Reproduction of *Exploration in Policy Mirror Descent*

Ethan Brooks

November 5, 2021

Introduction

Reproducing this paper is an important project for me for reasons both personal and general. The general reason is that the results presented in the paper are significantly state-of-the-art, and they will become the new baseline to compare future results against. Not only will a faster baseline speed up research, but the expectation within the field is that an author compares any new method with the current state-of-the-art.

As for the personal reasons, earlier this semester I was very excited about an idea relating to *Trust Region Policy Optimization* [1] which I had begun to develop into a conference submission. When I described the idea to a lab mate, he pointed me to two papers, *Guided Policy Search via Approximate Mirror Descent* [2] and this paper, *Exploration in Policy Mirror Descent* (EPMD) [3], which were sufficiently similar to my idea that I abandoned the project. Ever since I have been curious if this idea held any water.

I am also interested in the fact that, despite the impressiveness of the results that this paper reports, it was rejected from ICLR. By reproducing this paper and corresponding with its authors, I hope to gain some insight into the expectations that at least this particular conference places on paper submissions and to learn what sort of errors can thwart a submission as impressive as this one.

Work being replicated

In this section, I will give some background for EPMD and then describe specifically which aspects I replicated.

Background

There are several ways to justify the loss function that the EPMD authors propose, but the clearest, in my opinion, is by appeal to *Trust Region Policy Optimization* (TRPO). TRPO observes that small changes to policy parameters can yield large and potentially detrimental changes to policy distributions (the probabilities assigned to actions in a state). The paper presents theoretical and empirical evidence that enforcing a constraint on the KL Divergence between the pre- and post-update policy distributions can yield significant improvements in the performance of the agent.

TRPO proposes an algorithm that computes the optimal change in parameters given the policy gradient and the aforementioned constraint. In mathematical terms:

$$\max_{\theta} E_{s\sim\rho_{\bar{\pi}},a\sim\bar{\pi}(\cdot|s)}\left[\frac{\pi_\theta(a|s)}{\bar{\pi}(a|s)}Q^{\bar{\pi}}(s,a)\right] \tag{1}$$

subject to

$$E_{s\sim\rho_{\bar{\pi}}}\left[\mathcal{D}(\bar{\pi}(\cdot|s)\parallel\pi_\theta(\cdot|s))\right]<\delta \tag{2}$$

where $\bar{\pi}$ is the old policy, π_θ is the updated policy, and $s\sim\rho_{\bar{\pi}}$ indicates that state s is sampled from a trajectory generated by policy $\bar{\pi}$. What is relevant to our purposes is that expression 1 maximizes the expected improvement in cumulative reward while expression 2 enforces the aforementioned constraint between the old and the new policy.

The greatest drawback of TRPO is that the algorithm involves inverting a matrix, which complicates the implementation and drags down computational performance. Simpler rival algorithms arose in the aftermath of TRPO (e.g. Proximal Policy optimization [4]), but one of the most compelling was *Guided Policy Search via Approximate Mirror Descent* (GPSAMD) [2], which observed that expression 1 is in fact convex in π_θ. Thus, one can compute the optimal policy satisfying the constrained optimization without any matrix inversions and then train the actual policy to match this optimal policy using supervised learning (This was essentially the idea that I had been working on).

Exploration in Policy Mirror Descent

EPMD offers two contributions beyond those of GPSAMD. First it replaces the constraint (expression 2) with a "KL penalty" and adds an entropy bonus:

$$\max_{\theta} E_{\ \rho\sim\pi_\theta}\left[r(\rho)-\tau\mathcal{D}(\pi_\theta\parallel\bar{\pi}))+\tau'\mathcal{H}(\pi_\theta)\right] \tag{3}$$

where $\rho \sim \pi_\theta$ is a trajectory generated by policy π_θ, $r(\rho)$ is the accumulated reward, τ and τ' are hyperparameters, and $\mathcal{H}$ signifies entropy. The second term of this expression represents the KL penalty and the third term encourages exploration (a high entropy policy will take more random actions and visit more states in the environment). It also uses theory from earlier work [5] to derive the closed form policy that optimizes this objective:

$$\pi^*(\rho) = \frac{\bar{\pi}(\rho) \exp\left\{\frac{r(\rho) - \tau' \log \bar{\pi}(\rho)}{\tau + \tau'}\right\}}{\sum_{\rho'} \bar{\pi}(\rho') \exp\left\{\frac{r(\rho') - \tau' \log \bar{\pi}(\rho')}{\tau + \tau'}\right\}} \tag{4}$$

The second contribution of EPMD is to modify the supervised learning loss as follows: while GPSAMD (and some other work, e.g. Soft Actor Critic [6]) use the loss function

$$\pi_\theta \leftarrow \arg\min_{\pi} \mathcal{D}(\pi \parallel \pi^*) \tag{5}$$

EPMD reverses the KL expression:

$$\pi_\theta \leftarrow \arg\min_{\pi} \mathcal{D}(\pi^* \parallel \pi), \tag{6}$$

The paper observes that expression 5 is more likely to drive the densities of π_θ to zero for many actions, causing "mode collapse," where a policy permanently stops visiting some state/action pairs. In contrast, expression 6 discourages zero values, so that a policy will revisit even those state/action pairs that initially yielded poor values. This is important because a state/action pair might dramatically improve its value later in learning under an improved policy.

