Simple random search of static linear policies is competitive for reinforcement learning

Horia Mania
hmania@berkeley.edu

Aurelia Guy
lia@berkeley.edu

Benjamin Recht
brecht@berkeley.edu

Department of Electrical Engineering and Computer Science
University of California, Berkeley

Abstract

Model-free reinforcement learning aims to offer off-the-shelf solutions for controlling dynamical systems without requiring models of the system dynamics. We introduce a model-free random search algorithm for training static, linear policies for continuous control problems. Common evaluation methodology shows that our method matches state-of-the-art sample efficiency on the benchmark MuJoCo locomotion tasks. Nonetheless, more rigorous evaluation reveals that the assessment of performance on these benchmarks is optimistic. We evaluate the performance of our method over hundreds of random seeds and many different hyperparameter configurations for each benchmark task. This extensive evaluation is possible because of the small computational footprint of our method. Our simulations highlight a high variability in performance in these benchmark tasks, indicating that commonly used estimations of sample efficiency do not adequately evaluate the performance of RL algorithms. Our results stress the need for new baselines, benchmarks and evaluation methodology for RL algorithms.

1 Introduction

Model-free reinforcement learning (RL) aims to offer off-the-shelf solutions for controlling dynamical systems without requiring models of the system dynamics. Such methods have successfully produced RL agents that surpass human players in video games and games such as Go [16, 28]. Although these results are impressive, model-free methods have not yet been successfully deployed to control physical systems, outside of research demos. There are several factors prohibiting the adoption of model-free RL methods for controlling physical systems: the methods require too much data to achieve reasonable performance, the ever-increasing assortment of RL methods makes it difficult to choose what is the best method for a specific task, and many candidate algorithms are difficult to implement and deploy [11].

Unfortunately, the current trend in RL research has put these impediments at odds with each other. In the quest to find methods that are *sample efficient* (i.e. methods that need little data) the general trend has been to develop increasingly complicated methods. This increasing complexity has led to a reproducibility crisis. Recent studies demonstrate that many RL methods are not robust to changes in hyperparameters, random seeds, or even different implementations of the same algorithm [11, 12]. Algorithms with such fragilities cannot be integrated into mission critical control systems without significant simplification and robustification.

Furthermore, it is common practice to evaluate and compare new RL methods by applying them to video games or simulated continuous control problems and measure their performance over a small number of independent trials (i.e., fewer than ten random seeds) [8–10, 14, 17, 19, 21–27, 31, 32]. The most popular continuous control benchmarks are the MuJoCo locomotion tasks [3, 29], with

the Humanoid model being considered "one of the most challenging continuous control problems solvable by state-of-the-art RL techniques [23]." In principle, one can use video games and simulated control problems for beta testing new ideas, but simple baselines should be established and thoroughly evaluated before moving towards more complex solutions.

To this end, we aim to determine *the simplest* model-free RL method that can solve standard benchmarks. Recently, two different directions have been proposed for simplifying RL. Salimans et al. [23] introduced a derivative-free policy optimization method, called Evolution Strategies. The authors showed that, for several RL tasks, their method can easily be parallelized to train policies faster than other methods. While the method of Salimans et al. [23] is simpler than previously proposed methods, it employs several complicated algorithmic elements, which we discuss at the end of Section 3. As a second simplification to model-free RL, Rajeswaran et al. [22] have shown that linear policies can be trained via natural policy gradients to obtain competitive performance on the MuJoCo locomotion tasks, showing that complicated neural network policies are not needed to solve these continuous control problems. In this work, we combine ideas from the work of Salimans et al. [23] and Rajeswaran et al. [22] to obtain the simplest model-free RL method yet, a derivative-free optimization algorithm for training static, linear policies. We demonstrate that a simple random search method can match or exceed state-of-the-art sample efficiency on the MuJoCo locomotion tasks, included in the OpenAI Gym.

Henderson et al. [11] and Islam et al. [12] pointed out that standard evaluation methodology does not accurately capture the performance of RL methods by showing that existing RL algorithms exhibit high sensitivity to both the choice of random seed and the choice of hyperparameters. We show similar limitations of common evaluation methodology through a different lens. We exhibit a simple derivative free optimization algorithm which matches or surpasses the performance of more complex methods when using the same evaluation methodology. However, a more thorough evaluation of ARS reveals worse performance. Moreover, our method uses static linear policies and a simple local exploration scheme, which might be limiting for more difficult RL tasks. Therefore, better evaluation schemes are needed for determining the benefits of more complex RL methods. Our contributions are as follows:

- In Section 3, for applications to continuous control, we augment a basic random search method with three simple features. First, we scale each update step by the standard deviation of the rewards collected for computing that update step. Second, we normalize the system's states by online estimates of their mean and standard deviation. Third, we discard from the computation of the update steps the directions that yield the least improvement of the reward. We refer to this method as *Augmented Random Search* (ARS).

- In Section 4, we evaluate the performance of ARS on the benchmark MuJoCo locomotion tasks, included in the OpenAI Gym. Our method learns static, linear policies that achieve high rewards on all MuJoCo tasks. No neural networks are used, and yet state-of-the-art average rewards are achieved. For example, for Humanoid-v1 ARS finds linear policies which achieve average rewards of over 11500, the highest value reported in the literature. To put ARS on equal footing with competing methods, we evaluate its sample complexity over three random seeds and compare it to results reported in the literature [9, 22, 23, 26]. ARS matches or exceeds state-of-the-art sample efficiency on the locomotion tasks when using standard evaluation methodology.

- For a more thorough evaluation, we measured the performance of ARS over a hundred random seeds and also evaluated its sensitivity to hyperparameter choices. Though ARS successfully trains policies for the MuJoCo tasks a large fraction of the time when hyperparameters and random seeds are varied, ARS exhibits large variance. We measure the frequency with which ARS finds policies that yield suboptimal locomotion gaits.

2   Problem setup

Problems in reinforcement learning require finding policies for controlling dynamical systems that maximize an average reward. Such problems can be abstractly formulated as

$$\max_{\theta \in \mathbb{R}^d} \ \mathbb{E}_\xi \left[r(\pi_\theta, \xi) \right] , \tag{1}$$

2

where θ parametrizes a policy $\pi_\theta \colon \mathbb{R}^n \to \mathbb{R}^p$. The random variable ξ encodes the randomness of the environment, i.e., random initial states and stochastic transitions. The value $r(\pi_\theta, \xi)$ is the reward achieved by the policy π_θ on one trajectory generated from the system. In general one could use stochastic policies π_θ, but our proposed method uses deterministic policies.

Basic random search. Note that the problem formulation (1) aims to optimize reward by directly optimizing over the policy parameters θ. We consider methods which explore in the parameter space rather than the action space. This choice renders RL training equivalent to derivative-free optimization with noisy function evaluations. One of the simplest and oldest optimization methods for derivative-free optimization is *random search* [15].

