Proximal Policy Optimazation with Optimal Value Estimation

Nicolas Mon [*1] **Zihao Fan** [*1]

Abstract

Proximal Policy Optimization has recently emerged as a strong performing reinforcement learning algorithm. Through its clipped surrogate loss function, PPO allows one to essentially perform trust region policy updates in a way that is compatible with Stochastic gradient descent. Because of this feature, PPO is able to achieve good performance while remaining a relatively simple algorithm with few moving parts. However, we believe more sample efficient learning can be achieved if we compute the gradient update of PPO via an estimation of the *optimal* value function as opposed to the value function of the current policy. To this end, we propose a novel algorithm that combines the clipped surrogate objective function of PPO an optimal value function estimator from the also popular Deep Deterministic Policy Gradient Algorithm. We describe our algorithm in detail, explore the effects of different hyper-parameters settings on our new algorithm, and finally compare results from the best found settings to results from traditional PPO. While our results are somewhat promising, more experiments need to be done to further assess our algorithm's potential gains in sample efficiency.

1. Introduction

Deep Neural Networks have been shown to be able to both learn effective representations of data and also have proven to be effective function approximators. As such, they have come to find use within Reinforcement learning algorithms, approximating optimal policies, value functions, or system dynamics. (Arulkumaran et al., 2017)

Further, there has been a significant amount of recent research in using deep neural networks as policy networks, and optimizing them directly with a version gradient descent. While at these first Vanilla policy gradient methods suffered from poor sample efficiency and lack of robustness, many algorithmic advancements have been made in regards to these types of approaches. One such approach that has led to increases in performance of many Policy gradient algorithms is the use of experience replay memory for storing the data an agent creates through interacting with its environment. Experience Replay serves to decorrelate samples as batches are sampled stochastically from the Experience replay buffer, resulting in less variance between in gradient updates and thus more stable learning (Mnih et al., 2016)

However, even so traditional policy gradient methods still suffered a lack of robustness. These models tended to be unstable, as you needed to be very careful about the learning rate, and the size of your gradient updates to ensure your policy does not diverge. Thankfully, Schulman et al. introduced an algorithm known as Trust Region Policy Optimization (TRPO), which guarantees monotonic improvement and thus performs much better at optimizing large nonlinear policies such as neural networks. This breakthrough was a huge step forward for the field of Deep Reinforcement Learning, but unfortunately the algorithm is relatively complicated and difficult to implement.(Schulman et al., 2015)

Next came the Proximal Policy Optimization, a relatively new algorithm that claims the benefits if TRPO while remaining relatively simple and clean. Proximal Policy Optimization bakes a gradient clip into its surrogate objective function, allowing the algorithm the same data efficiency and robustness of TRPO while still only using first order optimization. Because PPO strikes such a nice balance between performance, data efficiency, and simplicity, it has emerged as a solid algorithm for the Deep Reinforcement learning toolbox. (Schulman et al., 2017)

2. Proximal Policy Optimization and DDPG

Proximal Policy Optimization (PPO) proposed by (Schulman et al., 2017) aims to provide a policy gradient method that has a balance between sample complexity, computation simplicity and well-time. It is proved to perform better than A2C, CEM and similar to ACER on both continous tasks and Atari environments. DDPG by (Lillicrap et al., 2015) generalized DPG by adding a replay buffer and a tar-

[*]Equal contribution [1]School of Information, UC Berkeley, Berkeley, USA. Correspondence to: <>.

Final project for CS294-112, Fall, 2017

get network into the training and achieved robust learning in continous environments. In this section, we review the objective functions and algorithms of PPO and DDPG and state our notations.

2.1. Policy Gradient Methods

Policy gradient methods are a family of on-policy methods that estimates the expectation of current policy returns and improve the policy by taking gradient ascent steps. The objective function of policy gradient methods are most commonly formulated as follows,

$$L^{PG} = \hat{E}_t[log\pi_\theta(a_t|s_t)\hat{A}_t] \tag{1}$$

where π_θ is a stochasitc policy and the hat sign denotes the empirical expectation. Policy gradient methods are consider to have high variance so advantage $\hat{A}_t$ is used here instead of return $\hat{R}_t$ for variance reduction. Empirically, taking multiple optimization on one roll out leads to large policy updates and may fail to get a good policy, which means vanilla policy gradient methods are sample inefficient.

2.2. TRPO

Vanilla policy gradients are unstable to train and have no guarantee that the policy would improve since the optimization is taken within parameter space. TRPO (Schulman et al., 2015) instead proposed a surrogate objective function with a constraint on the size of policy update. The surrogate loss is

$$max_\theta \hat{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t] \tag{2}$$

and the optimization is subject the constraint of the KL-divergence between updates should not exceed a fixed amount δ. Empirically, the hard constraint can be alternatively achieved by penalize a KL divergence term, making the loss function looks like this,

$$max_\theta \hat{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\theta_{old}}(*|s_t), \pi_\theta(*|s_t)]] \tag{3}$$

The surrogate loss of TRPO is a pessimistic bound of the true performance of policy π. Optimizing this surrogate loss in theory is guaranteed to improve the policy. However, as reported in (Schulman et al., 2017), the coefficient β for the KL divergence term is hard to select.