Minimizing the reverse KL Divergence is not trivial, and it turns out that modifications in equation 3 make the gradient mathematically tractable. Specifically, any KL Divergence expression entails an expectation over the first of the two distributions. While we can easily sample from the current distribution π_θ to compute a gradient for 5, we cannot directly sample from π^* in 6. Fortunately, with some mathematical trickery, we can compute this expression by instead sampling from $\bar{\pi}$, courtesy of the second term in expression 3:

$$\nabla_\theta \mathcal{D}(\pi^* \parallel \pi) = E_{s \sim \mathcal{D}, a \sim \bar{\pi}} \left[\pi^*(a|s) \log \pi_\theta(a|s)\right] \tag{7}$$

where

$$\pi^*(a|s) := \exp\left\{\frac{Q(s,a) - \tau' \log \bar{\pi}(a|s) - V(s)}{\tau + \tau'}\right\} \tag{8}$$

Practically – and a significant reason for my interest – the paper proposes an algorithm that is much easier to implement than GPSAMD, which requires the programmer to implement a "Linear Quadratic Regulator" for optimizing certain convex functions. In contrast, EPMD requires few changes to existing off-policy algorithms, besides a modified loss function.

Replication

Algorithm

EPMD proposes two algorithms:

1. "Reversed Entropy Policy Mirror Descent" (REPMD) does not use value functions or bootstrapping and has more theoretical guarantees. The paper applies REPMD to the "algorithmic" toy problems from OpenAI gym.

2. "Policy Mirror Actor Critic" (PMAC) introduces a value approximator to REPMD by expressing the optimal policy (4) in terms of the optimal value function. Value approximators can use bootstrapping, which dramatically improves sample efficiency. Consequently, the algorithm can be applied to complex environments like the MuJoCo [7] domain.

Because my interest in this paper is largely practical, and the results from PMAC incorporate the contributions of REPMD, I chose to replicate PMAC.

Environments

My goal was to replicate the blue lines in figure 1. This figure depicts performance on five common OpenAI Gym benchmarks implemented with the MuJoCo physics simulator. Each environment contains a multi-jointed robot. Reward is proportional to the speed at which the robot moves forward. Thus the task of the learning algorithm is to learn patterns of joint activation that propel the robot in one direction as quickly as possible. For detailed descriptions, consult the OpenAI Gym website.

One virtue of these environments is that in principle, there is no upper bound on the quality of the policy learned – the robot can always move faster. Thus one can evaluate both the *speed* of learning as well as the *quality* of learning, since better algorithms will converge to a more effective policy.

