ASSESSING GENERALIZATION OF PROXIMAL POLICY OPTIMIZATION ALGORITHM IN PROCGEN

A PREPRINT

Minjune Hwang, Tony Tu, Richard Mao

Department of Electrical Engineering and Computer Science
University of California, Berkeley

{mjhwang, tonytu16, richard.mao}@berkeley.edu

May 14, 2020

ABSTRACT

Deep reinforcement learning (RL) algorithms have reported successful results in numerous benchmark environments and real-world problems across different fields. Different algorithms and adjustments were proposed in literature to increase single-task performances on a limited set of benchmarks, including MuJoCo robotics control tasks [13] implemented in OpenAI gym [1] and the Atari Domain, yet generalization of a learning agent within the model and across different environments is less focused. Thus, we investigate sample efficiency and generalization performance with ProcGen [2, 3], a novel collection of game-like environments created to efficiently benchmark generalization and sample efficiency of RL algorithms. We experiment the effect of training for different timesteps with different numbers of levels on test-set performance. Further, we aim to examine how different training and generalization schemes, including dropout [12] and batch normalization [6], can affect generalization of our agent and convergence of its training procedure. We aim to provide this benchmark to analyze the result on sample efficiency and generalization with different training settings, techniques and adjustments.

***K*eywords** ProcGen · Reinforcement Learning · Proximal Policy Optimiazation · Generalization

1 Introduction

MDPs (Markov decision processes) have been widely used as an effective tool to model a mathematical framework for decision making in real-world problems in control, robotics, economics, and traffic analysis. The objective in such modeling is to find optimal mapping from states to actions that maximizes a cumulative return from each state. The advance in hardware and the development in deep neural networks enabled researchers to create and investigate deep reinforcement learning (RL) algorithms with new techniques and adjustments that attempt to efficiently find optimal policies in such problems, testing their performances in benchmark environments and real-world problems across numerous applicable fields.

Among such methods, classical dynamic programming algorithms like Q-learning [16] attempt to find the optimal policy by iteratively updating the value of states over time. Mnih et al. have incorporated deep neural networks into such model (DQN) [7], and this has been followed by success of DQN models with adaptions and improvement [10, 11].

Policy gradient methods are another family of reinforcement learning algorithms that attempt to directly optimize the parametrized policy with neural network function approximators. Instead of approximating values of states with Bellman updates, these methods approximate the estimated policy gradient by enumerating trajectories since we do not know the reward network directly. By doing so, we do not approximate the Q function to infer the optimal policy, but instead optimize the policy π, directly in the policy. Compared to DQN, policy gradient methods can be applicable in

more various tasks as we can easily compute stochastic policy in continuous space, even when the complexity of the reward function is extremely high since we only use trajectories to approximate policy gradients.

From this idea, Schulman et al. proposed a method known as trust region policy optimization (TRPO) to maximize the reward along with a penalty for action distributions change by adding a KL-divergence term and using a Natural gradient [10]. Later, proximal policy optimization (PPO), which was proposed to improve previous methods by using a clipping function instead of a KL divergence [11]. Schulman et al. reported that PPO methods still has some of the benefits of TRPO has, but they are much simpler to implement, more general, and have better empirical sample complexity. In this project, as we aim to benchmark sample efficiency and generalization, we use proximal policy optimization for training the agent, since it is the state-of-the-art policy gradient based method, which is capable for more general application with high sample efficiency compared to DQN.

However, while the development of these algorithms were thoroughly examined through their performances on popular benchmark environments, such as continuous control tasks in OpenAI gym [1], one has to pay more attention to testing sample efficiency and generalization in RL algorithms. ProcGen [2], a novel collection of recently created by OpenAI, is suitable for examining such task. ProcGen consists of 16 procedurally generated environments with high diversity, fast experiment, and tunable difficulty. The main benefit of using ProcGen for benchmarking sample efficiency and generalization is that it allows us to easily tune both the number of pre-training steps and the number of different training levels across different environments provided. In this project, we aim to examine if training our agent on more levels can result in better testing time performance when the total timesteps are kept same for each trial. Also, we investigate how applying generalization schemes like dropout or batch normalization can affect the agent's generalization performace.

The main goal of this article is to provide a benchmark for sample efficiency and generalization of PPO agents in ProcGen environments, with different numbers of training levels and different generalization schemes. We also aim to review prior works related to schemes demonstrated in this article and provide a comparative study.

2 Background

2.1 Policy Gradient Methods

2.1.1 Policy Gradient Optimization

Policy gradient generally works by creating an estimator of the policy gradient based on trajectories and plugging it back into the stochastic gradient descent algorithm.

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

Here we are taking the expectation over a batch of samples, in an alternation between sampling and optimization. $\hat{A}_t$ is an estimator of the advantage function at timestep t. The estimator $\hat{g}$ can be calculated by differentiating the objective function

$$L^{PG}(\theta) = \hat{E}_t[log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2}$$

Although it is intuitive to perform multiple steps of optimization on the loss L, doing so often results in significantly large policy updates.

