A Scalable Finite Difference Method for Deep Reinforcement Learning

Anonymous authors
Paper under double-blind review

Abstract

Several low-bandwidth distributable black-box optimization algorithms have recently been shown to perform nearly as well as more refined modern methods in some Deep Reinforcement Learning domains. In this work we investigate a core problem with the use of distributed workers in such systems. Further, we investigate the dramatic difference in performance between the popular Adam gradient descent algorithm and the simplest form of stochastic gradient descent. These investigations produce a stable, low-bandwidth learning algorithm that achieves 100% usage of all connected CPUs under typical conditions.

1 Introduction

Reinforcement learning (RL) is a sub-field of machine learning that is concerned with finding an optimal policy π^* to direct an agent through a Markov decision process (MDP) so as to maximize an objective $J(\pi)$. Most model-free methods to accomplish this are derived from algorithms like *Q-learning* (Watkins, 1989), *policy gradients* (Sutton et al., 1999), and combinations thereof (Mnih et al., 2016; Schulman et al., 2017; Bhatnagar et al., 2009). In this work, we are interested in policy gradient methods where the policy is represented by a set of real parameters $\theta \in \mathbb{R}^d$. These methods attempt to find π^* by iteratively tuning θ through gradient ascent on $J(\pi_\theta)$.

Alternative methods for policy optimization such as black-box optimizers are less common in modern RL literature, though they can be effective (Salimans et al., 2017; Mania et al., 2018; Such et al., 2017). One such method is *evolution strategies* (Salimans et al., 2017) (ES). ES is a distributable learning algorithm that optimizes a population of policies neighboring π_θ by stochastically perturbing the policy parameters θ and maximizing the expected reward of these perturbations. Unlike other common methods, ES is not derived from dynamic programming and does not take advantage of any temporal information about the decision process when optimizing this population. Nonetheless, ES has been shown to be competitive with powerful learning algorithms like TRPO (Schulman et al., 2015) and A2C (Mnih et al., 2016). ES is particularly valuable in cases where bandwidth between computers is limited because ES workers only need to communicate two values to a central server per sequence of interactions with a decision process.

Regardless of method, a common factor in many RL algorithms is the Adam gradient descent algorithm (Kingma & Ba, 2014). Use of Adam is widespread and alternative gradient following rules are often not considered, despite the non-stationary nature of the data used to train policies in online RL algorithms. In this work we introduce two modifications to the simplest gradient following rule that bring it in line with Adam without using any statistical properties of the gradient. Further, we address a core problem with the computational efficiency of ES by incorporating data from perturbations of older policies in future gradient estimations.

2 Background

In this work we consider the undiscounted-return finite-horizon online reinforcement learning context (see Sutton & Barto (2018)).

2.1 Finite Difference Algorithms

In reinforcement learning, finite difference (FD) algorithms adjust the parameters of a policy π_θ to maximize the objective $J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]$ where $R(\tau)$ is called the *reward function*. To simplify our notation, we will hereafter refer to $J(\pi_\theta)$ as $J(\theta)$.

In this work we are only concerned with finite trajectories, e.g. finite sequences of state, action and reward triples

$$\tau = \{(s_0, a_0, r_0), (s_1, a_1, r_1)...(s_T, a_T, r_T)\}, \tag{1}$$

created by the interaction between an agent and an MDP starting from s_0 and continuing until a terminal state is reached. To maximize $J(\theta)$, the gradient $\nabla J(\theta_u)$ can be used to iteratively tune the parameters θ_u where u is the number of updates that have been applied to θ by the learning process. The simplest update rule for tuning θ_u is

$$\theta_{u+1} = \theta_u + \eta\nabla J(\theta_u), \tag{2}$$

where η is a hyper-parameter called the *learning rate*. This update rule is known as stochastic gradient descent, although in the RL setting it is typically used to maximize an objective rather than minimize one. FD methods estimate the gradient $\nabla J(\theta_u)$ by perturbing θ_u in some way to form new parameters α

$$\alpha = \theta_u + \delta. \tag{3}$$

A trajectory τ_α is then collected with the resulting policy and its cumulative reward is computed

$$R(\tau_\alpha) = \sum_{i=0}^{T} r_i. \tag{4}$$

Because only a single trajectory is typically used to evaluate the perturbed parameters α we will refer to $R(\tau_\alpha)$ as $R(\alpha)$ for simplicity. The change in reward for a perturbation is then

$$\Delta R = \frac{R(\alpha) - R_{\text{ref}}}{||\delta||}, \tag{5}$$

where $R_{\text{ref}} = R(\theta_u)$ in the forward difference case or $R_{\text{ref}} = R(\theta_u - \delta)$ in the central difference case (Peters & Schaal, 2008). A gradient estimate $\mathbf{g}_{\text{FD}}$ can then be accumulated over N perturbations as

$$\mathbf{g}_{\text{FD}} = \frac{1}{N}\sum_{i=1}^{N} \Delta R_i \frac{\delta_i}{||\delta_i||}, \tag{6}$$

where N is a hyper-parameter called the *batch size*.