2.3. PPO

PPO also uses a CLIP surrogate loss that is also a pessimistic bound of the true performance of the policy π. The

Algorithm 1 PPO

for iteration=1, 2, ... **do**
  for actor=1, 2, ..., N **do**
    Run policy π_θ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
    Optimize surrogate L w.r.t. θ, with K epochs and mini batch size M $\leq$ NT
    $\theta_{old} \leftarrow \theta$
  end for
end for

CLIP loss of PPO is

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{4}$$

which penalize large updates on the policy while keeping the computation efficient. PPO can be considered as a simple substitution for TRPO.

In addition to the clipped policy gradient loss, the optimization also needs to compute a state-dependent advantage function which has a biased but much lower variance estimation of the returns. This advantage function is also computed by jointly learning a state value function $V(s)$. Specifically, $V(s)$ can be modeled as an auxiliary output of the policy network, which could take advantage of the shared weights of the policy network. Also, to explicitly introduce a loss that encourages exploration, a entropy term is usually added. The full objective function for PPO in our baseline model is

$$L(\theta) = \hat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{5}$$

In the training phase, we run the policy for a fixed T time steps and compute the objective function with the sampled roll out. The advantage estimation $\hat{A}_t$ can be therefore computed by

$$\hat{A}_t = \delta_t + (\gamma\lambda\delta_{t+1}) + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{6}$$

where

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{7}$$

The full PPO algorithm is shown in Algorithm 1.

2.4. DDPG

DDPG is a model-free, off-policy actor-critic algorithm that learns policies in continuous action space with two separate neural networks, an actor network $\mu(s|\theta^\mu)$ and a critic network $Q(s, a|\theta^Q)$. DDPG was inspired by DQN (Mnih et al., 2013) and achieved a robust and stable value function learning with a replay buffer and a target network. Since the algorithm uses a regression loss on the Bellman error. It's critical to set problem as a well-defined supervised learning algorithm, which has a stable target and i.i.d.

data. The replay buffer can minimized the correlation between transitions and target network makes the regression object stable. DDPG learns the policy through the following two objectives.

$$L(\theta^Q) = E[(Q(s_t, a_t|\theta^Q) - r(s_t, a_t) \quad (8)$$
$$- \gamma Q(s_{t+1}, \mu(s_{t+1}|\theta^Q)))^2] \quad (9)$$

$$J_{\theta^\mu} = E[Q(s, a_{\theta^\mu}|\theta^Q)] \quad (10)$$

3. Algorithm

Given the computation of advantage function $\hat{A}_t$, we find that since the advantage is computed by the value network that has shared weights with the policy network, it also gets optimized at the same frequency as the policy network. The estimation error will therefore propagate through the L^{CLIP} term which will slow down the training of the policy. We believe that, since a critic network can be learned off-policy, we could decouple the training of value function $V(s|\theta)$ and the policy $\pi_\theta(s)$ by using a separate critic network. If the value function could get update more frequently than the policy network, it could provide more accurate advantage estimation thereby accelerate the training and increase the sample efficiency.

To this end, we decide to combine a independent critic network trained off-policy into the PPO algorithm. By introducing this critic network, we have to introduce a extra policy network as well to train jointly in the continuous space environments. Finally we end up having a independent DDPG algorihtm trained within the PPO training loop.

We denote the actor network and critic network as $\mu_\psi(s)$ and $Q_\varphi(s, a)$. After introducing a separate critic network, the object function of the PPO algorithm would be

$$L(\theta) = \hat{E}_t[L_t^{CLIP}(\theta) - c_1(\pi_\theta - \mu_\psi)^2 + c_2 S[\pi_\theta](s_t)] \quad (11)$$

and the advantage $\hat{A}_t$ can be computed by the new δ_t,

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \quad (12)$$
$$= r_t + \gamma E_a[Q(s_{t+1}, a)] - E_a[Q(s_t, a_t)] \quad (13)$$
$$= r_t + \gamma Q(s_{t+1}, \mu(s_{t+1})) - Q(s_t, \mu(s_t)) \quad (14)$$

The $c1$ here is a loss term that encourages our policy network produce a similar mean action to the action selected by $\mu(s)$. This term is only introduced at the beginning part of the training and will be annealed to zero after some iterations.

