Comparison of Simple Search algorithms and Deep Learning methods for Locomotion Tasks

Gitika Meher Karumuri
Department of Electrical and Computer Engg.
University of California San Diego
9500 Gilman Dr, La Jolla, CA 92093
`gkarumur@eng.ucsd.edu`

1 Abstract

Model-free reinforcement learning aims to offer off-the-shelf solutions for controlling dynamical systems without requiring models of the system dynamics. Efficient search algorithms which surpass the performance of the popular RL algorithms have been introduced in the previous few years. This work is a comparison of the Augmented random search algorithm with a popular Deep Learning (Deep Deterministic Policy Gradients) algorithm for solving Locomotion Tasks. Experiments are carried out in the continuous action space environments for different Locomotion tasks offered by Pybullet. Both the chosen algorithms are model-free and can be easily scaled to any environment. Experiments show that ARS achieves high rewards in fewer episodes for these tasks. It is also observed that the time taken for each episode and the computing resources for running the ARS algorithm is comparatively smaller.

2 Introduction

Although the current model-free RL algorithms are producing impressive solutions to various problems, the 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. The methods that are sample efficient i.e. methods that need little data, are increasingly complicated to develop. This increasing complexity has led to a reproducibility crisis. Also, many RL methods are not robust to changes in hyper-parameters, random seeds, or even different implementations of the same algorithm. In contrast, ARS is a simple model-free RL method, a derivative-free optimization algorithm for training static, linear policies. Unlike the RL policies which show high sensitivity toward both the choice of random seed and the choice of hyperparameters, ARS achieves a good result without a dependence on these factors.

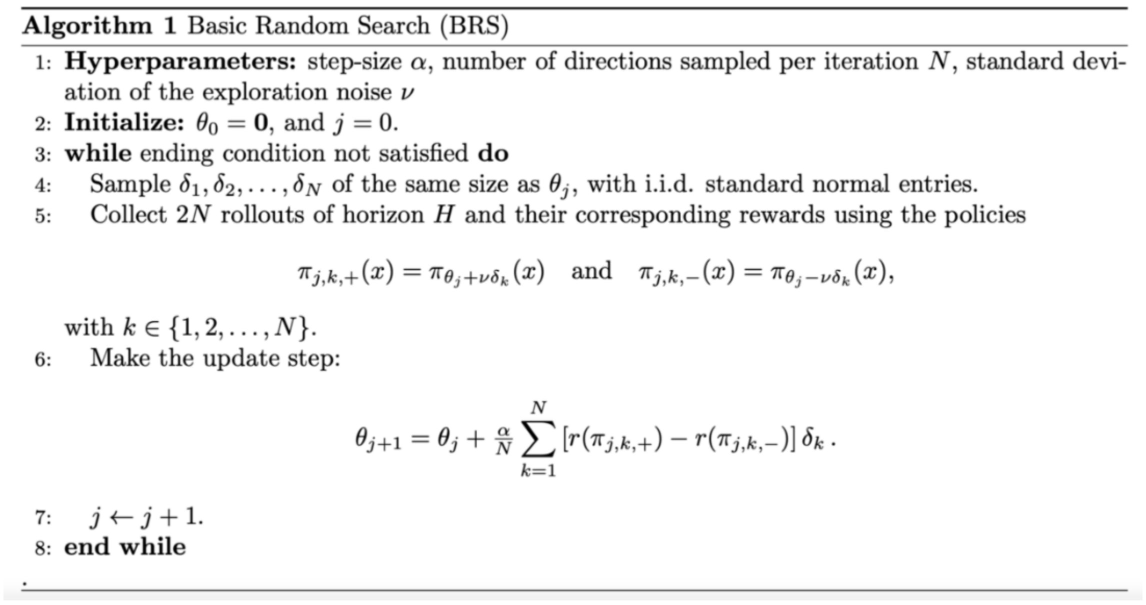
- Literature Review covers a few alternatives and approaches for solving the Locomotion Environments that have come up in the past few years.
- I introduce the family of Random Search algorithms and explain the Basic Random Search and Augmented Random Search algorithms with psuedo-code.
- I introduce the Deep learning algorithms for this problem setup and elaborate on the Actor Critic methods and the Deep Deterministic Policy Gradients algorithm.
- The Experiments section talks about the implementation details and the results.