2.2 Evolution Strategies

ES (Salimans et al., 2017) is a method for optimizing a distribution of policies that estimates a quantity similar to $\mathbf{g}_{\text{FD}}$ by stochastically perturbing the policy parameters θ with multi-variate Gaussian noise

$$\alpha = \theta_u + \sigma\epsilon, \tag{7}$$

where $\epsilon \sim \mathcal{N}(0, I) \in \mathbb{R}^{\dim(\theta)}$ and $0 < \sigma < \infty$. The ES gradient estimator is then

$$\mathbf{g}_{\text{ES}} = \frac{1}{\sigma}\mathbb{E}[R(\alpha)\epsilon] \tag{8}$$

$$\approx \frac{1}{\sigma N}\sum_{i=1}^{N} R(\alpha_i)\epsilon_i. \tag{9}$$

Notice that, while similar to $\mathbf{g}_{\text{FD}}$, $\mathbf{g}_{\text{ES}}$ does not center the reward of the perturbed parameters ($R_{\text{ref}} = 0$) or scale its gradient estimate by the size of each perturbation as in (6). However, recent work (Raisbeck et al., 2020) has shown that $\mathbf{g}_{\text{ES}} \approx c\,\mathbf{g}_{\text{FD}}$ for some scalar $c \in \mathbb{R}^+$.

ES can be made into a highly scalable distributed algorithm by collecting perturbations ϵ and their associated rewards $R(\alpha)$ on independent asynchronous CPUs. This enables a learner CPU to collect $(\epsilon, R(\alpha))$ pairs (referred to as a single *return*) from each worker CPU and compute $\mathbf{g}_{\text{ES}}$ to update θ_u as soon as N returns have arrived. This method of distribution is desirable because it is possible to compress each return into 2 values, which means ES requires very low bandwidth when compared with other distributed RL algorithms like R2D2 (Kapturowski et al., 2019), SEED RL (Espeholt et al., 2019), IMPALA (Espeholt et al., 2018) and others.

3 Learning Algorithm

A core issue in the implementation of ES is that trajectories in a decision process do not necessarily all take the same amount of time to collect. This means that some workers may take more time than others to return information to the learner, and this asynchronicity could lead to information loss if the learner has computed a new policy by the time a worker finishes testing a perturbation. To address this problem, Salimans et al. (2017) dynamically limited the number of time-steps an agent is allowed to interact with the decision process for before a trajectory is artificially cut off. While this solution mitigates the waiting problem for trajectories that may take a long time to collect, it inevitably introduces a bias to the information used to estimate the reward gradient; the only trajectories with complete information are those that do not get cut off early, which favors shorter trajectories. Further, this approach only guarantees 50% usage of connected workers in the worst case (Salimans et al., 2017).

3.1 Using Delayed Returns

We will now introduce an approach that enables workers to continually compute and test parameters α without terminating episodes early or the learner discarding any data computed prior to the current update. To do this, we will incorporate returns computed from perturbations of prior policy parameters θ_{u-n} when estimating $\mathbf{g}_{\text{FD}}$ where $n \leq u$ is some number of update steps that have been taken by the learning algorithm. This is possible if we treat perturbed parameters α from prior updates θ_{u-n} as perturbations of the current update θ_u which have also been biased by the sum of updates to θ that have been computed over the prior n update steps by the learning algorithm.

We begin with a forward difference estimator of the policy gradient where $\delta = \sigma\epsilon$ as in ES,

$$\mathbf{g}_{\text{FD}} = \frac{1}{N}\sum_{i=1}^{N}[R(\alpha_i) - R(\theta_u)]\frac{\sigma\epsilon_i}{||\sigma\epsilon_i||^2}. \tag{10}$$

Then, to allow returns from θ_{u-n} to contribute to $\mathbf{g}_{\text{FD}}$, we treat a reward sampled from a perturbed old policy $R(\theta_{u-n} + \sigma\epsilon)$ as a reward sampled from the current policy whose perturbation ϵ has been biased.

$$R(\theta_{u-n} + \sigma\epsilon) = R(\theta_u + (\sigma\epsilon - \nu)), \tag{11}$$

where the bias ν is the difference between θ_u and θ_{u-n},

$$\nu = \theta_u - \theta_{u-n}, \tag{12}$$

this allows us to treat all perturbations equally

$$\begin{aligned} \alpha &= \theta_u + \sigma\epsilon - \nu && (13)\\ &= \theta_u + \theta_{u-n} + \sigma\epsilon - \theta_u && (14)\\ &= \theta_{u-n} + \sigma\epsilon. && (15) \end{aligned}$$

Note that for $n = 0$ this reduces to the perturbations used by ES in Equation 7. Next we modify Equation 10 to allow for returns from any θ_{u-n} by replacing the Gaussian noise ϵ with the biased Gaussian noise λ where

$$\begin{aligned} \lambda &= \sigma\epsilon - \nu && (16)\\ &= \sigma\epsilon + \theta_{u-n} - \theta_u, && (17) \end{aligned}$$

which yields our method to approximate $\nabla J(\theta_u)$

$$\mathbf{g}_{\text{DFD}} = \frac{1}{N}\sum_{i=1}^{N}[R(\alpha_i) - R(\theta_u)]\frac{\lambda_i}{||\lambda_i||^2}. \tag{18}$$

We will call this method the *delayed finite difference* (DFD) gradient estimator.

3.2 DFD Implementation

We now provide two algorithms which implement the central learner and asynchronous workers for DFD. A return from our workers will contain a perturbation, its reward, the length of the trajectory used to compute this reward, and the update to θ that was perturbed. This means a single return from our worker is $(R(\alpha), \epsilon, T, u)$. To avoid outlier rewards causing undesirable behavior in the learning process, we standardize each batch of returned rewards by the mean μ_{R} and standard deviation σ_{R} of rewards in that batch.