Our PPO based policy gradient method with DDPG as value function estimator can be summarized as Algorithm 2

Algorithm 2 PPO with optimal value estimation

Initialize PPO policy network$\pi_\theta(s)$
Initialize DDPG Q network $Q_\varphi(s, a)$ and policy network $\mu_\psi(s)$
for iteration=1, 2, ... **do**
  for actor=1, 2, ..., N **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps and save the transitions to the Replay Buffer RB
    Compute advantage estimates $\hat{A}_1$, ..., $\hat{A}_T$ with $Q_\varphi(s, a)$ and $\mu_\psi(s)$
  end for
  Optimize surrogate L wrt θ, with K epochs and mini batch size M $\leq$ NT
  $\theta_{old} \leftarrow \theta$
  Sample transitions TR from RB
  Update $Q_\varphi(s, a)$ and $\mu_\psi(s)$ for ddpg_per_ppo * ($\frac{NT}{M}$) times with TR
  if iteration $>$ target_network_lagging **then**
    Update $Q_\varphi^{target}(s, a)$ and $\mu_\psi^{target}(s)$
  end if
end for

4. Experiments

In order to gain a perspective on how each hyperparameter would affect our algorithms performance, we first ran hyperparameter sweeps on 3 separate parameters in order to view their effect. We ran these sweeps in the half cheetah environment to find good hyperparameter values for our algorithm, while keeping the values of the other hyperparameters constant during the sweep. Due to limitations in time and computing power, we were only conducted one run at each hyperparameter while leaving the rest at default values. If we had more time, we would more rigorously test the effects of different hyperparameter settings, either through multiple runs, a grid search, or by testing in different environments.

The parameters we tested in separate sweeps include how many updates of the DDPG networks per PPO policy update, how fast we anneal to pure PPO, and the entropy coefficient. The results of these sweeps are shown as Figure 1, Figure 2 and Figure 3.

After conducting these hyperparameter sweeps, we picked the top performing values and tested our algorithm with these hyperparameters against Open AIs PPO2 baseline performance at its default hyperparameters. We tested these algorithms with the HalfCheetah and the Hopper, running for 3 rollouts in each environment. The hyperparameter values we selected include 8 ddpg updates per ppo update, a 0.0001 coefficient of entropy, and annealing over 50 updates.