3 Literature Review

There have been many Reinforcement learning approaches proposed over the last few years which attempted at solving the Continuous action space problems, especially for the locomotion tasks. Since the provision of infrastructure for the optimization-related applications like physical simulations for robotics and other physical systems, Mujoco or pybullet [1] simulations have been the baseline comparisons for all these algorithms.
Constitutional ways of approaching these problems have been using Deep RL architectures. There have been efforts to reduce the compute time and improve the efficiency by using Policy Gradients, Q-Network architectures, replay buffers, model free architectures etc. [2] explains a lot of existing deep RL methods used for different control problems. I looked into model free deep RL architectures with results comparable to the state of the art on the locomotion environments. One of them is the Deep Deterministic Policy Gradients [3] algorithm which was introduced in 2015. It is derived from the Actor-critic architecture with improvements like experience replay. In 2018, UC Berkely [4] proposed a random search algorithm for addressing this problem. Their proposal was simple, robust and efficient, inspired from a derivative-free policy optimization method, called Evolution Strategy[5] which would surpass the performance of the current state-of-the-art RL architecture. [6] is another similar effort that was made in 2017.

4 Random Search Algorithms

Random search is a family of numerical optimization methods that do not require the gradient of the problem to be optimized, and Random Search can hence be used on functions that are not continuous or differentiable. Such optimization methods are also direct-search, derivative-free, or otherwise called black-box methods. 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. Augmented Random Search (ARS) method relies on three augmentations of BRS that build on successful heuristics employed in deep reinforcement learning. Both BRS and ARS use the Method of Finite Differences in which to update the weights effectively, the agent takes a random matrix of tiny values and adds them to the weights. The AI then adds the exact same matrix, same values, but this time with negative weights. It then repeats this many times and the end result is an agent trying to perform a task with slightly different weights. It then adjusts those weights according to the weight configurations depending on the best rewards.



Algorithm 1 Basic Random Search (BRS)

```
1: Hyperparameters: step-size $\alpha$, number of directions sampled per iteration $N$, standard deviation of the exploration noise $\nu$
2: Initialize: $\theta_0 = \mathbf{0}$, and $j = 0$.
3: while ending condition not satisfied do
4:     Sample $\delta_1, \delta_2, \dots, \delta_N$ of the same size as $\theta_j$, with i.i.d. standard normal entries.
5:     Collect $2N$ rollouts of horizon $H$ and their corresponding rewards using the policies
           $\pi_{j,k,+}(x) = \pi_{\theta_j + \nu\delta_k}(x)$ and $\pi_{j,k,-}(x) = \pi_{\theta_j - \nu\delta_k}(x)$,
       with $k \in \{1, 2, \dots, N\}$.
6:     Make the update step:
           $\theta_{j+1} = \theta_j + \frac{\alpha}{N} \sum_{k=1}^{N} [r(\pi_{j,k,+}) - r(\pi_{j,k,-})]\, \delta_k$ .
7:     $j \leftarrow j + 1$.
8: end while
```