Algorithm 1 DFD Learner

Input: η, σ, N, T_{lim}
Initialize: θ_0, T_{total}, u
while $T_{\text{total}} < T_{\text{lim}}$ **do**
  Transmit θ_u, u to workers
  Compute $R(\theta_u)$
  Collect N returns from workers
  Compute batch statistics μ_{R}, σ_{R}
  for $i = 0...N$ **do**
    $T_{\text{total}} \leftarrow T_{\text{total}} + T_i$
    $\lambda_i = \sigma\epsilon_i + \theta_{u-n_i} - \theta_u$
    $R(\alpha_i) \leftarrow \frac{R(\alpha_i)-\mu_{\text{R}}}{\sigma_{\text{R}}}$
  end for
  Compute $\mathbf{g}_{\text{DFD}}$ via (18)
  Compute θ_{u+1} via some update rule
  $u \leftarrow u + 1$
end while

Algorithm 2 DFD Worker

Input: σ
while running **do**
  Receive θ_u, u from learner
  Sample $\epsilon \sim \mathcal{N}(0, I)$
  Compute $\alpha = \theta_u + \sigma\epsilon$
  Collect $R(\alpha)$, T from decision process
  Transmit $R(\alpha)$, ϵ, T, u to learner
end while

Collecting $R(\theta_u)$ on the learner may be costly in practice. To circumvent this one might modify the worker to occasionally collect $R(\theta_u)$ instead of $R(\alpha)$, although this approach may require the learner to pause in settings where collecting $R(\theta_u)$ takes longer than it would take to collect N returns. Alternatively one might approximate $R(\theta_u)$ on the learner as the average of rewards $R(\alpha)$ instead of measuring $R(\theta_u)$ directly.

It is worth mentioning that, although all returns can be incorporated in $\mathbf{g}_{\text{DFD}}$, there may be extreme examples where one or more returns are far enough behind the current update we wish to discard them anyway (for example, if a worker loses connection to the learner for an extended period). To safeguard against such events we discarded returns that were more than U_{max} updates behind the current update, where U_{max} is a real-valued hyper-parameter greater than zero.

4 Following the Gradient

One of the most common update rules used by modern RL algorithms is the Adam update rule (Kingma & Ba, 2014). Adaptive methods like Adam are widely adopted in supervised learning where the statistical properties of a fixed set of training data are stationary and Adam's variance reduction technique is often beneficial. Adam's benefit to online reinforcement learning algorithms is less clear, as the distribution from which training data is generated is non-stationary. Recent evidence (Agarwal et al., 2020) suggests that the efficacy of Adam and similar optimizers may be significantly influenced by the magnitude of their updates ($||\theta_{u+1} - \theta_u||$) rather than the direction they move the parameters at each update. Further, in some settings it has been demonstrated that with the wrong learning rate Adam can actually cause the policy to become worse over the course of training (Henderson et al., 2018). Despite these facts, Adam still appears to be significantly better than the SGD update rule from (2).

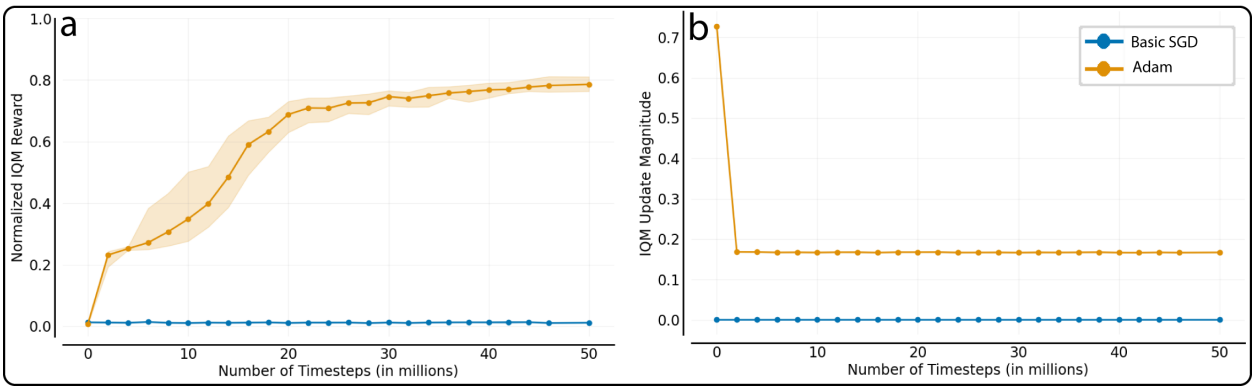

Figure 1: **Adam vs Basic SGD** An experiment demonstrating the dramatic difference between policies trained by Adam and SGD in the Hopper-v2 environment. Dotted lines represent the inter-quartile mean and shaded regions show pointwise 95% percentile stratified bootstrap confidence intervals over 10 random seeds. **a.** Reward curves for policies trained with Adam and the simplest form of SGD. Adam was always able to train effective policies in this environment and SGD was unable to train any effective policies. **b.** The magnitude of updates computed by Adam and SGD over training. The typical magnitude of updates computed by Adam was significantly larger than the typical magnitude of SGD updates. Note that although the plot appears to show a magnitude of zero for every update computed by SGD, the updates never had zero magnitude.

To demonstrate this, we conducted a simple experiment training a policy in the Hopper-v2 environment from the popular MuJoCo continuous control domain with fixed hyper-parameters using both Adam and Equation 2. Figure 1a clearly shows that Adam was able to train far better policies than the simplest form of SGD. However, notice that Figure 1b shows that the magnitude of updates computed by Adam far exceeded the magnitude of updates computed by SGD even though both update rules shared the same learning rate hyper-parameter η. We will now propose two approaches to improve SGD following this observation.