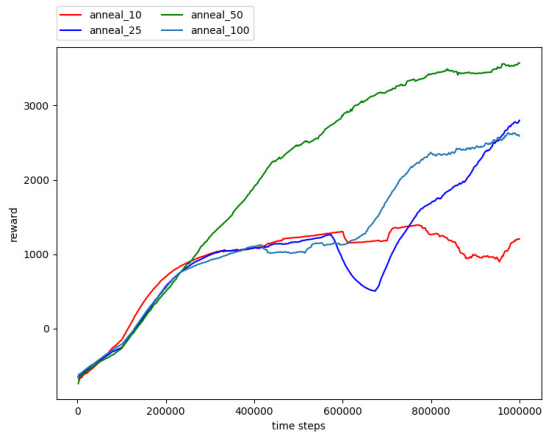

Figure 1. Annealing speed sweep

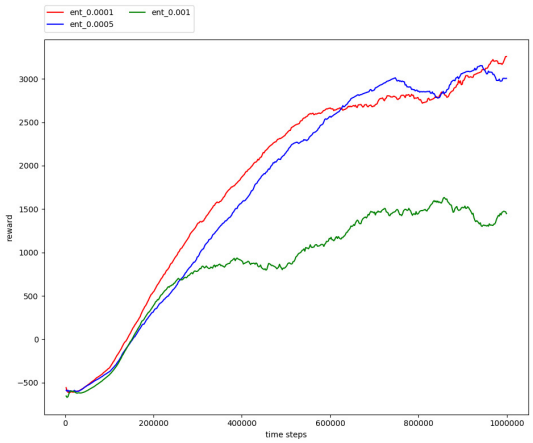

Figure 3. Entropy coefficient sweep

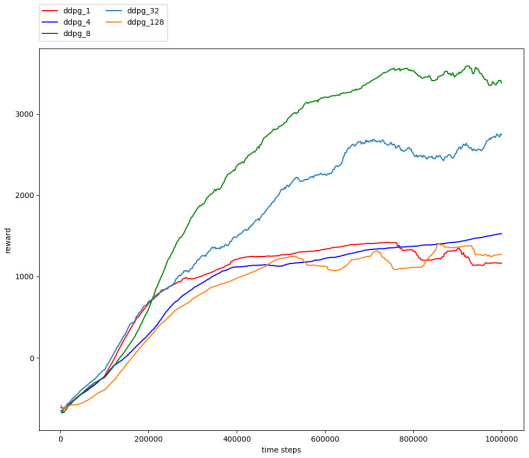

Figure 2. DDPG per PPO sweep

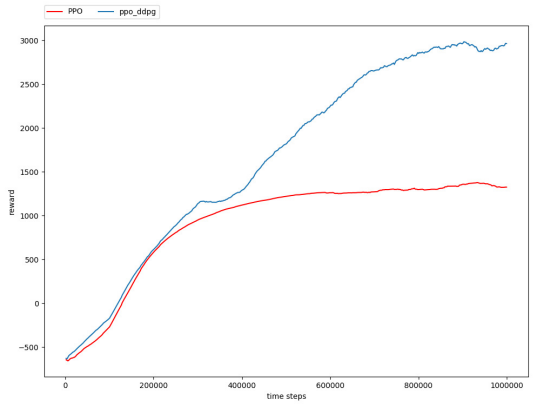

Figure 4. Learning curve comparison on HalfCheetah-V1

5. Conclusion

Our initial results from these experiments are somewhat promising. After finding good hyperparameters for the HalfCheetah Environment, our new algorithm outperformed the performance of Open AI's default PPO implementation. It is important to take this result with a grain of salt, as the default PPO implementation outperformed our model on the Hopper environment. This result is not entirely surprising: we found optimal hyperparameters for the HalfCheetah, while testing it again a PPO implementation whose default environment is Hopper. Indeed, this was not an absolutely fair comparison. Therefore, we need to conduct more experiments to really find out if our algorithm does give the increased sample efficiency that we hope. Based on our current results, we say this seems promising but more research need to be done.

References

Arulkumaran, Kai, Deisenroth, Marc Peter, Brundage, Miles, and Bharath, Anil Anthony. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

Langley, P. Crafting papers on machine learning. In Langley, Pat (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Lillicrap, Timothy P, Hunt, Jonathan J, Pritzel, Alexander,

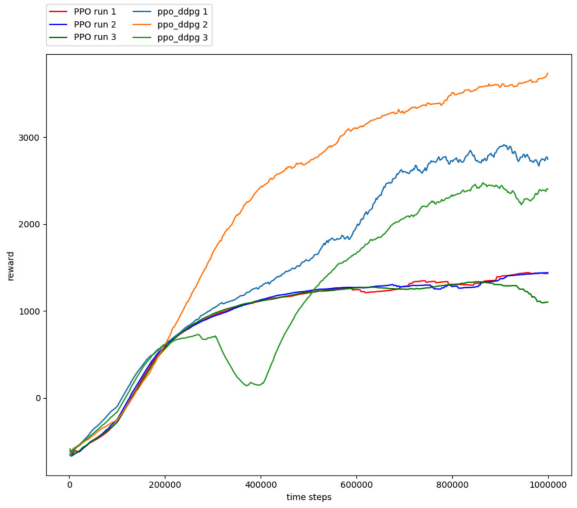

Figure 5. 3 Runs on HalfCheetah-V1

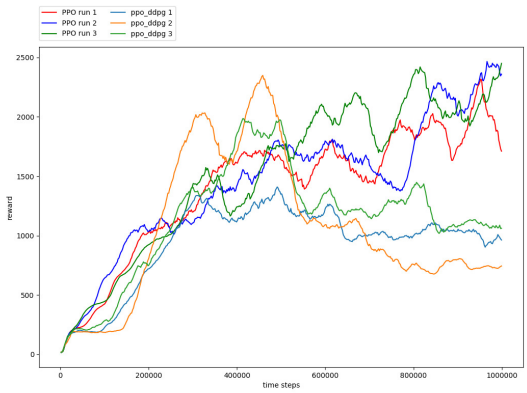

Figure 6. 3 Runs on Hopper-V1

Heess, Nicolas, Erez, Tom, Tassa, Yuval, Silver, David, and Wierstra, Daan. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Mnih, Volodymyr, Badia, Adria Puigdomenech, Mirza, Mehdi, Graves, Alex, Lillicrap, Timothy, Harley, Tim, Silver, David, and Kavukcuoglu, Koray. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1928–1937, 2016.

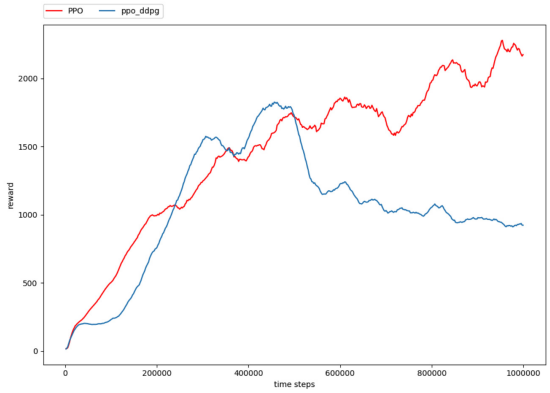

Figure 7. Learning curve comparison on Hopper-V1

Schulman, John, Levine, Sergey, Abbeel, Pieter, Jordan, Michael, and Moritz, Philipp. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1889–1897, 2015.

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