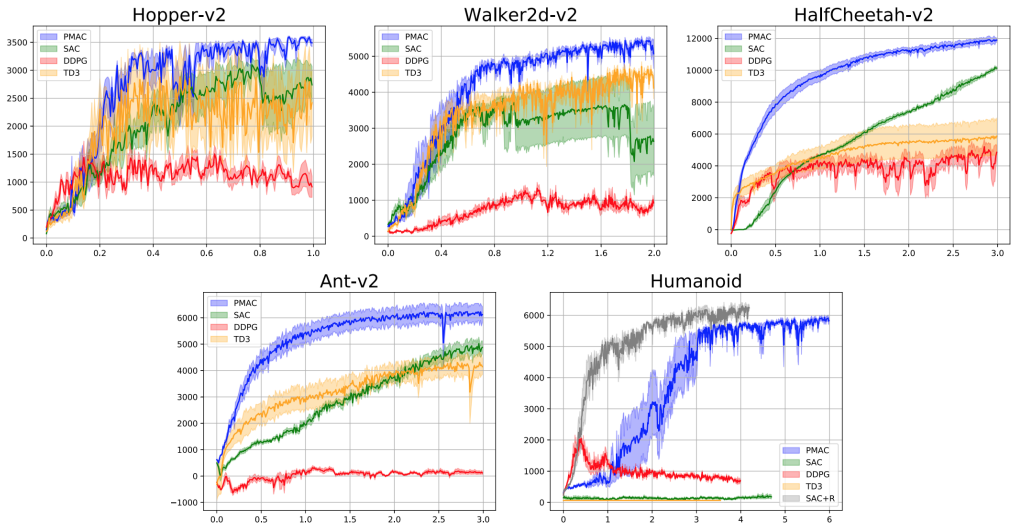


Figure 1: Learning curves of Deep Deterministic Policy Gradient [8] (red), Twin Delayed Deep Deterministic policy gradient [9] (yellow), Soft Actor Critic (SAC) [6] (green) and Policy Mirror Actor Critic (blue) on Mujoco tasks (with SAC with the reparameterization trick (gray) added on Humanoid). Plots show mean reward with standard error during training, averaged over five different instances with different random seeds. X-axis is millions of environment steps.

Replication Activities

I began this project by reading the paper in detail and determining that the policy was off-policy and bore significant resemblance to the Soft Actor Critic (SAC) algorithm. Indeed, SAC optimizes the same expression (3) but omits the second term (the KL penalty) and uses the original, non-reversed KL objective (5). I found a well-written implementation of SAC [10] and began adapting the code to the specifications of PMAC provided in the paper.

Despite my initial assumption that this process would go smoothly, I encountered several harrowing pitfalls that initially thwarted my efforts. I will briefly discuss the two that were most interesting and challenging.

First, despite the *apparent* similarity between the loss functions in SAC and PMAC, it is easy to overlook the fact that PMAC samples actions from the *reference policy*, $\bar{\pi}$, not the current policy, π_θ. In theory, the reference policy can be any distribution. However, the authors propose the the pre-update policy – i.e. the policy before the current gradient update. This requires the programmer to duplicate the current policy before performing any gradient updates and then sample actions used in the policy update

from the reference policy, given the states sampled from the replay buffer (a necessary component of off-policy learning [11]).

A second innocuous detail that contained, as they say, a devil: some implementations of reinforcement learning algorithms normalize the outputs of the policy network. Most training environments provide upper and lower bounds on the action space. In order to spare the network from having to learn these bounds, these implementations use the tanh function to squash the network outputs to the range $(-1, 1)$ and then scale the result to the action-space bounds. For a stochastic policy (like PMAC), the network outputs mean and standard deviation which parameterize a Gaussian distribution. The algorithm samples an action from this distribution, applies the tanh, and finally scales to the action space. Thus, when calculating log probabilities, one must take care to use the *pre*-tanh network outputs, since this is what was actually sampled from the distribution.

The third bug that proved the most elusive was that even after using the pre-tanh value, one could not simply use the log prob of the pre-tanh value for the log prob of the action, i.e.

$$\log P(a) \neq \log P(z) \tag{9}$$

where a is the action and z is the pre-tanh value ($a = \tanh(z)$). Au contraire,

$$\log P(a) = \log P(z) - \log(1 - a^2) \tag{10}$$

This last bug cost me *extensive* debugging efforts.

Despite these debugging discoveries, the algorithm failed to learn even on the simplest environments (`Pendulum-v0`). In desperation, I appealed to the mercy (and perhaps the pride) of the first author, who responded quickly and willingly shared his source code with me. After confirming that his code quickly mastered `Pendulum-v0`, I performed a detailed side-by-side comparison of his code with mine, and ascertained that he had incorporated several optimizations that the paper had omitted:

1. His implementation clamps the target policy (expression 8) between zero and 0.9.

2. He adds a regularization loss $.001 * \mu^2$ on the value of the Gaussian mean output by the policy network.

3. He adds a regularization loss $.001 * \sigma^2$ on the value of the Gaussian standard deviation output by the policy network.

Removing any one of these optimizations caused the algorithm to fail to learn. None of these optimizations are theoretically justified. It is worth noting that while a probability is constrained to the range $[0, 1]$, a probability density can assume any positive value. Thus, there is actually a theoretical argument *against* the clamp.

A second important detail that thwarted my initial implementation was the claim in the paper that the algorithm performs 100 gradient updates per environment step. It is very unusual to perform more than one gradient update per environment step (and common to perform far fewer). Through communication with the author and analysis of his source code, I ascertained that this was a typo and the algorithm, like most of its kind, performs only one gradient update per environment step.

An experienced practitioner might argue that any one of these optimizations or changes is obvious or could be inferred. But collectively, they compounded the difficulty of implementation and I confess that without access to the author's source code, it is unlikely that I ever could have gotten the algorithm to work. I speculate that this problem is endemic to research papers, since authors rarely have the opportunity to "debug" the specifications in their papers by having someone else use it to implement the algorithm. I was fortunate to have access to the main author, but I wonder if someone without an academic email address would have received the same response.

