2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

Proximal Deterministic Policy Gradient

Marco Maggipinto [1], Gian Antonio Susto[1], and Pratik Chaudhari[2]

Abstract— **This paper introduces two simple techniques to improve off-policy Reinforcement Learning (RL) algorithms. First, we formulate off-policy RL as a stochastic proximal point iteration. The target network plays the role of the variable of optimization and the value network computes the proximal operator. Second, we exploits the two value functions commonly employed in state-of-the-art off-policy algorithms to provide an improved action value estimate through bootstrapping with limited increase of computational resources. Further, we demonstrate significant performance improvement over state-of-the-art algorithms on standard continuous-control RL benchmarks.**

I. Introduction

Actor Critic (AC) [1] algorithms have become the de facto standard in continuous control Reinforcement Learning tasks allowing the employment of powerful function approximation methods such as Deep Neural Networks (DNNs) to directly learn the control policy. While very effective in simulated environments, poor sample efficiency have limited their deployment on real systems, where querying the environment for new samples is expensive.

High variance of the gradient estimate [2] is at the foundation of such inefficiency. Algorithms like TRPO [3] and PPO [4] operate in a sample regime that fail to provide good approximations of the true gradient [5] with considerable impact on performance, moreover, their on policy nature requires new data to be collected at each optimization step wasting all past transitions. Deterministic Policy Gradient (DPG) algorithms [6] improve upon these methods by employing deterministic policies and off-policy updates; the former limit the source of randomness to the sole environment with a consequent reduction in the number of samples required for gradient estimation and the latter allows for data reuse by storing past transitions in a replay buffer and employing them during the entire training procedure.

Another source of error in policy updates is a poor action value function estimate. Here multiple factors come into play. On the one hand, Overestimation Bias [7] causes Q-learning algorithms to exhibit a consistent overestimation of the action value, with potentially divergent errors; to mitigate such problem, Double Q-learning [8] employs two independent Q-functions trained with a mixed update, however, [9] showed that this approach is not suitable in an AC setting and proposes a similar solution where the Time Difference (TD) update is performed with the minimum of the two action value functions.
On the other hand, the bootstrapping nature of TD updates results in a regression problem that changes over time making the optimization procedure very tricky and even unstable (if combined with off-policy updates and function approximation in the infamous deadly triad [2]). Most AC algorithms employ target networks that are slowly updated during training to provide a stable regression target. This approach was initially introduced in Double Deep Q-Networks (DDQN) [8] and since then it has been adopted by Deep Deterministic Policy Gradient (DDPG) [10], Twin Delayed Deep Deterministic policy gradient (TD3) [9], Soft Actor Critic (SAC) [11]. Target networks play a fundamental role in the optimization procedure and algorithms are typically very sensible to the speed which the networks are updated at.

In a Deep RL setting, TD learning is performed by minimizing a surrogate loss function (typically the Mean Square Error) with Stochastic Gradient Descent (SGD) based algorithms, the most common choice is Adam [12] that has proven effective for training DNNs. In this work, we propose an alternative optimization procedure to tackle the sample efficiency problem that provides a principled interpretation of target networks and minimizes a single loss function combining both policy and value updates. Our procedure employs *Time Damped* Stochastic Proximal Gradient (SPG) [13] [14] iteration, widespread in convex optimization, combined with bootstrapped action value estimates and is able to provide improved performance compared to state-of-the-art algorithms on continuous control tasks. More in details, we endow TD3 with a proximal gradient optimization procedure and we exploit the two Q-networks already used in the original algorithm to limit overestimation bias also to provide a more accurate action value estimate via bootstrapping that allows better policy updates.

II. Related work

The first successful application of DNNs in RL dates back to [15] where Deep Q-learning was introduced to Play Atari games at human level capabilities. Target networks where first introduced in [8] where they proposed an improved version of Deep Q-learning for the same control task. Since then, such algorithm has been the reference for Q-function estimation with DNNs.

In continuous control tasks, Q-learning methods are not enough to learn a policy; in fact, finding the maximizing action would require the solution of a maximization problem every time the agent needs to act on the environment, with prohibitive computational costs. Here, AC algorithms comes

*This work was not supported by any organization

[1]Marco Maggipinto and Gian Antonio Susto are with the Department of Information Engineering, University of Padova, Padova, Italy marco.maggipinto@dei.unipd.id, gianantonio.susto@dei.unipd.it

[2]Pratik Chaudhari is with the Electrical and Systems Engineering Department, University of Pennsylvania, Philadelphia, PA, USA pratikac@seas.upenn.edu

978-1-7281-6211-9/20/$31.00 ©2020 IEEE

into play where a parametrized policy learns to maximize the total expected reward. These methods typically follows the policy iteration [2] paradigm where at each time step the Q-function is estimated and then the policy is made greedy w.r.t. it. Hence Deep Q-learning is still a fundamental part of methods such as DDPG [10], A3C [16], TD3 [9], SAC [11] that are all related to our method. Alternative approaches that follows the AC paradigm but employ sample estimates of the Q-function can be found in [17], TRPO [3], PPO [4] and P3O [18].