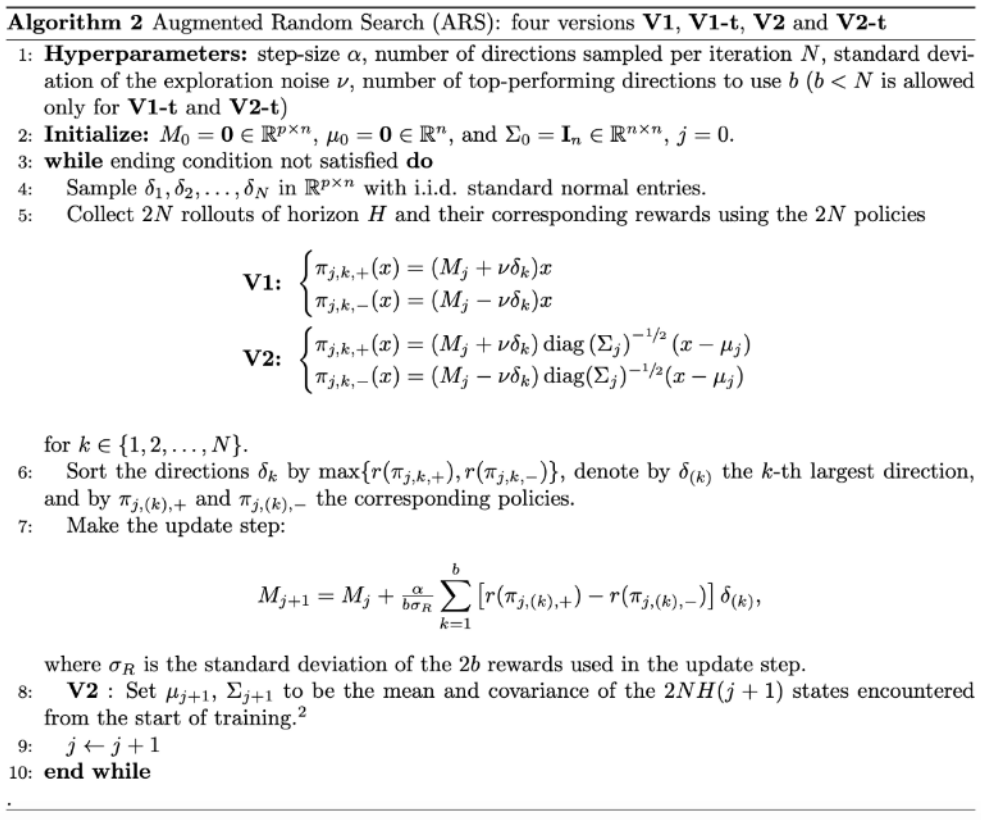
.

4.1 Basic Random Search

The idea of Basic Random Search is to pick a parameterized policy π_θ, perturb the parameters θ by applying $+\nu\delta$ and $-\nu\delta$ where ν< 1 is a constant noise and δ is a random number generated from a normal distribution. The actions based on $\pi(\theta + \nu\delta)$ and $\pi(\theta - \nu\delta)$ obtain the rewards r$(\theta + \nu\delta)$ and r$(\theta - \nu\delta)$ resulting from those actions respectively. The average Δ is calculated for all δ and the parameters θ are updated using a learning rate α. The pseudo-code for the algorithm is presented above.

4.2 Augmented Random Search

The ARS is an improved version of BRS, it contains a three additional enhancements that makes it better when compared to BRS. ARS uses the method of finite differences to adjust its weights and learn how to perform a task.



Algorithm 2 Augmented Random Search (ARS): four versions **V1**, **V1-t**, **V2** and **V2-t**

```
1: Hyperparameters: step-size $\alpha$, number of directions sampled per iteration $N$, standard deviation of the exploration noise $\nu$, 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 $\delta_1, \delta_2, \dots, \delta_N$ 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 $k \in \{1, 2, \dots, N\}$.
6:     Sort the directions $\delta_k$ 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 $\sigma_R$ is the standard deviation of the $2b$ rewards used in the update step.
8:     V2 : Set $\mu_{j+1}$, $\Sigma_{j+1}$ to be the mean and covariance of the $2NH(j+1)$ states encountered from the start of training.[2]
9:     $j \leftarrow j + 1$
10: end while
```

.

Dividing by the Standard Deviation (σ_r) :
As iterations go on, the difference between r$(\theta + \nu\delta)$ and r$(\theta - \nu\delta)$ can vary significantly, with the learning rate α fixed, the update $\theta^{j+1} = \theta^j + \alpha\Delta$ might oscillate considerably depending on the values of α and Δ. The range of variations possible is different for different instances and this hurts the updates. As the goal is to make θ converge towards values that maximize rewards, this is averted by dividing $\alpha\Delta$ by σ_r which is the standard deviation of the collected rewards.