A primitive form of random search, which we call *basic random search* (BRS), simply computes a finite difference approximation along the random direction and then takes a step along this direction without using a line search. Our method ARS, described in Section 3, is based on this simple strategy. For updating the parameters θ of a policy π_θ, BRS and ARS exploit update directions of the form:

$$\frac{r(\pi_{\theta+\nu\delta}, \xi_1) - r(\pi_{\theta-\nu\delta}, \xi_2)}{\nu}, \tag{2}$$

for two i.i.d. random variables ξ_1 and ξ_2, ν a positive real number, and δ a zero mean Gaussian vector. It is known that such an update increment is an unbiased estimator of the gradient with respect to θ of $\mathbb{E}_\delta \mathbb{E}_\xi \left[r(\pi_{\theta+\nu\delta}, \xi) \right]$, a smoothed version of the objective (1) which is close to the original objective when ν is small [20]. When the function evaluations are noisy, minibatches can be used to reduce the variance in this gradient estimate. Evolution Strategies is a version of this algorithm with several complicated algorithmic enhancements [23]. Another version of this algorithm is called Bandit Gradient Descent by Flaxman et al. [6]. The convergence of random search methods for derivative free optimization has been understood for several types of convex optimization [1, 2, 13, 20]. Jamieson et al. [13] offer an information theoretic lower bound for derivative free convex optimization and show that a coordinate based random search method achieves the lower bound with nearly optimal dependence on the dimension.

The rewards $r(\pi_{\theta+\nu\delta}, \xi_1)$ and $r(\pi_{\theta-\nu\delta}, \xi_2)$ in Eq. (2) are obtained by collecting two trajectories from the dynamical system of interest, according to the policies $\pi_{\theta+\nu\delta}$ and $\pi_{\theta-\nu\delta}$, respectively. The random variables ξ_1, ξ_2, and δ are mutually independent, and independent from previous trajectories. One trajectory is called an *episode* or a *rollout*. The goal of RL algorithms is to approximately solve problem (1) by using as few rollouts from the dynamical system as possible.

3   Our proposed algorithm

We now introduce the Augmented Random Search (ARS) method, which relies on three augmentations of BRS that build on successful heuristics employed in deep reinforcement learning. Throughout the rest of the paper we use M to denote the parameters of policies because our method uses linear policies, and hence M is a $p \times n$ matrix. The different versions of ARS are detailed in Algorithm 1.

The first version, ARS **V1**, is obtained from BRS by scaling the update steps by the standard deviation σ_R of the rewards collected at each iteration; see Line 7 of Algorithm 1. As shown in Section 4, ARS **V1** can train linear policies, which achieve the reward thresholds previously proposed in the literature, for five MuJoCo benchmarks. However, ARS **V1** requires a larger number of episodes, and it cannot train policies for the Humanoid-v1 task. To address these issues in Algorithm 1 we also propose ARS **V2**. This version of ARS trains policies which are linear maps of states normalized by a mean and standard deviation computed online. Finally, to further enhance the performance of ARS, we introduce a third algorithmic enhancement, shown in Algorithm 1 as ARS **V1-t** and ARS **V2-t**. These versions of ARS can drop perturbation directions that yield the least improvement of the reward. Now, we motivate and offer intuition for each of these algorithmic elements.

Scaling by the standard deviation σ_R. As the training of policies progresses, random search in the parameter space of policies can lead to large variations in the rewards observed across iterations. As a result, it is difficult to choose a fixed step-size α which does not allow harmful variations in the size of the update steps. Salimans et al. [23] address this issue by transforming the rewards into rankings and then using the adaptive optimization algorithm Adam for computing the update step. Both of

---

Algorithm 1 Augmented Random Search (ARS): four versions **V1**, **V1-t**, **V2** and **V2-t**

1: **Hyperparameters:** step-size α, number of directions sampled per iteration N, standard deviation of the exploration noise ν, number of top-performing directions to use b ($b < N$ is allowed only for **V1-t** and **V2-t**)
2: **Initialize:** $M_0 = \mathbf{0} \in \mathbb{R}^{p \times n}$, $\mu_0 = \mathbf{0} \in \mathbb{R}^n$, and $\Sigma_0 = \mathbf{I}_n \in \mathbb{R}^{n \times n}$, $j = 0$.
3: **while** ending condition not satisfied **do**
4:     Sample $\delta_1, \delta_2, \ldots, \delta_N$ in $\mathbb{R}^{p \times n}$ with i.i.d. standard normal entries.
5:     Collect $2N$ rollouts of horizon H and their corresponding rewards using the $2N$ policies

$$\textbf{V1:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu\delta_k)x \\ \pi_{j,k,-}(x) = (M_j - \nu\delta_k)x \end{cases}$$

$$\textbf{V2:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu\delta_k) \operatorname{diag}(\Sigma_j)^{-1/2}(x - \mu_j) \\ \pi_{j,k,-}(x) = (M_j - \nu\delta_k) \operatorname{diag}(\Sigma_j)^{-1/2}(x - \mu_j) \end{cases}$$

    for $k \in \{1, 2, \ldots, N\}$.
6:     **V1-t, V2-t:** Sort the directions δ_k by $\max\{r(\pi_{j,k,+}), r(\pi_{j,k,-})\}$, denote by $\delta_{(k)}$ the k-th largest direction, and by $\pi_{j,(k),+}$ and $\pi_{j,(k),-}$ the corresponding policies.
7:     Make the update step:

$$M_{j+1} = M_j + \frac{\alpha}{b\sigma_R} \sum_{k=1}^{b} \left[r(\pi_{j,(k),+}) - r(\pi_{j,(k),-}) \right] \delta_{(k)},$$

    where σ_R is the standard deviation of the $2b$ rewards used in the update step.
8:     **V2:** Set μ_{j+1}, Σ_{j+1} to be the mean and covariance of the $2NH(j + 1)$ states encountered from the start of training.[1]
9:     $j \leftarrow j + 1$
10: **end while**
.

---

these techniques change the direction of the updates, obfuscating the behavior of the algorithm and making it difficult to ascertain the objective Evolution Strategies is actually optimizing. Instead, to address the large variations of the differences $r(\pi_{M+\nu\delta}) - r(\pi_{M-\nu\delta})$, we scale the update steps by the standard deviation σ_R of the $2N$ rewards collected at each iteration (see Line 7 of Algorithm 1).

While training a policy for Humanoid-v1, we observed that the standard deviations σ_R have an increasing trend; see Figure 2 in Appendix A.2. This behavior occurs because perturbations of the policy weights at high rewards can cause Humanoid-v1 to fall early, yielding large variations in the rewards collected. Without scaling the update steps by σ_R, eventually random search would take update steps which are a thousand times larger than in the beginning of training. Therefore, σ_R adapts the step sizes according to the local sensitivity of the rewards to perturbations of the policy parameters. The same training performance could probably be obtained by tuning a step size schedule. However, one of our goals was to minimize the amount of tuning required.

Normalization of the states. The normalization of states used by ARS **V2** is akin to data whitening for regression tasks. Intuitively, it ensures that policies put equal weight on the different components of the states. To see why this might help, suppose that a state coordinate only takes values in the range $[90, 100]$ while another state component takes values in the range $[-1, 1]$. Then, small changes in the control gain with respect to the first state coordinate would lead to larger changes in the actions than the same sized changes with respect to the second state component. Hence, state normalization allows different state components to have equal influence during training.

Previous work has also implemented such state normalization for fitting a neural network model for several MuJoCo environments [19]. A similar normalization is used by ES as part of the virtual batch

---

[1] Of course, we implement this in an efficient way that does not require the storage of all the states. Also, we only keep track of the diagonal of Σ_{j+1}. Finally, to ensure that the ratio $0/0$ is treated as 0, if a diagonal entry of Σ_j is smaller than 10^{-8} we make it equal to $+\infty$.

normalization of the neural network policies [23]. In the case of ARS, the state normalization can be seen as a form of non-isotropic exploration in the parameter space of linear policies.