There has been attempts to improve AC algorithms from an optimization point of view, by proposing alternatives to the TD error with more complex loss functions; SBEED [19] provides a primal dual interpretation of the Bellman Equation that results in a minmax game to optimize the convex dual of the quadratic loss function. [20] proposes a kernel loss alternative to the MSE that enables improved training of the Q-function.

Proximal methods, while widespread in convex optimization, have been used to train DNNs only in [21] in a Supervised Learning setting.

III. Background

RL deals with the problem of learning a maximally rewarding behavior for an agent interacting with its environment. Formally, this can be cast in the framework of optimal policy estimation in a Markov Decision Process (MDP) [22]. A MDP is a tuple $(S, A, p(\boldsymbol{s}'|\boldsymbol{s}), r(\boldsymbol{s}, \boldsymbol{a}), \gamma)$ where S is the set of states that the environment can assume, A the set of actions that can be performed on the environment, $p(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})$ the probability of transitioning to state $\boldsymbol{s}'$ after taking action $\boldsymbol{a}$ in state $\boldsymbol{s}$, $r(\boldsymbol{s}, \boldsymbol{a})$ the reward function, that can be either deterministic or stochastic, and γ the discount factor used to weight future rewards and guarantee finite total rewards even for infinite time horizon problems. The goal of an RL algorithm is finding a policy $\pi(\boldsymbol{a}|\boldsymbol{s})$ that maximizes the total expected discounted reward:

$$J = E_{\pi}\left[\sum_{t=1}^{T} \gamma^t \, r(\boldsymbol{s}_t, \boldsymbol{a}_t)\right] \tag{1}$$

Where the expectation is taken over all sources of randomness in the MDP.

Policy Gradient methods tackle the problem by directly maximizing (1) with respect to the parameters $\boldsymbol{\phi}$ of a NN parametrizing the policy function, using gradient based optimization procedures. We know from the policy gradient theorem [23] that it is possible to express the gradient of (1) with respect to the policy parameters as an expectation over trajectories:

$$\nabla J(\boldsymbol{\phi}) = \mathbb{E}_{\pi}\left[\sum_{t=1}^{T} Q^{\pi}(\boldsymbol{s}_t, \boldsymbol{a}_t) \log \pi_{\boldsymbol{\phi}}(\boldsymbol{a}_t|\boldsymbol{s}_t)\right] \tag{2}$$

Where the Q-function (or critic) $Q(\boldsymbol{s}, \boldsymbol{a})$ is the total expected reward obtained starting from state $\boldsymbol{s}$, performing action $\boldsymbol{a}$ and then following policy π. The Q-function can be estimated using TD learning, exploiting the Bellman equation:

$$Q^{\pi}(\boldsymbol{s}, \boldsymbol{a}) = r + \gamma \mathbb{E}_{\pi}\left[Q^{\pi}(\boldsymbol{s'}, \boldsymbol{a'})\right] \tag{3}$$

Here r is the expected reward after taking action $\boldsymbol{a}$ in state $\boldsymbol{s}$. To simplify the optimization procedure and speed-up policy updates, the expectation over trajectories in (1) is typically replaced with an expectation over transitions, hence maximizing the marginal expected reward. The resulting policy gradient is as follows:

$$\nabla J(\boldsymbol{\phi}) = \mathbb{E}_{\pi}\left[Q^{\pi}(\boldsymbol{s}, \boldsymbol{a}) \log \pi_{\boldsymbol{\phi}}(\boldsymbol{a}|\boldsymbol{s})\right] \tag{4}$$

In on policy methods such as PPO the actions are sampled from the current policy while off-policy methods employs a replay buffer $\mathcal{B}$ where past transitions are stored.

The policy gradient theorem stated in (2) is valid for stochastic policies; there is an analogous for the deterministic case [6] where the policy gradient can simply be obtained by backpropagation through the Q-function; then gradient ascent steps are taken in order to make the policy greedy with respect to the actual estimate of the action values. When NNs are used as function approximators for the Q-functions, the Ballman update in (3) requires itself the minimization of a loss function called TD error with respect to the network parameters $\boldsymbol{\theta}$. The resulting algorithm solves the coupled optimization problem:

$$\begin{aligned}
\boldsymbol{\theta}^* =& \underset{\boldsymbol{\theta}}{\text{argmin}}\, E_{(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')\sim\mathcal{B}}\left[TD_{\boldsymbol{\theta}}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')\right] \\
\boldsymbol{\phi}^* =& \underset{\boldsymbol{\phi}}{\text{argmax}}\, E_{(\boldsymbol{s},\boldsymbol{a})\sim\mathcal{B}}\left[Q^{\pi}_{\boldsymbol{\theta}}(\boldsymbol{s}, \pi_{\boldsymbol{\phi}}(\boldsymbol{a}|\boldsymbol{s})\right]
\end{aligned} \tag{5}$$