4.1 Modifying SGD

First, we modify the SGD update rule so the magnitude of its updates more closely match the magnitude of updates computed by Adam. To do this we need to know the range of magnitudes in which Adam's updates can lie. In our experiments we found that Adam's first update was always the largest, which may be because Adam's bias-correction term $\frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$ is at its largest when $t = 1$ for the typical hyper-parameter settings $\beta_1 = 0.9$ & $\beta_2 = 0.999$. In Appendix B, we show that magnitude of this update is always $||\theta_1 - \theta_0|| = \eta\sqrt{d}$ (recall that $\theta \in \mathbb{R}^d$). In our experiments the magnitude of Adam updates quickly approached a constant factor of the magnitude of the first update, which we measured to be approximately 0.23. We found that with Adam's update magnitude did not exceed this minimum once it was reached, and all updates following that point had the minimum magnitude. These observations led us to modify the basic SGD update rule in (2) such that the magnitude of each update is fixed at $0.23\eta\sqrt{d}$

$$\theta_{u+1} = \theta_u + 0.23\eta\sqrt{d}\frac{\nabla J(\theta_u)}{||\nabla J(\theta_u)||}. \tag{19}$$

We will refer to this method as *modified SGD* (MSGD).

Second, we introduce a heuristic method to dynamically adjust the magnitude of updates computed by SGD over the course of training. This will cause SGD to take larger steps when the policy is not improving fast enough according to the heuristic and smaller steps otherwise. This method maintains an average of the policy's reward over the prior M updates

$$R_{\text{avg}} = \frac{1}{M}\sum_{i=1}^{M} R(\theta_{u-i}), \tag{20}$$

and at each step compares the policy's current reward $R(\theta_u)$ with this average and increases the learning rate of SGD if $R(\theta_u)$ is not above R_{avg} by at least a factor of ρ

$$\eta_{u+1} = \begin{cases} \eta_u - \epsilon_1 & \text{if } R(\theta_u) > \rho R_{\text{avg}} \\ \eta_u + \epsilon_2 & \text{otherwise,} \end{cases} \tag{21}$$

where $0 \leq \epsilon_1 \leq 1;\ 0 \leq \epsilon_2 \leq 1;\ 1 < \rho$. The value of η_u is always clipped such that $0.23\eta_0 \leq \eta_u \leq \eta_0$ where η_0 is the typical learning rate hyper-parameter. This leads us to the parameter update rule

$$\theta_{u+1} = \theta_u + \eta_u \sqrt{d} \frac{\nabla J(\theta_u)}{||\nabla J(\theta_u)||}, \tag{22}$$

which we will call *dynamic SGD* (DSGD).

5 Experiments

To test our contributions, we studied each of our methods in combination and independently in 4 of the MuJoCo continuous control environments. All experiments shared fixed hyper-parameters and were tested across 10 random seeds. At every update, the current policy weights were frozen and the policy was used to collect 10 trajectories from the environment. The reward for the policy was computed as the average cumulative reward over those trajectories. Scores were normalized using min-max normalization relative to the best and worst performing policies in any experiment for each environment. All plots were generated using the RLiable library (Agarwal et al., 2021). Full experimental details can be found in Appendix A.

5.1 Update Rule

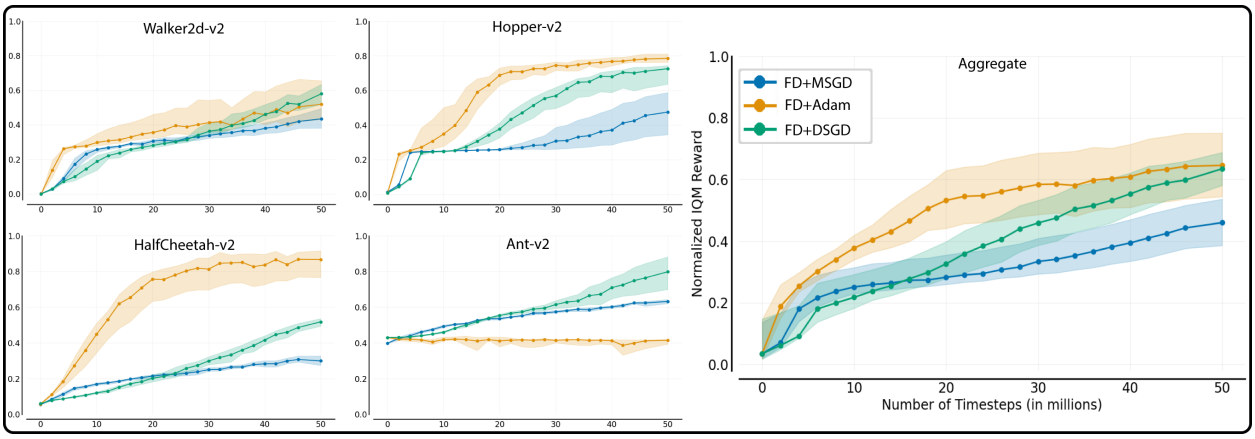

Figure 2: **The Effect of Modifying SGD** A comparison of Adam, MSGD, and DSGD on 4 MuJoCo environments. Dotted lines represent the inter-quartile mean and shaded regions show pointwise 95% percentile stratified bootstrap confidence intervals over 10 random seeds. MSGD did not match Adam in any environment, while DSGD was able to almost match Adam's performance in Walker2d-v2 and Hopper-v2. Adam was superior to both methods in HalfCheetah-v2 and unable to make any progress in Ant-v2.