2.1.2 Trust Region Policy Gradient Optimization (TRPO)

In TRPO, the goal is to maximize the objective function (shown below) with a Natural gradient and a KL-divergence term, which penalizes large changes is π_θ.

$$maximize_\theta \hat{E}_t = [\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{A}_t] \tag{3}$$

$$s.t.\ \hat{E}_t[KL[\pi_{\theta o}ld(\Delta|s_t), \pi_\theta(\Delta|s_t)]] \leq \delta \tag{4}$$

where $\theta_o ld$ is the original policy parameters vector before update. This optimization problem can be approximately solved using the conjugate gradient algorithm with a linear approximation to the objective and a quadratic approximation to the constraint.

2.1.3 Proximal Policy Gradient Optimization (PPO)

proximal policy optimization (PPO) has some of the benefits of TRPO but is much simpler to implement. PPO has better sample complexity (empirically), ease of tuning, and ensures that the deviation from the previous policy is relatively small during update. Our experiments aim to show that PPO outperforms other online policy gradient methods, and overall achieves a favorable balance between sample complexity, simplicity, and efficiency.

Shown below is the PPO algorithm that uses fixed-length trajectory segments (Proximal Policy Optimization Algorithms, Schulman et al. 2017) with actor-critic cycle. Each iteration, the N parallel actors collect T timesteps of data. Then we construct the surrogate loss on these NT timesteps of data, and optimize it with minibatch SGD.

for iteration=1, 2, . . . do
—for actor=1, 2, . . . , N do
——Run policy $\pi_{\theta old}$ in environment for T timesteps
——Compute advantage estimates $\hat{A}_1, \hat{A}_2 ... \hat{A}_T$
—end for
—Optimize surrogate L wrt θ with minibatch size M $\leq$ NT
—$\theta_{old} \leftarrow \theta$
end for

2.2 Dropout

Dropout [12] is a popular method in deep learning for preventing overfitting of a model. By randomly dropping units (or setting them to zero in the forward pass) and their connections from the neural network model at training time, it attempts to prevent those units from co-adapting too much for training dataset. Alternatively, training a neural network model with this adjustment can be considered as training a ensemble of models sharing some weights/parameters. At testing time, we scale the activations such that expected outputs are the same after scaling.

3 ProcGen Benchmark

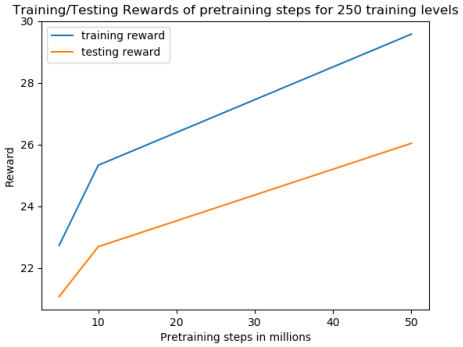
ProcGen Benchmark [2, 3] is a collection of 16 game-like environments constructed to measure sample efficiency and generalization of RL agents. These environments are procedurally generated, and its generative allows us to create distinct training sets and test sets for each environment. Also, environments in ProcGen benchmark have high diversity, fast evaluation, and tunable difficult, which make ProcGen well-suited for benchmarking sample efficiency to examine the effect of training our agent with different numbers of levels on its performance, with the total number of timesteps kept same for training of each agent. By using the easy mode, it also allows fast turnover of results in trials. In our experiment, we forked the github repo for training procgen [4] provided by OpenAI and modified the code to try different combinations of parameters and different adjustments. We used the fruibot environment in ProcGen to benchmark sample efficiency and generalization.

4 Experiments

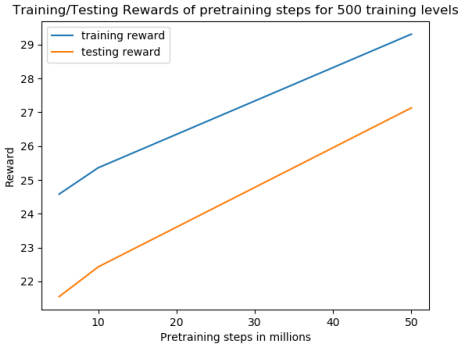
We conducted 3 main experiments, which included experimenting on how different numbers of pretraining steps, training levels, and how the addition of different layers such as dropout and batchnorm affected final testing rewards. For pretraining steps, we allowed the model to train for 5 million steps, 10 million steps, and then 50 million steps before plotting the final testing rewards to see the change over time. 4 graphs were produced, as we held the number of levels trained on, 50, 100, 250, and 500, constant. For training levels, it was done very much the same as pretraining steps except flipped around. We plotted the final testing rewards for training of 4 different training levels, 50, 100, 250, and 500, while keeping the number of pretraining steps, 5M, 10M, 50M, constant to produce 3 graphs. Testing was done by setting up an environment with a start level of 2000, which our previous number of training levels would not be able to reach, for 100 levels and was tested in conjunction with the training in ppo2's learn function. To test out different layers, we took the impala cnn model that is fed in as a network to ppo2, and added dropout layers after the reLU activation. A separate testing file was written specifically for dropout to load the trained model's parameters in after toggling the network for testing mode, as dropout needs to be removed during testing and inputting it directly into the ppo2.learn function doesn't enable this toggling. We attempted to add batchnorm as well, but ran out of time in the end as results were not matching up to expected testing rewards increase.