With $TD_{\boldsymbol{\theta}}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') = ||Q_{\boldsymbol{\theta}}(\boldsymbol{s}, \boldsymbol{a}) - r - Q_{\boldsymbol{\theta}}(\boldsymbol{s}', \pi_{\boldsymbol{\phi}}(\boldsymbol{a}|\boldsymbol{s}'))||^2$. Typically, to make SGD steps more stable, target networks are used both for the policy and the action value with parameters $\boldsymbol{\phi}'$ and $\boldsymbol{\theta}'$ that are averaged exponentially over time: $\boldsymbol{\phi}' = \tau\boldsymbol{\phi} + (1-\tau)\boldsymbol{\phi}'$, $\boldsymbol{\theta}' = \tau\boldsymbol{\theta} + (1-\tau)\boldsymbol{\theta}'$ with $\tau << 1$. The TD error is then computed as $TD_{\boldsymbol{\theta}}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') = ||Q_{\boldsymbol{\theta}}(\boldsymbol{s}, \boldsymbol{a}) - r - Q_{\boldsymbol{\theta}'}(\boldsymbol{s}', \pi_{\boldsymbol{\phi}'}(\boldsymbol{a}|\boldsymbol{s}'))||^2$ with gradient propagation only on $\boldsymbol{\theta}$.

IV. Proximal gradient methods

SGD based optimization algorithms perform exceptionally well at training DNNs especially in a Supervised setting where the data distribution does not change during training. In RL, however, this assumption is not true since data are collected with different policies at different time instants and the TD error targets evolve during time; all these factors make the optimization procedure difficult. Proximal Methods have shown appealing convergence and stability properties in convex optimization and can be an alternative to standard gradient based algorithms. Given a function $f : \mathbb{R}^N \rightarrow \mathbb{R}$ and a constant $\lambda \in \mathbb{R}$ we define the proximal operator for a point $\boldsymbol{y} \in \mathbb{R}^N$ as:

$$prox_{\lambda f}(\boldsymbol{y}) = \underset{\boldsymbol{x}}{argmin} f(\boldsymbol{x}) + \frac{1}{2\lambda}||\boldsymbol{x} - \boldsymbol{y}||^2 \tag{6}$$

For a convex function f the following theorem holds:

Theorem 1. *A vector $\boldsymbol{x}^* \in \mathbb{R}^N$ is a critical point for the function f iff $\boldsymbol{x}^* = prox_{\lambda f}(\boldsymbol{x}*)$*

For a detailed proof we refer the reader to [13]. This implies that, starting from a random point $\boldsymbol{x}_0$, by repeated applications of the proximal operator one can hope to reach the minimum of f; for this to happen, $prox_{\lambda f}(\boldsymbol{x})$ must be a contraction. While this is not true, the map is a firm non-expansion i.e. for every $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N$:

$$||\Delta||^2 \leq (\boldsymbol{x} - \boldsymbol{y})^T \Delta \tag{7}$$

With $\Delta = prox_{\lambda f}(\boldsymbol{x}) - prox_{\lambda f}(\boldsymbol{y})$.
Defining the damped proximal iteration with constant τ as:

$$\boldsymbol{x}_{k+1} = \tau \boldsymbol{x}_k + (1-\tau) prox_{\lambda f}(\boldsymbol{x_k}) \tag{8}$$

This iteration provably converges to a stationary point hence, in the convex setting, proximal iteration is an effective optimization method with convergence guarantees.

In the stochastic non-convex setting, the statements above do not hold true but the algorithm has still some desirable properties that make it a valid alternative to SGD. More in details, being f_k the loss function associated to the batch sampled at time k, the Stochastic Proximal Iteration (SPI), starting from a random point $\boldsymbol{x}_0$ is:

$$\boldsymbol{x}_{k+1} = prox_{\lambda f_k}(\boldsymbol{x}_k) \tag{9}$$

SPI can be interpreted as SGD performed on a smoothed loss function derived from the viscosity solution of the Hamilton Jacobi equation [21] $u(\boldsymbol{x}, t)$ defined as:

$$u(\boldsymbol{x}, t) = \underset{\boldsymbol{y}}{min} f(\boldsymbol{y}) + \frac{1}{2t}||\boldsymbol{x} - \boldsymbol{y}||^2 \tag{10}$$

In fact, if $prox_{tf}(\boldsymbol{x})$ exists, it is true that:

$$\nabla u(\boldsymbol{x}, t) = \frac{\boldsymbol{x} - prox_{tf}(\boldsymbol{x})}{t} \tag{11}$$

Hence, performing SGD updates on $u(\boldsymbol{x}, t)$ results in a *damped* SPI on the function f:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k - \nabla u_k(\boldsymbol{x}, t) \\ &= \boldsymbol{x}_k - \frac{\boldsymbol{x_k} - prox_{tf_k}(\boldsymbol{x_k})}{t} \\ &= (1 - \frac{1}{t})\boldsymbol{x}_k + \frac{1}{t} prox_{tf_k}(\boldsymbol{x_k}) \end{aligned} \tag{12}$$

Here t plays the role of λ in (6) and also serves as exponential averaging constant of the *damped* SPI. In particular, for $t = 1$ we recover SPI with $\lambda = t = 1$. As shown in [21] the function $u(\boldsymbol{x}, t)$ has smoother local minima and it's easier to optimize. For this reason, SPI has better stability and convergence properties than SGD.