Normalizing the States:
The normalization of states ensures that policies put equal weight on the different components of the states. For example, if a state component takes values in the range [90, 100] while another state component takes values in the range [1, 1] then, the first state component will dominate the computation, while the second won't have any effect in the case of BRS. The proposed normalization technique used consists of subtracting the current observed average of the state from the state input and dividing it by the standard deviation of the state:

$\frac{State\_Input - State\_Observed\_Average}{State\_Std}$

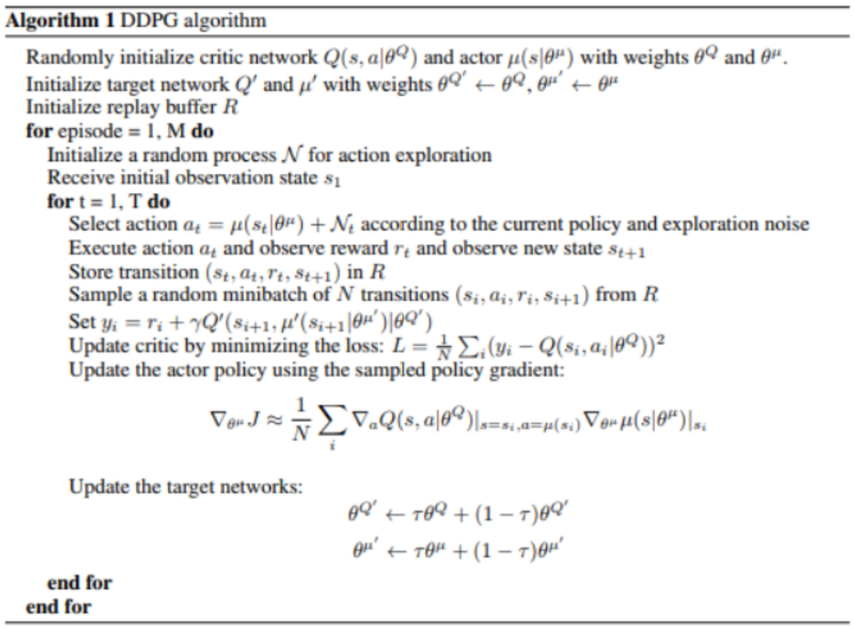
Using top performing directions:
In BRS, the process of collecting the average reward in each iteration, containing 2N episodes, each one following $\pi(\theta + \nu\delta)$ $\pi(\theta - \nu\delta)$ to collect the average rewards: r$(\theta + \nu\delta)$ and r$(\theta - \nu\delta)$ for all the 2N episodes has some pitfalls because if some of the rewards are small compared to the others, they will push the average down. The proposed way to remedy this issue is to sort the rewards in the decreasing order based on the key max(r$(\theta + \nu\delta)$, r$(\theta - \nu\delta)$) and to use only the top b rewards and their respective perturbation in the computation of the average reward. When b = N, the algorithm will be the same as the one without this enhancement. The pseudo-code for the algorithm is presented above..

5 Deep Learning Methods

There have been a wide number of approaches and algorithms to solve the continuous action space problems in Reinforcement Learning. The popular ones include Policy Gradient, Q-Learning approaches, TRPO, PPO, Guided searches etc. Actor Critic methods are hybrid methods that combine both value-based and policy-based networks. Given their prominence in the recent years and their precedence for successfully solving most of the Atari and Mujoco environments in the shortest time, I chose to compare the performance of an Actor Critic algorithm with that of search algorithms.

5.1 Introducing Actor Critic

Actor critic was introduced to overcome the short-comings of policy gradient updates. Policy Gradient methods wait until the end of episode to calculate the reward. Also, they may conclude that a high reward implies all equally good actions although in reality, that is hardly true. As a consequence, to have an optimal policy, we need a lot of samples. This produces slow learning and takes a lot of time to converge. The Actor Critic model is a better score function. Instead of waiting until the end of the episode, it makes an update at each step (TD Learning). The Critic model approximates the value function which calculates the maximum expected future reward given a state and an action. This value function replaces the reward function in policy gradient that calculates the rewards only at the end of the episode for the Actor module.



Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

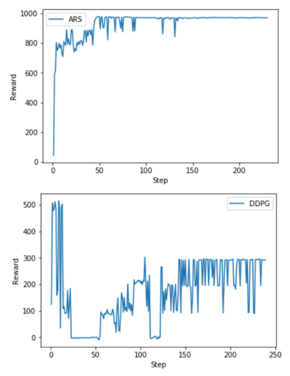
5.2 Deep Deterministic Policy Gradients

DDPG is a model-free policy based learning algorithm in which the agent will learn directly from the unprocessed observation spaces without knowing the domain dynamic information using the policy gradient methods which estimates the weights of an optimal policy through gradient ascent. That means the same algorithm can be applied across domains. DDPG employs Actor-Critic model in which the Actor brings the advantage of learning in continuous actions space without the need for extra layer of optimization while the Critic supplies the Actor with knowledge of the performance. It also employs the Replay Buffer which allows the DDPG agent to learn offline by gathering experiences collected from the environment agents and sampling experiences from large Replay Memory Buffer across a set of unrelated experiences.
DDPG uses four neural networks: a Q network, a deterministic policy network, a target Q network, and a target policy network. The target networks are time-delayed copies of their original networks that slowly track the learned networks to improve stability in learning. The pseudo code for the DDPG algorithm is presented above.

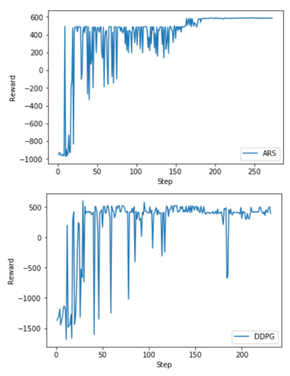
6 Experimental Setup and Results

Implementation Details: I used the compute services provided by Google Colaboratory for running different tasks using the two different algorithms. An instance of the same provides the user with a RAM of 12 GB and 4 core CPU power. I used the Pybullet [1] environment tasks using OpenAI gym [7] for this project as Mujoco issues paid licenses which are hardware-locked to 3 machines which would mena I cannot use services like Google Colab and Pybullet has almost all the same environments as of Mujoco. Both the implementations are in Python3. DDPG uses the pytorch libraries supported by python additionally. The OpenAI Gym provides benchmark reward functions for the different locomotion tasks which are used to calculate rewards. The environments used for comparison are 1. Inverted Pendulum 2. Half-Cheetah 3. Ant and 4. Humanoid
ARS: To avoid the computational bottleneck of communicating perturbations, a shared noise table which stores independent standard normal entries is used. Instead of communicating perturbations, the workers communicate indices in the shared noise table. A seed is randomly set with every initialization of the ARS class object. ARS V2 algorithm is the one used for all the experiments as [4] claims that this version is the one that works the best. No comparisons are carried out for different random search algorithms.
DDPG: The hyper-parameters and the random seeds used for running different tasks are unchanged and are set heuristically. Experimentation with the same has not been carried out as this would need more time and more resources.

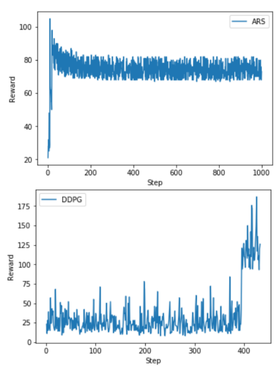
Results: The graphs below show the reward on the y-axis plotted against the episode step in the x-axis for both the ARS and the DDPG algorithms for different locomotion tasks.