We began by testing the impact of our modifications to SGD and compared them to Adam. To isolate the impact of each change to SGD we tested MSGD, DSGD, and Adam using our distributed system without incorporating delayed returns in any update.

We found that while MSGD was able to train effective policies in each environment, it was not able to match policies trained by Adam, as shown in Figure 2. However, the addition of the dynamic learning rate controller in DSGD was able to bring DSGD onto par with Adam in all but one environment. It is worth noting that although Adam was typically

faster to start gaining reward, the policies trained with Adam had a higher variance in reward across random seeds than policies trained with either MSGD or DSGD. This may indicate that Adam is more sensitive to initial conditions than either other update rule.

5.2 Delayed Returns

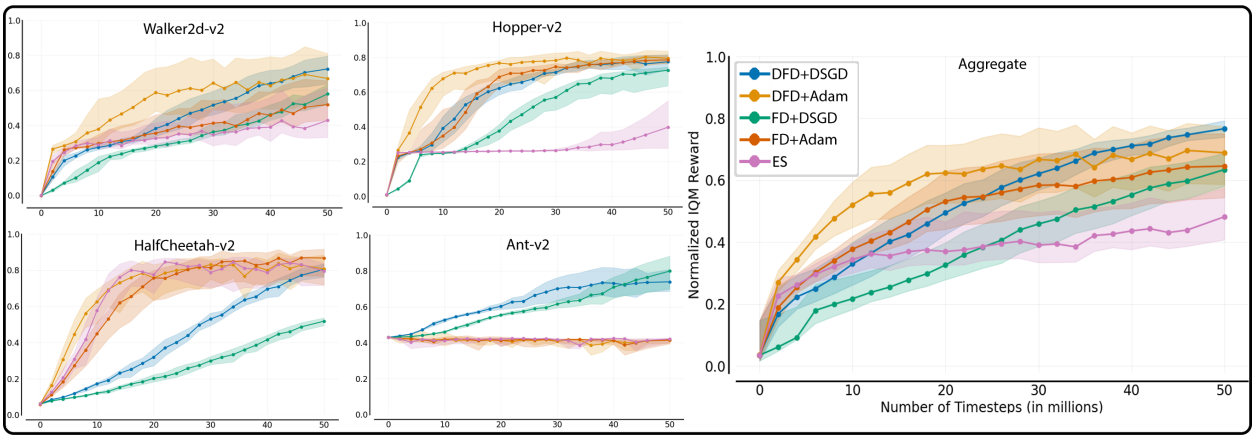

Figure 3: **Delayed Returns Ablation** A study to examine the impact of incorporating returns from delayed workers when updating the policy and a comparison of ES to our methods. Dotted lines represent the inter-quartile mean and shaded regions show pointwise 95% percentile stratified bootstrap confidence intervals over 10 random seeds. We found that using delayed returns was typically beneficial for both Adam and DSGD. The inclusion of delayed returns further reduced the gap between DSGD and Adam in HalfCheetah-v2, resulting in a similar final performance between the two update rules. ES was typically worse than FD with any combination of our contributions.

Next we tested the impact of incorporating delayed returns when computing updates to the policy with our system. To ensure the benefits of DSGD hold up under these new conditions, we tested DFD with both the DSGD update rule and Adam. We found that incorporating delayed returns was almost uniformly beneficial in the environments we studied, leading to high quality policies faster than otherwise in every setting we examined.

Table 1: Updates computed per optimizer — mean (standard deviation)

Environment	DSGD	DFD+DSGD
Ant-v2	1172.3 (12.6)	1485.4 (16.5)
HalfCheetah-v2	742.7 (2.5)	1250.0 (0)
Hopper-v2	1016.5 (15.9)	1541.5 (20.8)
Walker2d-v2	1180.6 (63.9)	1657.3 (68.9)

Table 2: Reward of best performing policies across random seeds — mean (standard deviation)

Method	Ant-v2	Walker2d-v2	Hopper-v2	HalfCheetah-v2	Sum of Ranks
DFD+Adam	211.1 (281.5)	2879.8 (823.0)	**3490.4** (803.7)	5501.2 (1556.0)	9
DFD+DSGD	2360.2 (833.1)	**2911.7** (749.0)	3388.6 (866.6)	5271.5 (1723.9)	10
FD+Adam	113.6 (253.2)	2355.6 (721.0)	3395.2 (911.3)	**5638.8** (1845.7)	11
FD+DSGD	**2538.2** (824.7)	2347.0 (615.4)	3001.9 (932.8)	3214.6 (1008.3)	14
FD+MSGD	1471.9 (408.6)	1748.5 (415.1)	2101.0 (532.3)	1664.4 (469.1)	20
ES	13.9 (195.9)	2065.7 (598.2)	1984.2 (324.2)	5429.3 (1690.7)	20

The benefit of incorporating returns from delayed workers may be due to the significant increase in the number of updates the optimizer can compute when no returns are discarded. To highlight the difference in updates computed between DSGD and DFD+DSGD in our experiments, we compare the mean and standard deviations of the number of updates computed by both methods across all environments in Table 1. Incorporating delayed returns resulted in a 31.1% mean increase in the number of updates computed by the optimizer across environments. This value depends on the rate at which returns are submitted to the learner and how quickly the learner can compute updates, which we discuss further in Appendix C.