V. Proposed Method

In this section we provide a detailed description of our algorithm and its optimization procedure. The proposed method is based on TD3 that has proven effective at solving continuous control tasks and provide state-of-the-art performance while being simple and easily reproducible. We provide here an interpretation of target networks that play a fundamental role in the optimization procedure of AC algorithms and are a widespread trick to make training more stable. Such interpretation can be easily derived from a few simple changes to (12); given variables $\boldsymbol{x}$ and $\boldsymbol{x'}$, we rewrite (12) as:

$$\begin{cases} \boldsymbol{x}_{k+1} = prox_{\lambda f_k}(\boldsymbol{x}'_k) \\ \boldsymbol{x}'_{k+1} = \tau \boldsymbol{x}_{k+1} + (1-\tau)\boldsymbol{x}'_k \end{cases} \tag{13}$$

Where we have introduced two hyper-parameters, λ and τ that control respectively the proximal term strength and the damping constant. This decouples the two terms as opposed to (12) where the single parameter t controls both, giving more freedom to tune the algorithm behavior. It is immediately clear how the time evolution of $\boldsymbol{x}$ in (13) is analogous to the parameters evolution during training of the ”fast” moving function. Similarly $\boldsymbol{x'}$ plays the role of the target parameters that slowly change during time. The hyper-parameter λ allows to control how close the two remains, which may help trading-off the update speed and how off-policy the data collected are. As in TD3 we employ target functions for the policy and the two action value networks with parameters denoted respectively as $\boldsymbol{\phi}'$, $\boldsymbol{\theta}'_1$ and $\boldsymbol{\theta}'_2$. The pair of Q-functions is used to reduce the overestimation bias and also to provide a more reliable estimate of the action value by bootstrapping; to train the two Q-networks we minimize the following TD error:

$$TD_{\boldsymbol{\theta}_i}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s'}) = huber(Q^{\pi}_{\boldsymbol{\theta}_i}(\boldsymbol{s}, \boldsymbol{a}) - y) \tag{14}$$

with $y = r + \underset{i \in 1,2}{min} Q_{\boldsymbol{\theta}'_i}(s', \pi_{\boldsymbol{\phi}'}(\boldsymbol{s}))$.
We employ the smooth-$L1$ loss (or huber) instead of the MSE. This choice is justified by the nature of the Bellman equation (3): the expected value over the next state and action is estimated in the TD error with a single transition and thus present a high variance; the huber loss put less weight on large errors compared to the MSE trusting less the expectation estimate. Moreover, since the targets change during training, the smooth-$L1$ loss may improve stability reducing strong changes in the parameters during a single optimization step. The policy network is trained to maximize the average action value of the two target Q-functions; the corresponding loss for a single transition $\ell_{\boldsymbol{\phi}}(\boldsymbol{s})$ is:

$$\ell_{\boldsymbol{\phi}}(\boldsymbol{s}) = -0.5\left(Q^{\pi}_{\boldsymbol{\theta}'_1}(\boldsymbol{s}, \pi_{\boldsymbol{\phi}}(\boldsymbol{s})) + Q^{\pi}_{\boldsymbol{\theta}'_2}(\boldsymbol{s},\ \pi_{\boldsymbol{\phi}}(\boldsymbol{s}))\right) \tag{15}$$

There are two main differences in the loss functions compared to standard TD3:

1) Target networks are used to compute the policy gradients instead of their ”fast” counterpart.
2) A bootstrapped estimate of the action value leverages both the the available Q-networks to reduce the approximation error.

The first difference implies that our method does not require delayed policy updates because the target networks change slowly during time. Moreover, the improved quality of the action value gives better gradient estimates for the policy. The resulting methods thus performs SPI on a single loss function $\ell(\boldsymbol{\theta}, \boldsymbol{\phi})$:

$$\begin{aligned} \ell(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\phi}) = E_{(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')\sim\mathcal{B}} \, [& TD_{\boldsymbol{\theta}_1}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}') \\ & +TD_{\boldsymbol{\theta}_2}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}') \\ & +\beta \, \ell_{\boldsymbol{\phi}}(\boldsymbol{s}) \;] \end{aligned} \tag{16}$$

Where we introduced the hyper-parameter β to control the scale of the two loss functions. We believe this expedient is important since the TD error has the same scale of the one step reward while the Q-function that is maximized by the policy has magnitude similar to the cumulative reward. For most Mujoco environments the difference is of almost three orders of magnitude.