The main empirical motivation for ARS **V2** comes from the Humanoid-v1 task. We were not able to train a linear policy for this task without the normalization of the states described in Algorithm 1. Moreover, ARS **V2** performs better than ARS **V1** on other MuJoCo tasks as well, as shown in Section 4. However, the usefulness of state normalization is likely to be problem specific.

Using top performing directions. To further improve the performance of ARS on the MuJoCo locomotion tasks, we propose ARS **V1-t** and **V2-t**. In the update steps used by ARS **V1** and **V2** each perturbation direction δ_k is weighted by the difference of the rewards $r(\pi_{j,k,+})$ and $r(\pi_{j,k,-})$. If $r(\pi_{j,k,+}) > r(\pi_{j,k,-})$, ARS pushes the policy weights M_j in the direction of δ_k. If $r(\pi_{j,k,+}) < r(\pi_{j,k,-})$, ARS pushes the policy weights M_j in the direction of $-\delta_k$. However, since $r(\pi_{j,k,+})$ and $r(\pi_{j,k,-})$ are noisy evaluations of the performance of the policies parametrized by $M_j + \nu\delta_k$ and $M_j - \nu\delta_k$, ARS **V1** and **V2** might push the weights M_j in the direction δ_k even when $-\delta_k$ is better, or vice versa. Moreover, there can be perturbation directions δ_k such that updating the policy weights M_j in either the direction δ_k or $-\delta_k$ would lead to sub-optimal performance. To address these issues, ARS **V1-t** and **V2-t** order decreasingly the perturbation directions δ_k, according to $\max\{r(\pi_{j,k,+}), r(\pi_{j,k,-})\}$, and then use only the top b directions for updating the policy weights; see Line 7 of Algorithm 1.

This algorithmic enhancement intuitively improves the performance of ARS because it ensures that the update steps are an average over directions that obtained high rewards. However, without theoretical investigation we cannot be certain of the effect of using this algorithmic enhancement, i.e., choosing $b < N$. When $b = N$ versions **V1-t** and **V2-t** are equivalent to **V1** and **V2**. Therefore, it is certain that after tuning ARS **V1-t** and **V2-t**, they will not perform any worse than ARS **V1** and **V2**.

Comparison to Salimans et al. [23]. ARS simplifies Evolution Strategies in several ways. First, ES feeds the gradient estimate into the Adam algorithm. Second, instead of using the actual reward values $r(\theta \pm \sigma\epsilon_i)$, ES transforms the rewards into rankings and uses the ranks to compute update steps. The rankings are used to make training more robust. Instead, our method scales the update steps by the standard deviation of the rewards. Third, ES bins the action space of the Swimmer-v1 and Hopper-v1 to encourage exploration. Our method surpasses ES without such binning. Fourth, ES relies on policies parametrized by neural networks with virtual batch normalization, while we show that ARS achieves state-of-the-art performance with linear policies.

4   Empirical results on the MuJoCo locomotion tasks

Implementation details. We implemented a parallel version of Algorithm 1 using the Python library Ray [18]. To avoid the computational bottleneck of communicating perturbations δ, we created a shared noise table which stores independent standard normal entries. Then, instead of communicating perturbations δ, the workers communicate indices in the shared noise table. This approach has been used in the implementation of Evolution Strategies by Moritz et al. [18] and is similar to the approach proposed by Salimans et al. [23]. Our code sets the random seeds for the random generators of all the workers and for all copies of the OpenAI Gym environments held by the workers. All these random seeds are distinct and are a function of a single integer to which we refer as *the random seed*. Furthermore, we made sure that the states and rewards produced during the evaluation rollouts were not used in any form during training.

We evaluate the performance of ARS on the MuJoCo locomotion tasks included in the OpenAI Gym-v0.9.3 [3, 29]. The OpenAI Gym provides benchmark reward functions for the different MuJoCo locomotion tasks. We used these default reward functions for evaluating the performance of the linear policies trained with ARS. The reported rewards obtained by a policy were averaged over 100 independent rollouts. For the Hopper-v1, Walker2d-v1, Ant-v1, and Humanoid-v1 tasks the default reward functions include a survival bonus, which rewards RL agents with a constant reward at each timestep, as long as a termination condition (i.e., falling over) has not been reached. During training, we removed these survival bonuses, a choice we motivate in Appendix A.1. We also defer to Appendix A.3 the sensitivity analysis of ARS to the choice of hyperparameters.

Three random seeds evaluation: We compare the different versions of ARS to the following methods: Trust Region Policy Optimization (TRPO), Deep Deterministic Policy Gradient (DDPG), Natural Gradients (NG), Evolution Strategies (ES), Proximal Policy Optimization (PPO), Soft Actor Critic (SAC), Soft Q-Learning (SQL), A2C, and the Cross Entropy Method (CEM). For the performance of these methods we used values reported by Rajeswaran et al. [22], Salimans et al. [23], Schulman et al. [26], and Haarnoja et al. [9]. In light of well-documented reproducibility issues of reinforcement learning methods [11, 12], reporting the values listed in papers rather than rerunning these algorithms casts prior work in the most favorable light possible.

Rajeswaran et al. [22] and Schulman et al. [26] evaluated the performance of RL algorithms on three random seeds, while Salimans et al. [23] and Haarnoja et al. [9] used six and five random seeds respectively. To put all methods on equal footing, for the evaluation of ARS, we sampled three random seeds uniformly from the interval $[0, 1000)$ and fixed them. For each of the six OpenAI Gym MuJoCo locomotion tasks we chose a grid of hyperparameters[2], shown in Appendix A.6, and for each set of hyperparameters we ran ARS **V1**, **V2**, **V1-t**, and **V2-t** three times, once for each of the three fixed random seeds.

Table 1 shows the average number of episodes required by ARS, NG, and TRPO to reach a prescribed reward threshold, using the values reported by Rajeswaran et al. [22] for NG and TRPO. For each version of ARS and each MuJoCo task we chose the hyperparameters which minimize the average number of episodes required to reach the reward threshold. The corresponding training curves of ARS are shown in Figure 3 of Appendix A.2. For all MuJoCo tasks, except Humanoid-v1, we used the same reward thresholds as Rajeswaran et al. [22]. Our choice to increase the reward threshold for Humanoid-v1 is motivated by the presence of the survival bonuses, as discussed in Appendix A.1.

Task	Threshold	Average # episodes to reach reward threshold				NG-lin	NG-rbf	TRPO-nn
		ARS						
		V1	V1-t	V2	V2-t			
Swimmer-v1	325	100	100	427	427	1450	1550	N/A [3]
Hopper-v1	3120	89493	51840	3013	1973	13920	8640	10000
HalfCheetah-v1	3430	10240	8106	2720	1707	11250	6000	4250
Walker2d-v1	4390	392000	166133	89600	24000	36840	25680	14250
Ant-v1	3580	101066	58133	60533	20800	39240	30000	73500
Humanoid-v1	6000	N/A	N/A	142600	142600	≈130000	≈130000	UNK[4]

Table 1: A comparison of ARS, NG, and TRPO on the MuJoCo locomotion tasks. For each task we show the average number of episodes required to achieve a prescribed reward threshold, averaged over three random seeds. We estimated the number of episodes required by NG to reach a reward of 6000 for Humanoid-v1 based on the learning curves presented by Rajeswaran et al. [22].