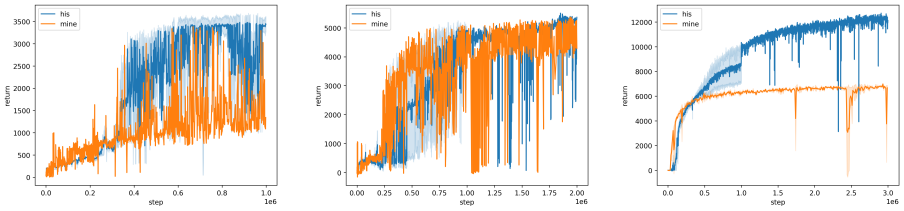
Experiments

I ran experiments on all five MuJoCo environments from the original paper. To check the honesty of the results that the authors reported in the original paper, I ran their source code alongside my implementation and graphed the relative performance of both implementations (figure 2). As the graphs demonstrate, I succeeded in reproducing some of the authors' graphs, but their implementation clearly outperformed mine on some. This suggests that there may have be a bug that remains unfound.

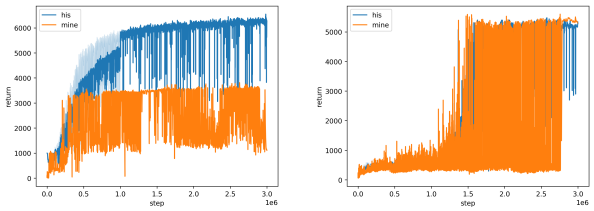
A few things to note about the results:

1. The first author's implementation closely matches the results in his paper. This is impressive because I did not use a random seed and I have found in the past that most off-policy algorithms (including Soft Actor Critic) exhibit high variance between seeds. Indeed, the error bars on the plots of his implementation are quite narrow.

2. His results are state of the art relative to the Soft Actor Critic (SAC) algorithm as reported in *that* paper. However, the SAC results in his



(a) Hopper-v2 (b) Walker-v2 (c) HalfCheetah-v2



(d) Ant-v2 (e) Humanoid-v2

Figure 2: Comparison of my implementation with the implementation received from the first author. Results drawn from two random seeds per line.

Figure 3: Results on MuJoCo environments reported in the Soft Actor Critic paper.

paper seem not to match the results in the SAC paper (figure 3). Comparing his results with those reported in the SAC paper, in most tasks, he matches SAC. He exceeds SAC's performance on Walker2d, but actually underperforms SAC on HalfCheetah. Some of the SAC discrepancies are due to the fact that he runs some of these environments for a different number of time-steps than the SAC paper. Another possible reason is that the SAC paper reports results for version 1 of these environments whereas he reports results for version 2. Looking at the github repository for the OpenAI Gym code, there do not appear to be any obvious across the board changes the environments that would hurt performance across the board.

3. I did not have time to run the Humanoid environment for the 6 million steps that he presents in his paper. Indeed one of the two seeds that I ran appears to have flat-lined, but may have shown signs of life with some more training.

Extensions

Conclusion

Parenthetically, before reflecting on the experience of reproducing this paper, I will explain the mystery of this paper's initial rejection. According to the first author, a theoretical claim about global convergence was not correct. The revised paper presents a proof about sublinear regret. Despite my initial indignation that a paper with such impressive results should be rejected, I will concede to the reviewers that these results do not outweight an incorrect theoretical claim.

When I began this project, I was hesitant to reach out to the authors about source code because it seemed like cheating and candidly, I wanted to put my skills to the test. In retrospect, much of the time I spent on this project before accessing the authors' code was wasted. It is unlikely that I would have independently discovered the specific combination of optimizations and associated hyperparameters that made the authors code work without an extensive investigation. In the future, accessing working source code will be my *first* step before attempting replication.

A somewhat daunting realization that I owe to this project is that small optimizations can prove the difference between success and failure. It is shocking to recognize that, had I pursued this idea myself, I might have abandoned it after initial implementations failed without the optimizations mentioned in the Replication Activities section. In some ways, this should encourage researchers to keep pursuing an idea even after initial failures, but it makes it difficult to ascertain whether an idea fails because of inherent flaws or because of implementation details that can be fixed.

References

[1] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[2] William H Montgomery and Sergey Levine. Guided policy search via approximate mirror descent. In *Advances in Neural Information Processing Systems*, pages 4008–4016, 2016.

[3] Jincheng Mei, Chenjun Xiao, Ruitong Huang, Dale Schuurmans, and Martin Muller. Exploration in policy mirror descent. ICLR 2019 Conference Withdrawn Submission, 2018.

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[5] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2775–2785, 2017.

[6] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[7] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[8] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[9] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[10] Navi Mumbai. pranz24/pytorch-soft-actor-critic, Apr 2019.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.