As in [9] we add clipped noise to the actions in order to smooth the action value functions. In particular, for each action $\bar{\boldsymbol{a}}$ in the batch B_k we sample noise from a normal distribution $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \sigma_q \boldsymbol{I})$ and then set:

$$\bar{\boldsymbol{a}} = \bar{\boldsymbol{a}} + clip(\boldsymbol{\epsilon}, -c, c) \tag{17}$$

Here c is an hyper-parameter and the clipping is performed element-wise on the vector $\boldsymbol{\epsilon}$.

The SPI procedure detailed in (13) requires for each batch the computation of the proximal operator. This can be done through full gradient descent on the loss function defined by the batch sampled from the replay buffer at time step k. More in details, we run n_{prox} gradient descent steps to minimize the proximal loss defined as:

$$\begin{aligned} \mathcal{L}_k^{prox} = \ell_k(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\phi}) + \frac{1}{2\lambda} \, (& ||\boldsymbol{\theta}_1 - \boldsymbol{\theta}'_{1,k}||^2 \\ & +||\boldsymbol{\theta}_2 - \boldsymbol{\theta}'_{2,k}||^2 \\ & +||\boldsymbol{\phi} + \boldsymbol{\phi}'_k||^2 \;) \end{aligned} \tag{18}$$

We use a single hyper-parameter λ to control the strength of the proximal term for both the policy and the Q-networks. In our implementation, we replace the L_2 norm with the MSE in order to have all the proximal terms scaled with respect to number of parameters.

The resulting algorithm, called Proximal Deterministic Policy Gradient (PDPG), is summarized in Algorithm 1.

VI. Experimental results

We provide in this section a comprehensive performance analysis of the proposed method to assess its capabilities in terms of sample efficiency and asymptotic performance with particular focus on the former being of fundamental interest in real applications where querying the environment for new samples is extremely expensive. In such a scenario being able to trade-off sample requirements with policy optimality is fundamental; in fact, it may be preferable to reach reasonable performance with few samples than having high asymptotic capabilities.

We train our agent on Mujoco [24] OpenAI gym [25] continuous control tasks, a challenging benchmark often used in literature to test RL algorithms dealing with continuous action and state spaces. We compare our method against state-of-the-art on-policy (PPO) and off-policy (TD3, SAC) algorithms. PPO [4] is an on-policy algorithm that exploit the policy gradient theorem and generalized advantage estimation [17] to learn an optimal policy. During training, a trust region constraint is imposed on the policy that is updated keeping it close to the one at the previous iteration; this results in more stable training and better performance. TD3 [9] is the algorithm which our method is based on, it learns a deterministic policy exploiting the deterministic policy gradient theorem and additional tricks describes in Section V to improve upon DDPG [10] which we do not include in our comparison being very similar to TD3 with lower performance. SAC [11] is a maximum entropy RL algorithm that learns a stochastic policy to maximize the total expected reward plus an entropy term in order to achieve high performance while being maximally exploring. TD3 and SAC are the most performing algorithms and show similar results.

Algorithm 1 PDPG

```
Input: $\tau$, $n_{prox}$, $\lambda$, batch size $n_B$, learning rate $\alpha$, exploration noise variance $\sigma$
Initialize network parameters $\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$, $\boldsymbol{\phi}$ randomly
$\boldsymbol{\theta}'_1 \leftarrow \boldsymbol{\theta}_1$, $\boldsymbol{\theta}'_2 \leftarrow \boldsymbol{\theta}_2$, $\boldsymbol{\phi}' \leftarrow \boldsymbol{\phi}$
repeat
  for $k = 1$ to $T$ do
    Collect transition $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', r)$ with exploratory action $\boldsymbol{a} = \pi_{\boldsymbol{\theta}}(\boldsymbol{s}) + \epsilon$ with $\epsilon \sim \mathcal{N}(0, \sigma \boldsymbol{I})$
    Store transition in the replay buffer $\mathcal{B}$

    Sample minibatch of $n_B$ transitions from the replay buffer $B_k \sim \mathcal{B}$
    Add clipped noise to actions in minibatch as in (17)
    for $i = 1$ to $n_{prox}$ do
      $\Delta\boldsymbol{\theta}_1$, $\Delta\boldsymbol{\theta}_2$, $\Delta\boldsymbol{\phi}$ = $\nabla\mathcal{L}_k^{prox}$
      $\boldsymbol{\theta}_1 \leftarrow \boldsymbol{\theta}_1 - \alpha\Delta\boldsymbol{\theta}_1$
      $\boldsymbol{\theta}_2 \leftarrow \boldsymbol{\theta}_1 - \alpha\Delta\boldsymbol{\theta}_2$
      $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \alpha\Delta\boldsymbol{\phi}$
    end for
    $\boldsymbol{\theta}'_1 \leftarrow \tau\boldsymbol{\theta}_1 + (1-\tau)\boldsymbol{\theta}'_1$
    $\boldsymbol{\theta}'_2 \leftarrow \tau\boldsymbol{\theta}_2 + (1-\tau)\boldsymbol{\theta}'_2$
    $\boldsymbol{\phi}' \leftarrow \tau\boldsymbol{\phi}(1-\tau)\boldsymbol{\phi}'$
  end for
until convergence
```