Table 1 shows that ARS **V1** can train policies for all tasks except Humanoid-v1, which is successfully solved by ARS **V2**. Secondly, we note that ARS **V2** reaches the prescribed thresholds for Swimmer-v1, Hopper-v1, and HalfCheetah-v1 faster than NG or TRPO, and matches the performance of NG on the Humanoid-v1. On Walker2d-v1 and Ant-v1, ARS **V2** is outperformed by NG. Nonetheless, ARS **V2-t** surpasses the performance of NG on these two tasks. Although TRPO hits the reward threshold for Walker2d-v1 faster than ARS, our method either matches or surpasses TRPO in the metrics reported by Haarnoja et al. [9] and Schulman et al. [26].

Precise comparisons to more RL methods are provided in Appendix A.2. Here we offer a summary. Salimans et al. [23] reported the average number of episodes required by ES to reach a prescribed reward threshold, on four of the locomotion tasks. ARS surpassed ES on all of those tasks. Haarnoja et al. [9] reported the maximum reward achieved by SAC, DDPG, SQL, and TRPO after a prescribed number of timesteps, on four of the locomotion tasks. With the exception of SAC on HalfCheetah-v1 and Ant-v1, ARS outperformed competing methods. Schulman et al. [26] reported the maximum reward achieved by PPO, A2C, CEM, and TRPO after a prescribed number of timesteps, on four of

---

[2]Recall that ARS **V1** and **V2** take in only three hyperparameters: the step-size α, the number of perturbation directions N, and scale of the perturbations ν. ARS **V1-t** and **V2-t** take in an additional hyperparameter, the number of top directions used b ($b \leq N$).

[3]N/A means that the method did not reach the reward threshold.

[4]UNK stands for unknown.

the locomotion tasks. With the exception of PPO on Walker2d-v1, ARS matched or surpassed the performance of competing methods.

A hundred seeds evaluation: For a more thorough evaluation of ARS, we sampled 100 distinct random seeds uniformly at random from the interval $[0, 10000)$. Then, using the hyperparameters selected for Table 1, we ran ARS for each of the six MuJoCo locomotion tasks and the 100 random seeds. The results are shown in Figure 1. Such a thorough evaluation was feasible because ARS has a small computational footprint. As discussed in Appendix A.3, ARS is at least 15 times more computationally efficient on the MuJoCo benchmarks than competing methods.

Figure 1 shows that 70% of the time ARS trains policies for all the MuJoCo locomotion tasks, with the exception of Walker2d-v1 for which it succeeds only 20% of the time. Moreover, ARS succeeds at training policies a large fraction of the time while using a competitive number of episodes.
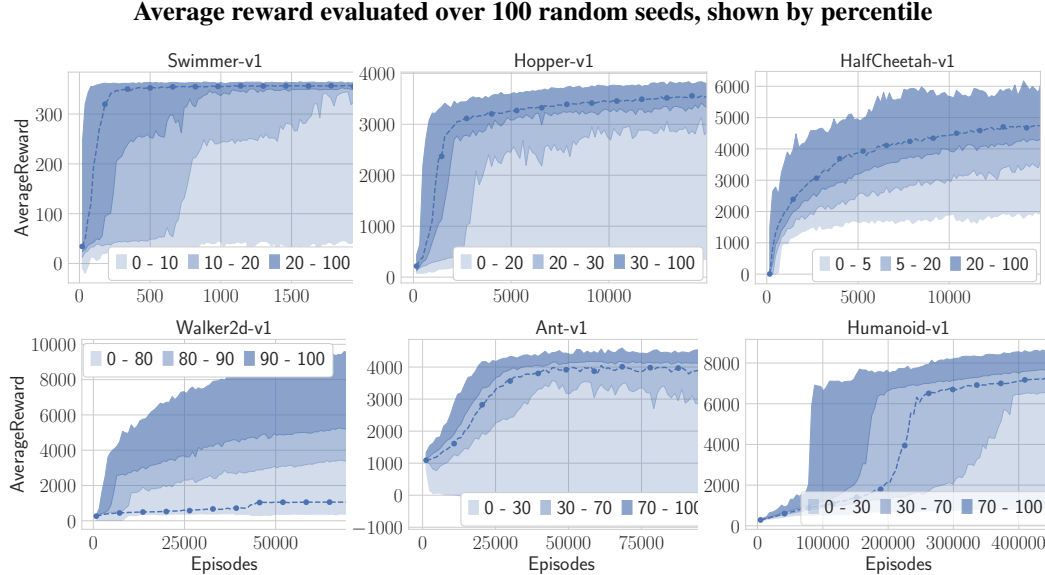
Average reward evaluated over 100 random seeds, shown by percentile

Figure 1: An evaluation of ARS over 100 random seeds on the MuJoCo locomotion tasks. The dotted lines represent median rewards and the shaded regions represent percentiles. For Swimmer-v1 we used ARS **V1**. For Hopper-v1, Walker2d-v1, and Ant-v1 we used ARS **V2-t**. For HalfCheetah-v1 and Humanoid-v1 we used ARS **V2**.

There are two types of random seeds represented in Figure 1 that cause ARS to not reach high rewards. There are random seeds on which ARS eventually finds high reward policies when sufficiently many iterations of ARS are performed, and there are random seeds which lead ARS to discover locally optimal behaviors. For the Humanoid model, ARS found numerous distinct gaits, including ones during which the Humanoid hops only on one leg, walks backwards, or moves in a swirling motion. Such gaits were found by ARS on the random seeds which cause slower training. While multiple gaits for Humanoid models have been previously observed [10], our evaluation better emphasizes their prevalence. The presence of local optima is inherent to non-convex optimization, and our results show that RL algorithms should be evaluated on many random seeds for determining the frequency with which local optima are found. Finally, we remark that ARS is the least sensitive to the choice of random seed used when applied to HalfCheetah-v1, a task which is often used for the evaluation of sensitivity of algorithms to the choice of random seeds.

Linear policies are sufficiently expressive for MuJoCo: We discussed how linear policies can produce diverse gaits for the MuJoCo models, showing that they are sufficiently expressive to capture diverse behaviors. Table 2 shows that linear policies can also achieve high rewards on all the MuJoCo locomotion tasks. In particular, for Humanoid-v1 and Walker2d-v1, ARS found policies that achieve significantly higher rewards than any other results we encountered in the literature. These results show that linear policies are perfectly adequate for the MuJoCo locomotion tasks, reducing the need for more expressive and more computationally expensive policies.

7

Maximum reward achieved

Task	ARS	Task	ARS	Task	ARS
Swimmer-v1	365	HalfCheetah-v1	6722	Ant	5146
Hopper-v1	3909	Walker	11389	Humanoid	11600

Table 2: Maximum average reward achieved by ARS, where we took the maximum over all sets of hyperparameters considered and the three fixed random seeds.

5   Discussion

With a few algorithmic augmentations, basic random search of static, linear policies achieves state-of-the-art sample efficiency on the MuJoCo locomotion tasks. Surprisingly, no special nonlinear controllers are needed to match the performance recorded in the RL literature. Moreover, since our algorithm and policies are simple, we were able to perform extensive sensitivity analysis. This analysis brings us to an uncomfortable conclusion that the current evaluation methods adopted in the deep RL community are insufficient to evaluate whether proposed methods are actually solving the studied problems.

The choice of benchmark tasks and the small number of random seeds do not represent the only issues of current evaluation methodology. Though many RL researchers are concerned about minimizing sample complexity, *it does not make sense to optimize the running time of an algorithm on a single problem instance.* The running time of an algorithm is only a meaningful notion if either (a) evaluated on a family of problem instances, or (b) when clearly restricting the class of algorithms.

