(1,1)	(1,1)
CartPole-v1		(4,4)	(4,4)	(1,1)	(1,1)	(4,1)	(4,1)
Swimmer-v3		(1,1)	(1,1)	(1,1)	(4,1)	(1,1)	(1,1)
InvertedPendulum-v2		(4,4)	(4,4)	(1,1)	(4,4)	(4,1)	(16,1)
Reacher-v2		(16,16)	(16,16)	(1,1)	(16,4)	(1,1)	(1,1)
Hopper-v3		(4,1)	(4,1)	(1,1)	(1,1)	(1,1)	(1,1)
Noise for exploration							
Acrobot-v1		1e-2	1e-3	1e-1	1e-1	1e-1	1e-1
MountainCarContinuous-v0		1e-1	1e-1	1e-1	1e-1	1e-1	1e-1
CartPole-v1		1e-2	1e-2	1e-1	1e-1	1e-2	1e-2
Swimmer-v3		1e-1	1e-1	1e-2	1e-1	1e-1	1e-1
InvertedPendulum-v2		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
Reacher-v2		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
Hopper-v3		1e-1	1e-1	1e-1	1e-1	1e-1	1e-1

Table 5: Table of best hyperparameters for PSVFs

Learning rate policy	Policy:	[]		[32]		[64,64]	
	Metric:	avg	last	avg	last	avg	last
Acrobot-v1		1e-2	1e-2	1e-4	1e-4	1e-4	1e-2
MountainCarContinuous-v0		1e-2	1e-3	1e-2	1e-4	1e-3	1e-4
CartPole-v1		1e-2	1e-2	1e-2	1e-4	1e-3	1e-4
Swimmer-v3		1e-3	1e-3	1e-3	1e-3	1e-3	1e-3
InvertedPendulum-v2		1e-2	1e-3	1e-4	1e-4	1e-4	1e-4
Reacher-v2		1e-3	1e-2	1e-4	1e-4	1e-4	1e-4
Hopper-v3		1e-3	1e-3	1e-4	1e-4	1e-4	1e-3
Learning rate critic							
Acrobot-v1		1e-3	1e-4	1e-2	1e-2	1e-3	1e-2
MountainCarContinuous-v0		1e-4	1e-3	1e-2	1e-4	1e-3	1e-3
CartPole-v1		1e-2	1e-2	1e-2	1e-3	1e-2	1e-4
Swimmer-v3		1e-4	1e-4	1e-4	1e-4	1e-4	1e-4
InvertedPendulum-v2		1e-3	1e-2	1e-3	1e-4	1e-4	1e-3
Reacher-v2		1e-2	1e-2	1e-3	1e-3	1e-4	1e-4
Hopper-v3		1e-2	1e-2	1e-4	1e-4	1e-2	1e-4
Noise for exploration							
Acrobot-v1		1.0	1.0	1e-1	1e-1	1e-1	1e-1
MountainCarContinuous-v0		1.0	1e-1	1e-1	1.0	1e-1	1e-1
CartPole-v1		1.0	1.0	1.0	1e-1	1e-1	1e-1
Swimmer-v3		1.0	1.0	1.0	1.0	1.0	1.0
InvertedPendulum-v2		1.0	1.0	1e-1	1e-1	1e-1	1e-1
Reacher-v2		1.0	1.0	1.0	1.0	1e-1	1e-1
Hopper-v3		1.0	1.0	1e-1	1e-1	1e-1	1.0

Table 6: Table of best hyperparameters for PAVFs

Learning rate policy	Policy:	[]		[32]		[64,64]	
	Metric:	avg	last	avg	last	avg	last
MountainCarContinuous-v0		1e-2	1e-3	1e-3	1e-4	1e-4	1e-4
Swimmer-v3		1e-3	1e-3	1e-3	1e-3	1e-3	1e-3
InvertedPendulum-v2		1e-2	1e-3	1e-3	1e-4	1e-4	1e-4
Reacher-v2		1e-3	1e-3	1e-4	1e-4	1e-4	1e-4
Hopper-v3		1e-3	1e-4	1e-4	1e-4	1e-4	1e-3
Learning rate critic							
MountainCarContinuous-v0		1e-4	1e-4	1e-4	1e-3	1e-4	1e-3
Swimmer-v3		1e-4	1e-4	1e-4	1e-4	1e-4	1e-4
InvertedPendulum-v2		1e-3	1e-2	1e-2	1e-4	1e-2	1e-3
Reacher-v2		1e-3	1e-3	1e-3	1e-2	1e-3	1e-3
Hopper-v3		1e-4	1e-3	1e-3	1e-2	1e-4	1e-3
Noise for exploration							
MountainCarContinuous-v0		1.0	1e-1	1e-1	1e-1	1e-1	1e-1
Swimmer-v3		1.0	1.0	1.0	1.0	1.0	1.0
InvertedPendulum-v2		1.0	1.0	1e-1	1e-1	1e-1	1e-1
Reacher-v2		1e-1	1e-1	1e-1	1.0	1.0	1.0
Hopper-v3		1.0	1.0	1e-1	1e-1	1e-1	1.0

Table 7: Table of best hyperparameters for DDPG

Learning rate policy	Policy:	[]		[32]		[64,64]	
	Metric:	avg	last	avg	last	avg	last
MountainCarContinuous-v0		1e-2	1e-2	1e-2	1e-4	1e-3	1e-3
Swimmer-v3		1e-3	1e-3	1e-2	1e-2	1e-2	1e-2
InvertedPendulum-v2		1e-4	1e-4	1e-3	1e-3	1e-3	1e-4
Reacher-v2		1e-4	1e-3	1e-2	1e-2	1e-3	1e-3
Hopper-v3		1e-2	1e-2	1e-2	1e-4	1e-2	1e-2
Learning rate critic							
MountainCarContinuous-v0		1e-4	1e-4	1e-4	1e-3	1e-3	1e-3
Swimmer-v3		1e-3	1e-3	1e-3	1e-3	1e-2	1e-3
InvertedPendulum-v2		1e-3	1e-3	1e-3	1e-4	1e-3	1e-3
Reacher-v2		1e-3	1e-3	1e-3	1e-3	1e-3	1e-3
Hopper-v3		1e-3	1e-3	1e-4	1e-4	1e-4	1e-4
Noise for exploration							
MountainCarContinuous-v0		1e-2	1e-2	1e-2	1e-1	1e-1	1e-1
Swimmer-v3		1e-1	1e-1	1e-2	1e-2	1e-2	1e-1
InvertedPendulum-v2		1e-1	1e-1	1e-2	1e-2	1e-2	1e-2
Reacher-v2		1e-1	1e-2	1e-1	1e-1	1e-1	1e-1
Hopper-v3		1e-1	1e-1	1e-1	1e-2	1e-1	1e-2