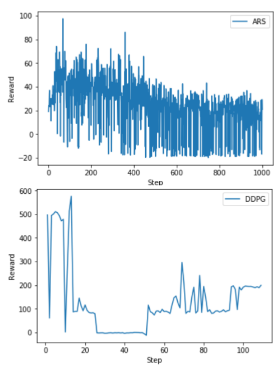
ANT- V0



HALF CHEETAH-V0



INVERTED PENDULUM- V0



HUMANOID - V0

We observe that the ARS algorithm achieves higher rewards in fewer episodes in all the tasks except for Humanoid-V0 in which the ARS algorithm struggles to converge even in a 1000 episodes.

7 Discussion

Ant-v0: We observe that ARS achieves rewards in an order of +1000 under 50 episodes while DDPG runs for around 250 episodes with a non-consistent but an average reward of around +300. As the number of episodes increase, we see that the rewards obtained by ARS are quite stably high. The rewards obtained by the DDPG algorithm oscillate a lot in this case. Another point to note here is that neither of the algorithms record a negative reward for any episode.
HalfCheetah-v0: Both the algorithms record a negative reward in the initial episodes and start to learn. For around 150 episodes, the reward obtained by ARS oscillates a lot. It stabilizes after this point and stays steadily at around +550. DDPG does achieve rewards around +500 the same time but takes a little longer to stabilize.
InvertedPendulum-v0: ARS algorithm starts with low positive rewards and quickly achieves high positive rewards of around + 100 in under 30 episodes and stabilizes quickly too around +80. DDPG oscillates from 0 reward to +65 for 400 episodes and picks up after that. It also touches rewards upto +175 but is not consistent and oscillates nevertheless.
Humanoid-v0: ARS oscillates continuously for this task between negative rewards and low positive rewards. I was not able to reproduce the results quoted by [4] probably because of the random seed. The humanoid figure was not able to stand up in the renderings produced by the ARS algorithm. [4] also claims that they have observed different gaits for this particular task depending on the seed. DDPG starts with high positive rewards of around +500 and falls down quickly. It later picks up and stabilizes around a reward of +300.

With a few algorithmic augmentations, basic random search of static, linear policies achieves state-of-the-art sample efficiency on most of the locomotion tasks tested on. It is also possible to perform an extensive sensitivity analysis when the algorithm used is simple. I used the same code in the case both the algorithms for different tasks. Since we saw that ARS doesn't perform the best for all the tasks, rather than trying to develop general purpose algorithms, it might be better to focus on specific problems of interest and find targeted solutions.

8 Conclusion

As discussed by [4], the control problems require a more thorough evaluation strategy than comparing the convergence of the reward in terms of number of episodes. As we saw, all the RL architectures are concerned about minimizing sample complexity and hence the time of convergence, it makes more sense to optimize the running time of an algorithm on a single problem instance or a class of algorithms. But since the solution is highly task dependent, hyper-parameter tuning cannot be eliminated. ARS does perform much better for some tasks compared to DDPG but also does fail miserably in the case of some others. This is not captured by the evaluation metric that is currently used. Also, it might be worth nothing that the algorithm complexity of ARS is very low and could perform well as compared to the model based algorithms as the latter incur many computational challenges.

9 Code

Github Link: https://github.com/Gitikameher/Augmented-Random-Search-and-DDPG-on-locomotion-tasks

References

[1] Pybullet. `https://pypi.org/project/pybullet/`, 2017. Accessed: 2019-03-09.

[2] Mario Srouji, Jian Zhang, and Ruslan Salakhutdinov. Structured control nets for deep reinforcement learning. 02 2018.

[3] 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. *CoRR*, abs/1509.02971, 2015.

[4] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search of static linear policies is competitive for reinforcement learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 1800–1809. Curran Associates, Inc., 2018.

[5] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *CoRR*, abs/1703.03864, 2017.

[6] Aravind Rajeswaran, Kendall Lowrey, Emanuel V. Todorov, and Sham M Kakade. Towards generalization and simplicity in continuous control. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6550–6561. Curran Associates, Inc., 2017.

[7] Gym. `https://gym.openai.com/`, 2016. Accessed: 2019-03-09.