Common RL practice, however, does not follow either (a) or (b). Instead, researchers run an algorithm $\mathcal{A}$ on a task $\mathcal{T}$ with a given hyperparameter configuration, and plot a "learning curve" showing the algorithm reaches a target reward after collecting X samples. Then the "sample complexity" of the method is reported as the number of samples required to reach a target reward threshold, with the given hyperparameter configuration. However, any number of hyperparameter configurations can be tried. Any number of algorithmic enhancements can be added or discarded and then tested in simulation. For a fair measurement of sample complexity, should we not count the number of rollouts used for all tested hyperparameters?

Through optimal hyperparameter tuning one can artificially improve the perceived sample efficiency of a method. Indeed, this is what we see in our work. By adding a third algorithmic enhancement to basic random search (i.e., enhancing ARS **V2** to **V2-t**), we are able to improve the sample efficiency of an already highly performing method. Considering that most of the prior work in RL uses algorithms with far more tunable parameters and neural nets whose architectures themselves are hyperparameters, the significance of the reported sample complexities for those methods is not clear. This issue is important because a meaningful sample complexity of an algorithm should inform us on the number of samples required to solve a new, previously unseen task.

In light of these issues and of our empirical results, we make several suggestions for future work:

- Simple baselines should be established before moving forward to more complex benchmarks and methods. We propose the Linear Quadratic Regulator as a reasonable testbed for RL algorithms. LQR is well-understood when the model is known, problem instances can be easily generated with a variety of different levels of difficulty, and little overhead is required for replication; see Appendix A.4 for more details.

- When games and physics simulators are used for evaluation, separate problem instances should be used for tuning and evaluating RL methods. Moreover, large numbers of random seeds should be used for statistically significant evaluations.

- Rather than trying to develop general purpose algorithms, it might be better to focus on specific problems of interest and find targeted solutions.

- More emphasis should be put on the development of model-based methods. For many problems, such methods have been observed to require fewer samples than model-free methods. Moreover, the physics of the systems should inform the parametric classes of models used for different problems. Model-based methods incur many computational challenges themselves, and it is quite possible that tools from deep RL, such as improved

tree search, can provide new paths forward for tasks that require the navigation of complex and uncertain environments.

Acknowledgments

References

[1] A. Agarwal, O. Dekel, and L. Xiao. Optimal algorithms for online convex optimization with multi-point bandit feedback. pages 28–40, 2010.

[2] F. Bach and V. Perchet. Highly-smooth zero-th order online optimization. *Conference on Learning Theory*, 2016.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI gym, 2016.

[4] S. Dean, H. Mania, N. Matni, B. Recht, and S. Tu. On the sample complexity of the linear quadratic regulator. *arxiv:1710.01688*, 2017.

[5] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control. *Proceedings of the International Conference on Machine Learning*, pages 1329–1338, 2016.

[6] A. D. Flaxman, A. T. Kalai, and H. B. McMahan. Online convex optimization in the bandit setting: gradient descent without a gradient. *Proceedings of the ACM-SIAM symposium on Discrete algorithms*, pages 385–394, 2005.

[7] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine. Q-prop: Sample-efficient policy gradient with an off-policy critic. *International Conference on Learning Representations*, 2016.

[8] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement learning with deep energy-based policies. *Proceedings of the International Conference on Machine Learning*, 2017.

[9] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arxiv:1801.01290*, 2018.

[10] N. Heess, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, A. Eslami, M. Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arxiv:1707.02286*, 2017.

[11] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. *arXiv:1709.06560*, 2017.

[12] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arxiv:1708.04133*, 2017.

[13] K. G. Jamieson, R. Nowak, and B. Recht. Query complexity of derivative-free optimization. pages 2672–2680, 2012.

[14] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *International Conference on Learning Representations*, 2016.

[15] J. Matyas. Random optimization. *Automation and Remote control*, 26(2):246–253, 1965.

[16] 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, 2015.

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

[18] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging ai applications. *arxiv:1712.05889*, 2017.

[19] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. *arxiv:1708.02596*, 2017.

[20] Y. Nesterov and V. Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017.

[21] 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. *arxiv:1706.01905*, 2017.

[22] A. Rajeswaran, K. Lowrey, E. Todorov, and S. Kakade. Towards generalization and simplicity in continuous control. *Advances in Neural Information Processing Systems*, 2017.

[23] T. Salimans, J. Ho, X. Chen, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arxiv:1703.03864*, 2017.

[24] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. pages 1889–1897, 2015.

[25] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *International Conference on Learning Representations*, 2015.

[26] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arxiv:1707.06347*, 2017.

[27] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. *Proceedings of the International Conference on Machine Learning*, 2014.

[28] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[29] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[30] S. Tu and B. Recht. Least-squares temporal difference learning for the linear quadratic regulator. *arxiv:1712.08642*, 2017.

[31] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas. Sample efficient actor-critic with experience replay. *Interational Conference on Learning Representations*, 2016.

[32] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *Advances in Neural Information Processing Systems*, 2017.

A Appendix

A.1 Reward shaping

For the Hopper-v1, Walker2d-v1, Ant-v1, and Humanoid-v1 tasks the default reward functions include a survival bonus, which rewards RL agents with a constant reward at each timestep, as long as a termination condition (i.e., falling over) has not been reached. For example, the environment Humanoid-v1 awards a reward of 5 at each time step, as long as the Humanoid model does not fall. Hence, if the Humanoid model stands still for 1000 timesteps, it will receive a reward of 5000 minus a small penalty for the actions used to maintain a vertical position. Furthermore, if the Humanoid falls forward at the end of a rollout, it will receive a reward higher than 5000.

It is common practice to report the sample complexity of an RL method by showing the number of episodes required to reach a reward threshold [7, 22, 23]. For example, Gu et al. [7] chose a threshold of 2500, while Rajeswaran et al. [22] chose a threshold of 5280. However, given the survival bonus awarded to Humanoid-v1, we do not believe these reward thresholds are meaningful for locomotion. In Table 1 and Section A.3 we use a reward threshold of 6000 to evaluate the performance of ARS on the Humanoid-v1 task, the threshold also used by Salimans et al. [23].

The survival bonuses awarded by the OpenAI gym discourage the exploration of policies that cause falling early on, which is needed for the discovery of policies that achieve locomotion. These bonuses cause ARS to find policies which make the MuJoCo models stand still for a thousand timesteps; policies which are likely local optima. These bonuses were probably included in the reward functions to help the training of stochastic policies since such policies cause constant movement through stochastic actions. To resolve the local optima problem for training deterministic policies, we subtracted the survival bonus from the rewards outputted by the OpenAI gym during training. For the evaluation of trained policies we used the default reward functions.

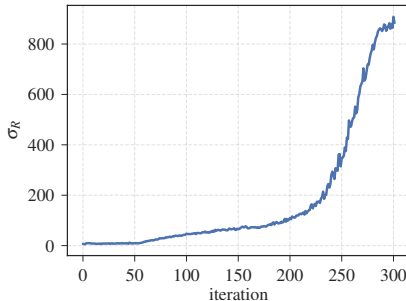## A.2 Supplementary empirical results

Figure 2: Showing the standard deviation σ_R of the rewards collected at each iteration, while training Humanoid-v1.