Figure 1 shows a comparison of the training curves of the algorithms listed above. We run our algorithm for ten different seeds in order to assess its stability with different conditions; the reward is averaged among ten episodes, keeping the default maximum episode length defined by the gym framework. In each plot, the solid lines represent the average value among different seeds while the shaded area indicates a single standard deviation from the average. The curves have been smoothed for visual clarity with an exponential average.

For TD3 we have reported the curves taken from the authors github repository [1] except for the environments where the authors provided curves for only 1 million steps (Ant,

[1] https://github.com/sfujim/TD3

TABLE I

Average time-steps required by each algorithms to reach the reward thresholds set approximately at one third and two thirds of the maximum reward achieved by the best algorithm.

	Ant		**HalfCheetah**		**Hopper**		**Humanoid**		**Walker2d**	
Thresholds	2000	4000	5000	10000	1000	2000	3000	6000	1800	3600
Ours	371500	**743500**	**101000**	**490500**	**92500**	**133000**	390000	**1426000**	**134500**	**227000**
SAC	**326000**	1135000	122000	624000	203000	299000	**288000**	2335000	314000	540000
TD3	432000	2002000	147500	1392500	147000	202000	402500	1750000	187500	365000
PPO	977306	822067	/	/	190874	359629	/	/	484557	231834

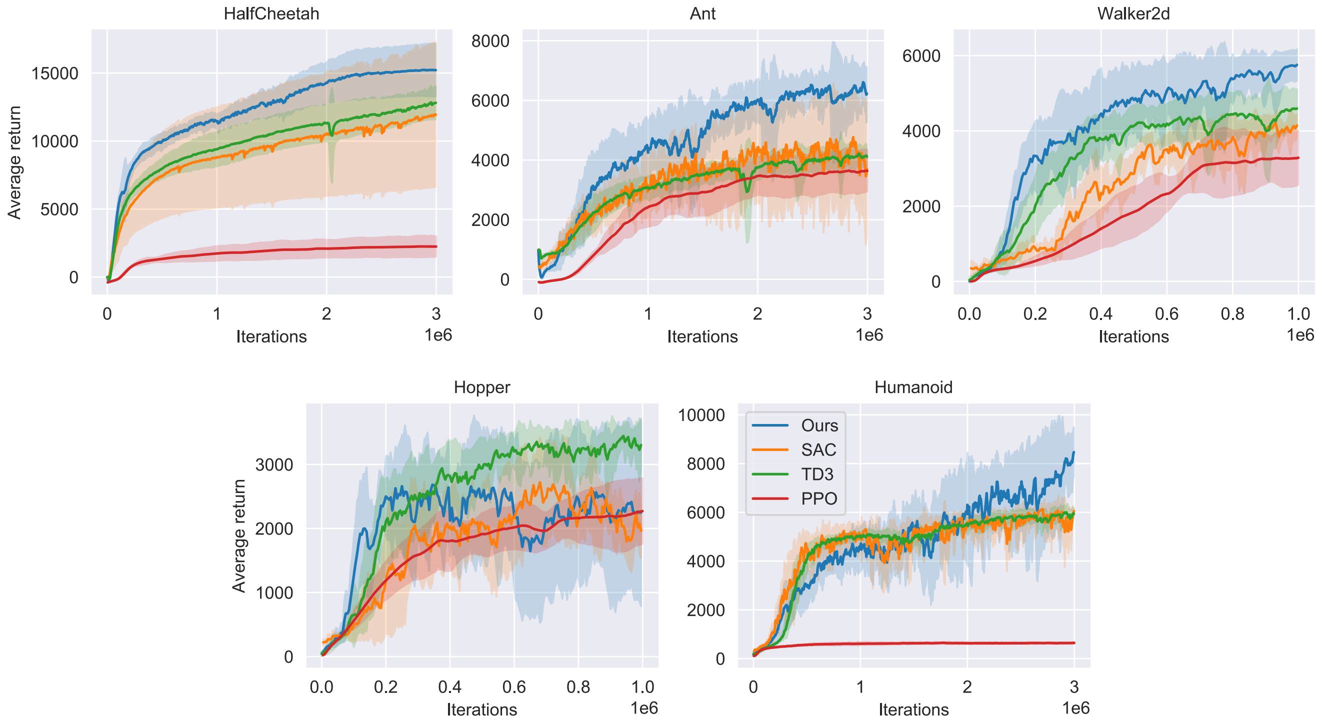

Fig. 1. Training curves on OpenAI gym continuous control benchmarks. Our methods consistently outperform concurrent approaches (SAC and TD3) and on policy methods (PPO).