Finally, we measured the performance of the best policy that a practitioner might expect to see when training a policy with any of the methods we tested. To do this, we measured the highest reward received by a policy on each random seed for each experiment. The mean and standard deviations for the collections of policy rewards across random seeds for each method can be found in Table 2. We believe this is useful because it is common for practitioners to run a learning algorithm for a fixed number of time-steps, then choose the best performing policy encountered by the learning algorithm over that time when training is finished. These measurements show the mean and standard deviation of the reward of policies chosen in this way across random seeds.

6 Discussion

Our experiments showed a clear advantage to incorporating each of our contributions in the final learning algorithm. We were able to close the gap between Adam and SGD by simply adjusting the magnitude of updates, and the resulting algorithm was robust to initial conditions. This suggests that Adam's benefit to online RL problems may be more due to the magnitude of its updates than its estimations of the first and second moments of the gradient. Incorporating returns from delayed worker CPUs was almost always beneficial in the settings we studied, resulting in better policies in nearly every environment.

6.1 Future Work

While we found incorporating delayed returns to be beneficial in these settings, their inclusion introduces a bias to the estimation of $\nabla J(\theta_u)$. The impact of this bias as the distance between the current policy parameters and those from a prior policy increases may be of interest to future research. Further, it may be possible to incorporate delayed returns an arbitrary number of times by a method analogous to the trust-regions established by TRPO (Schulman et al., 2015), or by incorporating PPO's (Schulman et al., 2017) clipping mechanism.

A topic of interest to future work may be investigating different methods of perturbing the policy. (Amari & Douglas, 1998) showed that the geometry of the space of policies is not Euclidean. One might consider methods to perturb the policy in an agent space (Raisbeck et al., 2021), or the natural space described by Amari & Douglas (1998) rather than the space of parameters.

Our modifications of SGD are not limited to usage in finite difference methods. An investigation of their effects on other RL algorithms and a comparison to update rules other than Adam may be topics of interest for future work. Finally, applications of our contributions to existing variations of SGD like the *momentum* method (Rumelhart et al., 1986) should be straightforward and may be beneficial.

7 Conclusions

We have introduced a scalable low-bandwidth method for black-box policy optimization using finite differences which does not prematurely terminate trajectories or stop workers from collecting data while the policy is being updated. Further, we proposed two modifications to the simplest gradient update rule that significantly narrow the gap between SGD and Adam.

Our method yields notable improvements over evolution strategies in continuous control tasks, and is consistent across initial conditions. We hope this work serves to move black-box optimization algorithms further into the light of modern approaches to deep reinforcement learning, and inspires future research into these methods.

References

Naman Agarwal, Rohan Anil, Elad Hazan, Tomer Koren, and Cyril Zhang. Disentangling adaptive gradient methods from learning rates, 2020. URL https://arxiv.org/abs/2002.11803.

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 2021.

S. Amari and S.C. Douglas. Why natural gradient? In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, volume 2, pp. 1213–1216 vol.2, 1998. doi: 10.1109/ICASSP.1998.675489.

Shalabh Bhatnagar, Richard S. Sutton, Mohammad Ghavamzadeh, and Mark Lee. Natural actor–critic algorithms. *Automatica*, 45(11):2471–2482, 2009. ISSN 0005-1098. doi: https://doi.org/10.1016/j.automatica.2009.07.008. URL https://www.sciencedirect.com/science/article/pii/S0005109809003549.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures, 2018. URL https://arxiv.org/abs/1802.01561.

Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference, 2019. URL https://arxiv.org/abs/1910.06591.

Peter Henderson, Joshua Romoff, and Joelle Pineau. Where did my optimum go?: An empirical analysis of gradient descent optimization in policy gradient methods, 2018. URL https://arxiv.org/abs/1810.02525.

Steven Kapturowski, Georg Ostrovski, Will Dabney, John Quan, and Remi Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=r1lyTjAqYX.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL https://arxiv.org/abs/1412.6980.

Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning, 2018. URL https://arxiv.org/abs/1803.07055.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016. doi: 10.48550/ARXIV.1602.01783. URL https://arxiv.org/abs/1602.01783.

Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21 (4):682–697, 2008. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2008.02.003. URL https://www.sciencedirect.com/science/article/pii/S0893608008000701. Robotics and Neuroscience.

John C. Raisbeck, Matthew Allen, Ralph Weissleder, Hyungsoon Im, and Hakho Lee. Evolution strategies converges to finite differences, 2020. URL https://arxiv.org/abs/2001.01684.

John C. Raisbeck, Matthew W. Allen, and Hakho Lee. Agent spaces, 2021. URL https://arxiv.org/abs/2111.06005.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, oct 1986. doi: 10.1038/323533a0.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017. URL https://arxiv.org/abs/1703.03864.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015. URL https://arxiv.org/abs/1502.05477.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning, 2017. URL https://arxiv.org/abs/1712.06567.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller (eds.), *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

Chris Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, England, 1989.

A Experimental Details

All experiments were run on a single c6a.8xlarge Amazon Web Services server instance. We conducted 3 at a time experiments in parallel on this machine where each experiment was given 4 vCPUs for workers and 4 vCPUs for the learner. All experiments were conducted with fixed hyper-parameters and each algorithm was tested over the same 10 random seeds. To compute the reward of the policy it was used to collect 10 trajectories after each update and the reward of the policy was measured as the average of the cumulative rewards of those trajectories.