Comparisons to related methods

Table 3 shows the number of timesteps required by ARS to reach a prescribed reward threshold, averaged over the three fixed random seeds. The hyperparameters were chosen based on the same evaluations performed for Table 1 and Figure 3. We compare ARS to ES and TRPO. For these two methods we show the values reported by Salimans et al. [23], who used six random seeds for evaluation. Salimans et al. [23] do not report sample complexity results for the Ant-v1 and Humanoid-v1 tasks. Table 3 shows that TRPO requires fewer timesteps than ARS to reach the prescribed reward threshold on Walker2d-v1. However, we see that ARS requires fewer timesteps than ES and TRPO on the Swimmer-v1, Hopper-v1, and HalfCheetah-v1 tasks.

Table 4 shows the maximum reward achieved by ARS[5], PPO, A2C, CEM, and TRPO after one million timesteps of the simulator have been collected, averaged over the three fixed random seeds.

---

[5]We explain our methodology for computing this value for ARS in Appendix A.5.
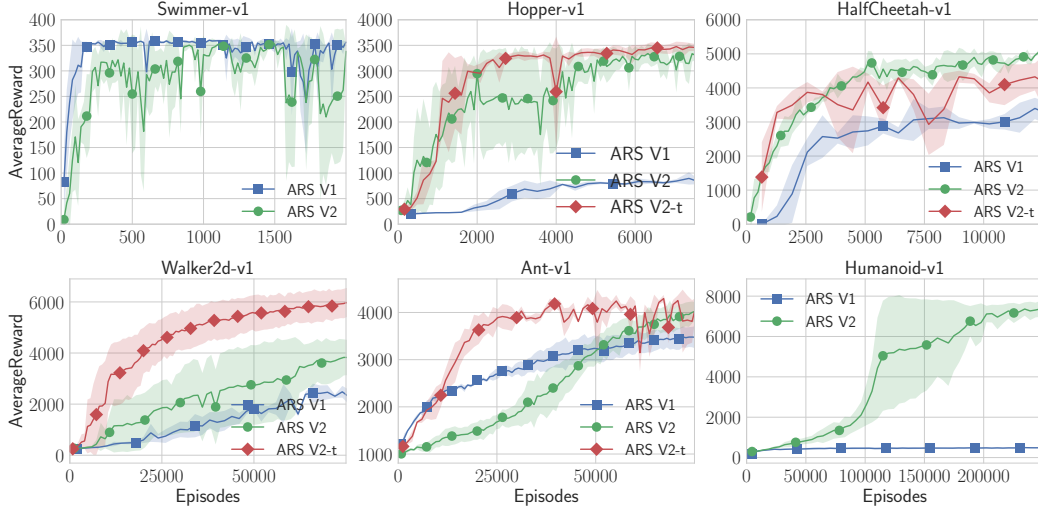
Figure 3: An evaluation of four versions of ARS on the MuJoCo locomotion tasks. The training curves are averaged over three random seeds, and the shaded region shows the standard deviation. ARS **V2-t** is only shown for the tasks to which it offered an improvement over ARS **V2**.

The hyperparameters were chosen based on the same evaluations performed for Table 1 and Figure 3. Schulman et al. [26] did not report performance of PPO, A2C, CEM, and TRPO on the Ant-v1 and Humanoid-v1 tasks of the OpenAI gym. Table 4 shows that ARS surpasses these four methods on the Swimmer-v1, Hopper-v1, and HalfCheetah-v1 tasks. On the Walker2d-v1 task PPO achieves a higher average maximum reward than ARS, while ARS achieves a similar maximum reward to A2C, CEM, and TRPO.

		Average # timesteps to hit Th.		
Task	**Threshold**	**ARS**	**ES**	**TRPO**
Swimmer-v1	128.25	$6.00 \cdot 10^4$	$1.39 \cdot 10^6$	$4.59 \cdot 10^6$
Hopper-v1	3403.46	$2.00 \cdot 10^6$	$3.16 \cdot 10^7$	$4.56 \cdot 10^6$
HalfCheetah-v1	2385.79	$5.86 \cdot 10^5$	$2.88 \cdot 10^6$	$5.00 \cdot 10^6$
Walker2d-v1	3830.03	$8.14 \cdot 10^6$	$3.79 \cdot 10^7$	$4.81 \cdot 10^6$

Table 3: A comparison of ARS and ES and TRPO methods on the MuJoCo locomotion tasks. For each task we show the average number of timesteps required by ARS to reach a prescribed reward threshold, averaged over three random seeds. For Swimmer-v1 we used ARS **V1**, while for the other tasks we used ARS **V2-t**. The values for ES and TRPO have been averaged over six random seeds and are taken from [23]. Salimans et al. [23] did not evaluate on Ant-v1 and they did not specify the exact number of timesteps required to train Humanoid-v1.

		Maximum average reward after # timesteps				
Task	**# timesteps**	**ARS**	**PPO**	**A2C**	**CEM**	**TRPO**
Swimmer-v1	10^6	361	≈ 110	≈ 30	≈ 0	≈ 120
Hopper-v1	10^6	3047	≈ 2300	≈ 900	≈ 500	≈ 2000
HalfCheetah-v1	10^6	2345	≈ 1900	≈ 1000	≈ -400	≈ 0
Walker2d-v1	10^6	894	≈ 3500	≈ 900	≈ 800	≈ 1000

Table 4: A comparison of ARS, PPO, A2C, CEM, and TRPO on the MuJoCo locomotion tasks. For each task we show the maximum rewards achieved after a prescribed number of simulator timesteps have been used, averaged over three random seeds. The values for PPO, A2C, CEM, and TRPO were approximated based on the figures presented by Schulman et al. [26].

Table 5 shows the maximum reward achieved by ARS, SAC, DDPG, SQL, and TRPO after a prescribed number of simulator timesteps have been collected. The hyperparameters for ARS were chosen based on the same evaluations performed for Table 1 and Figure 3. Table 5 shows that ARS

surpasses SAC, DDPG, SQL, and TRPO on the Hopper-v1 and Walker2d-v1 tasks, and that ARS is surpassed by SAC, DDPG, and SQL on the HalfCheetah-v1 taks. However, ARS performs better than TRPO on this task. On the Ant-v1 task, ARS is surpassed by SAC and performs similarly to SQL, but it outperforms DDPG and TRPO. We did not include values for Swimmer-v1 and Humanoid-v1 because Haarnoja et al. [9] did not use the OpenAI versions of these tasks for evaluation. Instead, they evaluated SAC on the rllab version [5] of these tasks. The authors indicated that Humanoid-v1 is more challenging for SAC than the rllab version because of the parametrization of the states used by the OpenAI gym, and that Swimmer-v1 is more challenging because of the reward function used.

Task	# timesteps	Maximum average reward after # timesteps				
		ARS	SAC	DDPG	SQL	TRPO
Hopper-v1	$2.00 \cdot 10^6$	3306	≈ 3000	≈ 1100	≈ 1500	≈ 1250
HalfCheetah-v1	$1.00 \cdot 10^7$	5024	≈ 11500	≈ 6500	≈ 8000	≈ 1800
Walker2d-v1	$5.00 \cdot 10^6$	4205	≈ 3500	≈ 1600	≈ 2100	≈ 800
Ant-v1	$1.00 \cdot 10^7$	2072	≈ 2500	≈ 200	≈ 2000	≈ 0

Table 5: A comparison of ARS, SAC, DDPG, SQL, and TRPO on the MuJoCo locomotion tasks. For each task we show the maximum rewards achieved after a prescribed number of simulator timesteps have been used. The values for ARS were averaged over three random seeds. The values for SAC, DDPG, SQL, and TRPO were approximated based on the figures presented by Haarnoja et al. [9], who evaluated these methods on five random seeds.