HalfCheetah) or didn't provide them at all (Humanoid); for such cases we run the experiments with the code available in the same repository. For SAC we employed the curves available at the project website [2] and used the OpenAi baselines [3] code to run experiments with PPO.
It is noticeable how our method consistently outperform both on policy and off-policy methods on most continuous control tasks. The Hopper environment exhibits a quite noisy behavior but the average performance are comparable with the other algorithms. Moreover, it takes much less for our method to reach an average return equal to the maximum performance obtained by the other methods. See for example the HalfCheetah environment where PDPG is able to match the performance of TD3 with half the number of samples.

To better characterize sample efficiency we report in Table I the average number of time-steps required by each algorithm to exceed a set of reward thresholds placed at approximately one third and two thirds of the maximum reward achieved by the best algorithm. The superior sample efficiency of our method here is evident, for most of the environments PDPG reaches the specified thresholds with much less samples than the others. In the Humanoid and Ant it shows less efficiency than SAC for the lower threshold but better efficiency for the higher one, moreover, it has consistently better asymptotic performance. We acknowledge that this sample efficiency comes at a cost: the computation of the proximal operator requires multiple gradient steps for each batch, slowing down the training; in our experiments we took five gradient steps for each batch hence the amount of computations required scaled accordingly. We believe that this is a fair price to pay since it reduces significantly the number of queries to the environment needed to train the agent properly.

[2] https://sites.google.com/view/soft-actor-critic
[3] https://github.com/openai/baselines

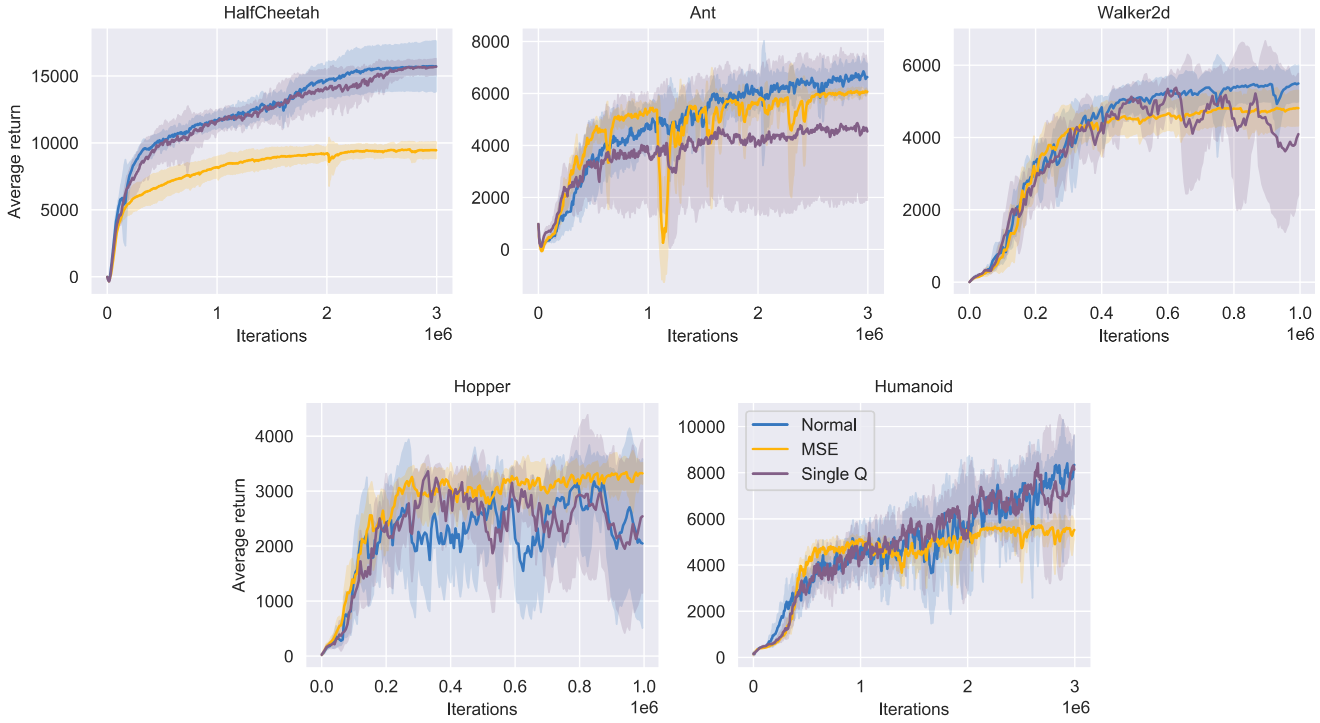

Fig. 2. Ablation study comparing the training curves of our method (blue) and its versions using the MSE in the TD error (yellow) and no bootstrapped value estimate (purple).