A.1 Policy Parameterization

Policies in our experiments parameterized an independent Gaussian for each element in the action vector. Rather than using state-independent variance for each of these distributions, our policies produced both the mean and variance for each action distribution. This means that for an environment with A actions, the policy had $2A$ outputs. The means of each distribution were taken from the first half of a policy's output, and the variances were taken from the second half. The variance of each distribution was linearly transformed from the interval $[-1, 1]$ given by the $\tanh$ activation function onto the interval $[0, 1]$ before the distribution was constructed.

A.2 Algorithmic Details

All hyper-parameters used in our experiments are listed in Table 3. We chose these parameters because they are similar to the equivalent parameters from Mania et al. (2018) and Salimans et al. (2017).

Following the practice of ES, we maintained running statistics about the observations encountered by any policy during training and used them to standardize each observation by subtracting the running mean and dividing by the running standard deviation before providing an observation to a policy. After the standardization each element, the elements of the observation vector were clipped to be on the interval $[-5, 5]$.

As mentioned in the main text, we standardize rewards on the learner at each update using statistics from each batch such that every batch of rewards had zero mean and unit variance. Further, we approximated $R(\theta_u)$ when computing $\mathbf{g}_{\text{DFD}}$ by taking the mean of rewards from perturbations of the current policy in each batch. That is,

$$R(\theta_u) \approx \frac{1}{B} \sum_{i=1}^{B} R(\theta_u + \sigma\epsilon_i), \tag{23}$$

where B is the number of returns in a batch of size N that were from perturbations of the current policy (e.g. $\theta_u + \sigma\epsilon$).

When collecting returns from connected workers the learner continually accepted all available returns until there were at least N. If there were more than N returns available, the remaining returns were placed back on the buffer to be collected at the next iteration of the loop.

Table 3: Hyper-parameters used in our experiments

Parameter	Value
Total Timesteps	50,000,000
η_0	0.01
σ	0.02
N	40
ϵ_1	0.03080
ϵ_2	0.01026
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	1e-8
Adam η	0.01
ρ	1.035
U_{max}	3
Reward Standardization	Yes
Policy Architecture	$64\tanh \rightarrow 64\tanh \rightarrow \tanh$
Policy Outputs	Diagonal Gaussian
Standardized Observations	Yes
Observations Clipped	[-5, 5]
Random Seeds	124, 125, 126, ... 134
Worker CPUs	4

B Adam Analysis

Curiously, we found that the first update computed by Adam always had a magnitude equal to $\eta\sqrt{d}$, where d is the dimension of θ and η is Adam's learning rate hyper-parameter. We can show this to be true algebraically by examining the Adam update rule at its first iteration.

We begin by determining the value of Adam's three terms a_t, m_t, v_t at the first step, where $m_0 = 0$ and $v_0 = 0$. We will denote the i-th element of each vector term with the superscript i, e.g. m_u^i denotes the i-th element of m at the u-th update. Note that β_1 and β_2 are scalars, so β_1^u denotes exponentiating β_1 to power u.

$$
\begin{aligned}
a_u &= \frac{\eta\sqrt{1-\beta_2^u}}{1-\beta_1^u} & m_{u+1}^i &= m_u^i\beta_1 + (1-\beta_1)\frac{\partial J}{\partial \theta_u^i} & v_{u+1}^i &= v_u^i\beta_2 + (1-\beta_2)[\frac{\partial J}{\partial \theta_u^i}]^2 \\
a_1 &= \frac{\eta\sqrt{1-\beta_2}}{1-\beta_1} & m_1^i &= (1-\beta_1)\frac{\partial J}{\partial \theta_0^i} & v_{u+1}^i &= v_u^i\beta_2 + (1-\beta_2)[\frac{\partial J}{\partial \theta_u^i}]^2
\end{aligned}
$$

Now we plug in a_1, m_1^i, v_1^i to the Adam update rule

$$
\begin{aligned}
\theta_{u+1}^i &= \theta_u^i + \frac{a_t m_u^i}{\sqrt{v_u^i}} \\
\theta_1^i &= \theta_0^i + \frac{a_1 m_1^i}{\sqrt{v_1^i}}
\end{aligned}
$$

Distributing terms and simplifying gives us

$$\begin{aligned}
\theta_1^i &= \theta_0^i + \frac{\eta\sqrt{1-\beta_2}(1-\beta_1)\frac{\partial J}{\partial \theta_0^i}}{(1-\beta_1)\sqrt{(1-\beta_2)[\frac{\partial J}{\partial \theta_0^i}]^2}} \\
&= \theta_0^i + \frac{\eta\sqrt{1-\beta_2}(1-\beta_1)\frac{\partial J}{\partial \theta_0^i}}{\sqrt{1-\beta_2}(1-\beta_1)\sqrt{[\frac{\partial J}{\partial \theta_0^i}]^2}} \\
&= \theta_0^i + \eta\frac{\frac{\partial J}{\partial \theta_0^i}}{\sqrt{[\frac{\partial J}{\partial \theta_0^i}]^2}} \\
&= \theta_0^i + \eta\frac{\frac{\partial J}{\partial \theta_0^i}}{|\frac{\partial J}{\partial \theta_0^i}|}
\end{aligned}$$

This means that every element in the vector computed by Adam at the first update is $\pm\eta$, where the sign of each element is the same as the sign of the corresponding partial derivative. Therefore, the magnitude of this vector is $\eta\sqrt{d}$.