A.3 Computational efficiency

The small computational footprint of linear policies and the embarrassingly parallel structure of ARS make our method ideal for training policies in a small amount of time, with few computational resources. In Tables 6 and 7 we show the wall-clock time required by ARS to reach an average reward of 6000, evaluated over 100 random seeds. ARS requires a median time of 21 minutes to reach the prescribed reward threshold, when trained on one m5.24xlarge EC2 instance with 48 CPUs. The Evolution Strategies method of Salimans et al. [23] took a median time of 10 minutes when evaluated over 7 trials. However, the authors do not clarify what 7 trials means, multiple trials with the same random seed or multiple trials with different random seeds. Moreover, Table 7 shows that ARS trains a policy in at most 10 minutes on 10 out of 100 seeds. Also, Table 6 shows that ARS requires up to 15 times less CPU time than ES.

Finally, we would like to point out that our method could be scaled to more workers. In that case, ARS **V2** will have a computational bottleneck in the aggregation of the statistics Σ_j and μ_j across workers. For successful training of policies, ARS **V2** does not require the update of the statistics (see Line 8 of Algorithm 1) to occur at each iteration. For example, in their implementation of ES, Moritz et al. [18] allowed each worker to have its own independent estimate of μ_j and Σ_j. With this choice, the authors used Ray to scale ES to 8192 cores, reaching a 6000 reward on Humanoid-v1 in 3.7 minutes. One could tune an update schedule for the statistics Σ_j and μ_j in order to reduce the communication time between workers or reduce the sample complexity of ARS. For the sake of simplicity we refrained from tuning such a schedule.

Algorithm	Instance Type	# CPUs	Median Time	CPU Time
Evolution Strategies	UNK	18	657 minutes	197 hours
	UNK	1440	10 minutes	240 hours
ARS **V2**	m5.24xlarge	48	21 minutes	16 hours
	c5.9xlarge	18	41 minutes	12 hours
	c4.8xlarge	18	57 minutes	17 hours

Table 6: An evaluation of the wall-clock time required to reach an average reward of 6000 for the Humanoid-v1 task. The median time required by ARS was evaluated over 100 random seeds. The values for ES were taken from the work by Salimans et al. [23], and were evaluated over 7 independent trials. UNK stands for unknown.

			# minutes by percentile			
Algorithm	**Instance Type**	**# CPUs**	**10th**	**25th**	**50th**	**75th**
	m5.24xlarge	48	10	13	21	45
ARS **V2**	c5.9xlarge	18	16	23	41	96
	c4.8xlarge	18	21	28	57	144

Table 7: A breakdown by percentile of the number of minutes required by ARS to reach an average reward of 6000 on the Humanoid-v1 task. The percentiles were computed over runs on 100 random seeds.

Sensitivity to hyperparameters:

It has been correctly noted in the literature that RL methods should not be sensitive to hyperparameter choices if one hopes to apply them in practice [9]. For example, DDPG is known to be highly sensitive to hyperparameter choices, making it difficult to use in practice [5, 9, 11]. In the evaluations of ARS presented above we used hyperparameters chosen by tuning over the three fixed random seeds. To determine the sensitivity of ARS to the choice of hyperaramaters, in Figure 4 we plot the median performance of all the hyperparameters considered for tuning over the three fixed random seeds. Recall that the grids of hyperparameters used for the different MuJoCo tasks are shown in Appendix A.6.

Interestingly, the success rates of ARS depicted in Figure 4 are similar to those shown in Figure 1. Figure 4 shows a decrease in median performance only for Ant-v1 and Humanoid-v1. The similarity between Figures 1 and 4 shows that the success of ARS is as influenced by the choice of hyperparameters as it is by the choice of random seeds. To put it another way, ARS is not highly sensitive to the choice of hyperparameters because its success rate when varying hyperaramaters is similar to its success rate when performing independent trials with a "good" choice of hyperparameters. Finally, Figure 4 shows that the performance of ARS on the HalfCheetah-v1 task, a problem often used for evaluations of sensitivity [9, 11], is the least sensitive to the choice of hyperparameter.

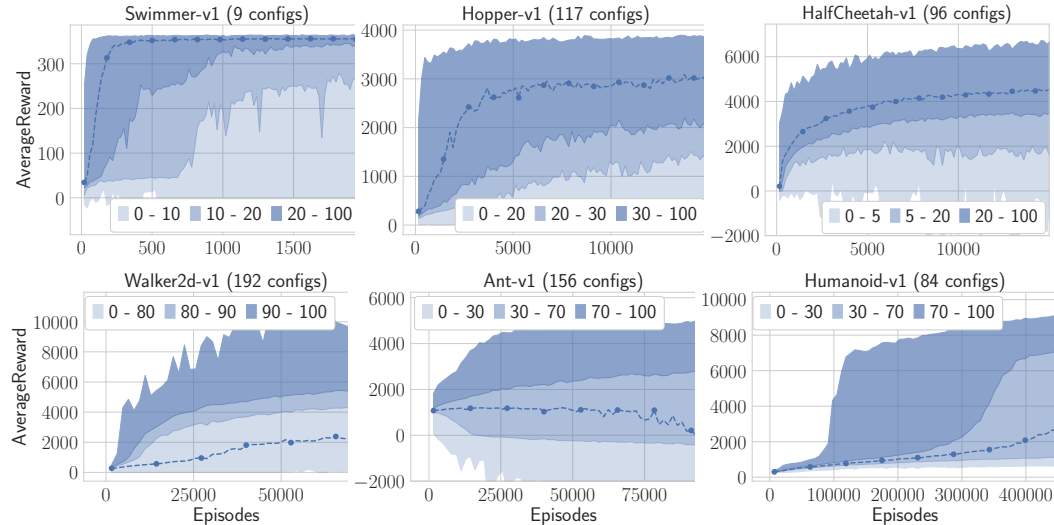**Evaluation of sensitivity to hyperparameters, shown by percentile**

Figure 4: An evaluation of the sensitivity of ARS to the choice of hyperparameters. The dotted lines represent median average reward and the shaded regions represent percentiles. We used all the learning curves collected during the hyperparameter tuning performed for the evaluation over the three fixed random seeds. For Swimmer-v1 we used ARS **V1**, and for the rest of the environments we used ARS **V2-t** (and implicitly **V2** when $b = N$).

A.4 Linear quadratic regulator

While the MuJoCo locomotion tasks considered above are popular benchmarks in the RL literature, they have their shortcomings. The maximal achievable awards are unknown as are the optimal policies, and the current state-of-the-art may indeed be very suboptimal. These methods exhibit high variance making it difficult to distinguish quality of learned policies. And, since it is hard to generate new instances, the community may be overfitting to this small suite of tests.

In this section we propose a simpler benchmark which obviates many of these shortcomings: the classical Linear Quadratic Regulator (LQR) with unknown dynamics. In control theory the LQR with known dynamics is a fundamental problem, which is thoroughly understood. In this problem the goal is to control a linear dynamical system while minimizing a quadratic cost. The problem is formalized in Eq. (3). The states x_t lie in $\mathbb{R}^n$, the actions u_t lie in $\mathbb{R}^p$, and the matrices A, B, Q, and R are have the appropriate dimensions. The noise process w_t is i.i.d. Gaussian. When the dynamics (A, B) are known, under mild conditions, problem (3) admits an optimal policy of the form $u_t = K x_t$ for some unique matrix K, computed efficiently from the solution of an algebraic Riccati equation. Moreover, the finite horizon version of problem 3 can be efficiently solved via dynamic programming.

$$\min_{u_0, u_1, \ldots} \lim_{T \to \infty} \frac{1}{T} \mathbb{E} \left[\sum_{t=0}^{T-1} x_t^\top Q x_t + u_t^\top R u_t \right] \tag{3}$$
$$\text{s.t. } x_{t+1} = A x_t + B u_t + w_t$$

LQR with unknown dynamics is considerably less well understood and offers a fertile ground for new research. Note that it is still trivial to produce a varied set of instances for LQR, and we can always compare the best achievable cost when the dynamics are known.