VII. Ablation Study

We provide in this section an ablation study to show the effect on the proposed method of the huber loss and the bootstrapped value estimation. We run a version of the method that employs the MSE in the TD error and also a version that doesn't use the bootstrapped estimation, but still keeps two Q-functions to avoid the overestimation bias.

In Figure 2 the training curves are reported for the two alternative versions compared to the standard approach. The employment of bootstrapped value estimates while not drastically changing performance seems to provide improved stability, this can be seen especially from the Ant and Walker environment where the Single Q algorithm has much higher variance. The huber loss has a drastic effect on the HalfCheetah environment, providing considerable performance improvement. On the remaining environments there is not substantial difference between the two loss functions.

In general, the standard method shows better performance and stability hence all the components employed are empirically justified by this study.

VIII. Conclusions

In this paper we proposed Proximal Deterministic Policy Gradient, an off-policy RL method for model free continuous control tasks that exploits proximal gradient methods and bootstrapping to better solve the TD error optimization problem. Proximal algorithms are appealing in an RL setting since they show improved convergence and stability properties compared to standard SGD. Moreover, we showed that proximal methods provide a natural interpretation of the target networks, a trick commonly employed in RL to stabilize training.

The resulting algorithm compare favourably with state-of-the-art off-policy and on-policy methods showing improved sample efficiency and asymptotic performance. The significant increase in sample efficiency makes our algorithm appealing for deployment in real environments, this possibility will be explored in a future work.

Appendix

A. Training details

We employed Feedforward Neural Networks with two hidden layers of 256 neurons each with ReLu [26] activations for both policy and critic.

As in [9] we perform a burn-in at the beginning of training where we sample random actions from the environment. The hyper-parameters employed are listed in Tables II, III. The exploration noise, action noise and noise clip are relative to the maximum value of the action that varies depending on the environment.

References

[1] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *NIPS*, S. A. Solla, T. K. Leen, and K. Müller, Eds. MIT Press, 2000, pp. 1008–1014.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

TABLE II

Hyper-parameter values used in all the environments.

Name	Value
Learning rate α	3e-4
Damping coefficient τ	0.005
Exploration noise σ	0.1
Noise clip c	0.5
Action noise σ_q	0.2
Batch size	256
Proximal steps n_{prox}	5
Policy loss strength β	0.01

TABLE III

Additional Hyper-parameter values.

Environment	Name	Value
Hopper, Walker	Burn-in	1000
All others	Burn-in	10000
HalfCheetah, Ant	Policy Weight Decay	0.0
All others	Policy Weight Decay	1e-5
Humanoid	Proximal Strength $(1/\lambda)$	10.0
Hopper, Walker	Proximal Strength $(1/\lambda)$	1.0
HalfCheetah, Ant	Proximal Strength $(1/\lambda)$	0.1

[3] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, 2015, pp. 1889–1897.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[5] A. Ilyas, L. Engstrom, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "Are deep policy gradient algorithms truly policy gradient algorithms?" *arXiv preprint arXiv:1811.02553*, 2018.

[6] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," 2014.

[7] S. Thrun and A. Schwartz, "Issues in using function approximation for reinforcement learning," in *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, 1993.

[8] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[9] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *arXiv preprint arXiv:1802.09477*, 2018.

[10] 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," *arXiv preprint arXiv:1509.02971*, 2015.

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

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[13] N. Parikh, S. Boyd, *et al.*, "Proximal algorithms," *Foundations and Trends® in Optimization*, vol. 1, no. 3, pp. 127–239, 2014.

[14] E. K. Ryu and S. Boyd, "Stochastic proximal iteration: a non-asymptotic improvement upon stochastic gradient descent," *Author website, early draft*, 2014.

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

[16] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *ICML*, 2016, pp. 1928–1937.

[17] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv:1506.02438*, 2015.

[18] R. Fakoor, P. Chaudhari, and A. J. Smola, "P3o: Policy-on policy-off policy optimization," *arXiv preprint arXiv:1905.01756*, 2019.

[19] B. Dai, A. Shaw, L. Li, L. Xiao, N. He, Z. Liu, J. Chen, and L. Song, "Sbeed: Convergent reinforcement learning with nonlinear function approximation," *arXiv preprint arXiv:1712.10285*, 2017.

[20] Y. Feng, L. Li, and Q. Liu, "A kernel loss for solving the bellman equation," in *Advances in Neural Information Processing Systems*, 2019, pp. 15 430–15 441.

[21] P. Chaudhari, A. Oberman, S. Osher, S. Soatto, and G. Carlier, "Deep relaxation: partial differential equations for optimizing deep neural networks," *Research in the Mathematical Sciences*, vol. 5, no. 3, p. 30, 2018.

[22] D. P. Bertsekas, "Reinforcement learning and optimal control," *Athena Scientific*, vol. 1, 2019.

[23] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

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

[25] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv:1606.01540*, 2016.

[26] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 315–323.