C The Dynamics of Delayed Returns

When we consider incorporating delayed returns into our finite-differences gradient approximation, the question arises: how does their introduction change the bias of our estimate of the gradient? We mentioned in section 3 a result from Salimans et al. (2017) which shows that there is a trade-off between the overall efficiency of a sequential finite-differences gradient approximator and bias in the estimate: the worst-case efficiency of a finite-differences gradient estimator may be arbitrarily bad if episodes of the process can take an arbitrarily long amount of time.

The solution in Salimans et al. (2017) is to accept the bias induced by terminating episodes early (i.e. after a certain number of time-steps) and to discard information from episodes which finish after the gradient of reward at θ_t is calculated. Our delayed returns method removes this bias by using returns regardless of how old the policy from which they were sampled is, but introduces several new factors which could lead to bias. Among these, the most important are (1) the change in the magnitude $||\alpha - \theta||$, and (2) the change in the spacial arrangement of the returns (i.e. delayed returns from θ_{t-k} are drawn from a distribution with mean θ_{t-k}, not θ_t).

Let us consider (1). The most important factor which makes a perturbation of the parameters useful for finite difference gradient approximation is the ability of the perturbation to approximate the partial derivative in the direction of the perturbation to a reasonable level of accuracy $\varepsilon > 0$ for each set of perturbed parameters α

$$\mathbb{E}\left[\left|\frac{R(\alpha) - J(\theta)}{||\alpha - \theta||}\frac{\alpha - \theta}{||\alpha - \theta||} - \nabla J(\theta) \cdot \frac{\alpha - \theta}{||\alpha - \theta||}\right|\right] < \varepsilon. \tag{24}$$

If J is a differentiable function, then at each point θ there is a ball of some radius $\delta > 0$ in which this inequality holds. So long as α remains within that ball for the current θ_t, it gives an acceptable approximation of the partial derivative. Under appropriate conditions, this holds so long as we keep our updates small. However, in optimization we would generally prefer to take larger steps, rather than smaller ones, when doing so is beneficial to the objective. These dynamics are highly particular to the dynamics of both the problem and the optimizer so in this paper we have resigned ourselves to empirical comparison. In such comparisons, we find incorporating delayed returns to be decisively beneficial.

Let us now consider (2). The spacial arrangement of returns from distributions about the previous parameters introduces a bias which is clear in concept, especially if updates are large in comparison to perturbations. The triangle inequality guarantees that if parameter updates are smaller than perturbations, there will be *some* perturbations from the prior parameters in every direction around the new parameters, however these will be still be shifted in distribution, having in our case approximately Gaussian distribution about the prior parameters. Because finite difference approximators of this kind approximate the gradient in stochastic directions, this not only means that the gradient approximation will be more refined in approximately the direction of the prior policy, but also that this part of the approximation will have greater weight.

While this is in our estimation the greatest source of bias induced by the incorporation of delayed returns, there is reason to believe that this bias may actually benefit the process of optimization. As numerous update methods incorporating "momentum" testify (including Adam) (Kingma & Ba, 2014) (Rumelhart et al., 1986), continuing to update the policy in the direction of prior updates often improves reward over the course of optimization. In practice, we observe that the overall method demonstrates significant improvements over methods which do not incorporate delayed returns, however we have not investigated the relative contributions of this effect and the sheer effect of additional returns on the speed of optimization.

Additionally, it is valuable to consider the outline of how incorporating delayed returns affects the total number of returns which may be considered. Let m be the number of machines connected in the optimization network and recall that N is the number of returns required for a single update. A standard asynchronous finite difference algorithm operating on m machines will have $m - 1$ wasted returns in each update, and the ignored returns are biased towards policies with intrinsically longer trajectories—that is, the returns used for the gradient approximation are disproportionately drawn from regions of the parameter space which produce agents that have shorter trajectories. As the number of machines decreases relative to the number of trajectories required for an update, $\frac{m}{N} \to 0$, these concerns fade, however this is also the situation in which the run-time of the optimization system (proportional to $\frac{N}{m}$) increases.

Thus, it is not possible to reduce this kind of bias without increasing the run-time and decreasing efficiency. By using delayed returns, we are able to exchange this kind of bias (not using certain returns) for an arguably lesser bias: using those returns in future updates. This enables us to efficiently use m close to N without issue; in practice, we have found that even high levels of delayed returns relative to non-delayed returns have a limited detrimental effect on the quality of the gradient. Since using delayed returns enables the algorithm to use the computation which would otherwise have been wasted by the $m - 1$ machines, we are able to perform a factor of approximately $\frac{m-1}{N}$ more updates per unit time *and* per interaction with the environment than Evolution Strategies and Finite Differences. Notice that this ratio increases with parallelization; the more machines are connected to the optimization network, the more beneficial delayed returns are.

In summary, it is not possible for a finite-differences gradient approximator to avoid biased approximation without severe effects on the worst-case run-time. Salimans et al. (2017) choose to bias the gradient by rejecting some slow returns in order to speed up the algorithm. In this paper, we suggest an alternative approach which introduces bias by changing *when* slow returns are incorporated, but not *if* they are incorporated. There is reason to believe that this results in significantly higher bias, especially because incorporating delayed returns increases the magnitude of the gradient in the direction of the past update. However the use of past gradient approximations is common in machine learning, where it is sometimes known as *momentum*. In practice, we find that the balance of these effects with the additional updates which are enabled by the use of delayed returns results in higher overall performance.