A natural model-based approach consists of estimating the transition matrices (A, B) from data and then solving for K by plugging the estimates in the Riccati equation. A controller K computed in this fashion is called a *nominal controller*. Though this method may not be ideally robust (see, e.g., Dean et al. [4]), nominal control provide a useful baseline to which we can compare other methods.

Consider the LQR instance introduced by Dean et al. [4] as a challenging low-dimensional instance for LQR with unknown dynamics.

$$A = \begin{bmatrix} 1.01 & 0.01 & 0 \\ 0.01 & 1.01 & 0.01 \\ 0 & 0.01 & 1.01 \end{bmatrix}, \ B = I, \ Q = 10^{-3} I, \ R = I . \tag{4}$$

The matrix A has eigenvalues greater than 1, and hence the system is unstable without some control. Moreover, if a method fails to recognize that the system is unstable, it may not yield a stable controller. In Figure 5 we compare ARS to nominal control and to a method using Q-functions fitted by temporal differencing (LSPI), analyzed by Tu and Recht [30].

While Figure 5b shows ARS to require significantly more samples than LSPI to find a stabilizing controller, we note that LSPI requires an initial controller K_0 which stabilizes a discounted version of problem (3). ARS does not require a special initialization. However, Figure 5b also shows that the nominal control method is orders of magnitude more sample efficient than both LSPI and ARS. Hence there is much room for improvement for pure model-free approaches.

We conjecture that the LQR instance (4) would also be particularly challenging for policy gradient methods or other methods that explore in the action space. When the control signal is zero, the linear system described by Eq. (4) has a small spectral radius ($\rho \approx 1.024$) and as a result the states x_t would blow up, but slowly. Therefore long trajectories are required for evaluating the performance of a controller. However, the variance of policy gradient methods grows with the length of the trajectories used, even when standard variance reduction techniques are used.

A.5 Maximum reward achieved after a prescribed number of timesteps

We explain our procedure for obtaining the maximum reward achieved by ARS after a prescribed number of timesteps, averaged over three random seeds. A natural method consists of finding the maximum reward achieved by ARS on each random seed and averaging those values. However, in
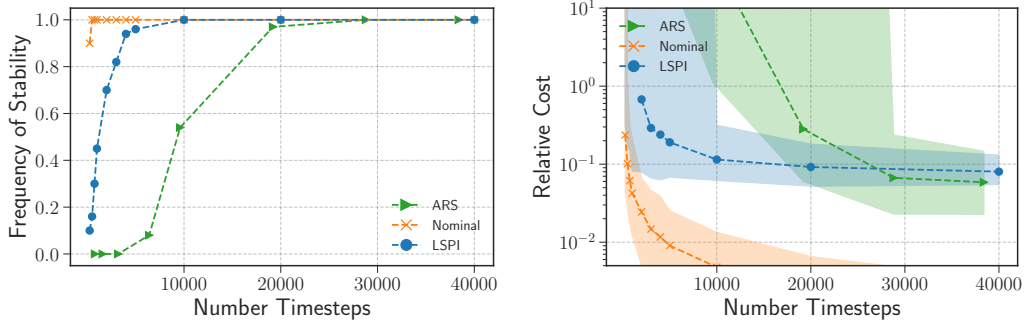
(a) A comparison of how frequently the controllers produced by ARS, the nominal synthesis procedure, and the LSPI method find stabilizing controllers. The frequencies are estimated from 100 trials. **(b)** A comparison of the relative cost of the controllers produced by ARS, the nominal synthesis procedure, and the LSPI method. The points along the dashed line denote the median cost, and the shaded region covers the 2-nd to 98-th percentile out of 100 trials.

Figure 5: A comparison of four methods when applied to the LQR problem (4).

Tables 4 and 5 we compare the performance of ARS to results taken from the figures presented by Schulman et al. [26] and Haarnoja et al. [9], who average training curves across random seeds. For a fair comparison we cannot compare the average of maxima with the maximum of an average of training curves. Therefore, we use the following method for estimating the average maximum reward achieved by ARS.

We begin by introducing some notation. Let $R_i^{(j)}$ be the reward achieved by ARS at iteration i, on the jth random seed. Also let $h_i^{(j)}$ be the total number of timesteps sampled by ARS up to iteration i, on the jth random seed. Then, we average the training curves of ARS across the three random seeds to obtain

$$\overline{R}_i = \frac{R_i^{(1)} + R_i^{(2)} + R_i^{(3)}}{3}.$$

If β is the prescribed budget of timesteps, let $\overline{h} = \min\{i \mid \max\{h_i^{(1)}, h_i^{(1)}, h_i^{(1)}\} \geq \beta\}$. Then, in Tables 4 and 5 we report

$$\overline{R}_{\max} = \max_{0 \leq i \leq \overline{h}} \overline{R}_i. \tag{5}$$

The estimate (5) is a conservative measure of the performance of ARS because $\overline{h}$ is the minimum over the random seeds of the number of iterations needed by ARS to deplete the available budget of timesteps.

A.6 Hyperparameters

	Swimmer-v1		Hopper-v1		HalfCheetah-v1
α :	0.01, 0.02, 0.025	α :	0.01, 0.02, 0.025	α :	0.01, 0.02, 0.025
ν :	0.03, 0.02, 0.01	ν :	0.03, 0.025, 0.02, 0.01	ν :	0.025, 0.02, 0.01
N :	1	N :	8, 16, 32	N :	4, 8, 16, 32
b :	1	b :	4, 8, 32	b :	2, 4, 8, 32
	Walker-v1		Ant-v1		Humanoid-v1
α :	0.01, 0.02, 0.025, 0.03	α :	0.01, 0.015, 0.02, 0.025	α :	0.01, 0.02, 0.025
ν :	0.025, 0.02, 0.01, 0.0075	ν :	0.025, 0.02, 0.01	ν :	0.01, 0.0075
N :	40, 60, 80, 100	N :	20, 40, 60, 80	N :	90, 230, 270, 310, 350
b :	15, 30, 100	b :	15, 20, 40, 80	b :	100, 200, 360

Table 8: Grids of hyperparameters used during hyperarameter tuning.

V2					V2-t				
Task	α	ν	N		**Task**	α	ν	N	b
Swimmer-v1	0.02	0.01	1		Swimmer-v1	0.02	0.01	1	1
Hopper-v1	0.02	0.02	4		Hopper-v1	0.01	0.025	8	4
HalfCheetah-v1	0.02	0.03	8		HalfCheetah-v1	0.02	0.03	32	4
Walker2d-v1	0.025	0.01	60		Walker2d-v1	0.03	0.025	40	30
Ant-v1	0.01	0.025	40		Ant-v1	0.015	0.025	60	20
Humanoid-v1	0.02	0.0075	230		Humanoid-v1	0.02	0.0075	230	230

Table 9: Hyperparameters for ARS **V2** and **V2-t** used for the results shown Figure 3.