(a) Pretraining with 50 levels

(b) Pretraining with 100 levels



(c) Pretraining with 250 levels



(d) Pretraining with 500 levels

Figure 1: Pretraining rewards

5 Results and Analysis

For the 3 experiments, we found that pretraining steps definitely had a noticeable outcome on the final rewards. Specifically in figure 1, we can see that across all 4 ranges of training levels, increasing pretraining steps from 5M to 10M to 50M all had noticeable testing and training reward increases. In the second experiment however 2, we found that varying the number of training levels appeared to have increased rewards in the testing environment up until 250 levels trained on, however after that no clear trend has been established and training on 500 levels had around the same effect as 250. For dropout testing, we did not generate specific graphs but reported final testing reward of 14.4 in our csv log, while the max reward in the baseline model was 13.8. The baseline model was 50 training levels for 5M pretraining steps, and the dropout model was trained with these same configurations as well. Not all models were included in the github to save space.

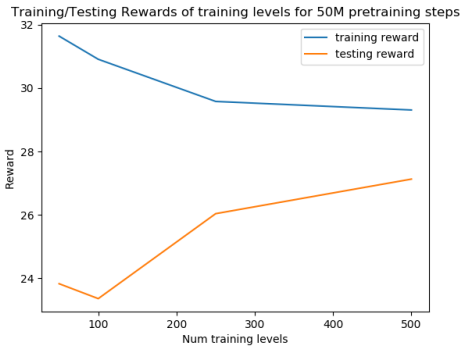
6 Conclusion and Future Improvements

In this paper, we experimented the effects of pretraining steps, number of training levels, and adding dropout by benchmarking the test-set performances of PPO agents. For pretraining steps, we saw a steady increase of testing reward up to 50M (and still increasing), whereas for number of training levels, we saw the gain flatten out after 250 levels. Implementing dropout saw very marginal increase in reward.

For further improvements, the performance of RL agents for sample efficiency and generalization should be measured in different novel benchmarks suited for such task, as there is recently an increasing number of tasks/benchmarks designed for testing an agent's generalization performance. These include Sonic benchmark [8] and the General Video Game AI framework [9]; the same experiment scheme from this project can be applied to those environments to create a benchmark for sample efficiency and generalization.

(a) Num levels with 5M steps

(b) Num levels with 10M steps



(c) Num levels with 50M steps

Figure 2: Number of Level rewards

Further research can also analyze the effect of other deep learning techniques, such as layer norm or regularization, to improve the performance of agents for sample efficiency and generalization. Testing of such task in other general benchmarks should be further investigated as well.

7 Contributions

Everyone contributed equally.

Minjune Hwang:

- Modified the training/testing scripts (based on OpenAI baseline) and ran sample experiments on GCP.
- Wrote following parts in the final report: abstract, introduction, background (Dropout), procgen, future improvements, and references.

Tony Tu:

- Trained models on GCP (step size 50, 100) for 5, 10, and 50 million steps
- Wrote following parts in the final report: background (policy gradient methods), TRPO, policy gradient, and experiments.

Richard Mao:

- Trained models (step size 250, 500) on GCP, created plots, coded dropout layer training/testing,
- Wrote following parts in the final report: result and conclusion

References

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

[2] K. Cobbe, C. Hesse, J. Hilton, J. Schulman. Leveraging procedural generation to benchmark reinforcement learning. *arXiv preprint arXiv:1912.01588*, 2019.

[3] K. Cobbe, C. Hesse, J. Hilton, J. Schulman. procgen. github repository, https://github.com/openai/procgen, 2019.

[4] K. Cobbe, C. Hesse, J. Hilton, J. Schulman. train-procgen. github repository, https://github.com/openai/train-procgen, 2019.

[5] P. Dhariwal, C. Hesse, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu. Openai baselines. github repository, https://github.com/openai/baselines, 2017.

[6] S. Ioffe and C. Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. *International Conference on Machine Learning*, 2015.

[7] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature, 518(7540)*, pp. 529–533, 2015.

[8] A. Nichol, V. Pfau, C. Hesse, O. Klimov, J. Schulman. Gotta learn fast: A new benchmark for generalization in RL. *CoRR*, 2018.

[9] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, S. M. Lucas. General video game ai: a multi-track framework for evaluating agents, games and content generation algorithms. *arXiv preprint arXiv:1802.10363*, 2018.

[10] J. Schulman, S. Levine, P. Abbeel, M. Jordan, P. Moritz. Trust region policy optimization. *International Conference on Machine Learning*, pp. 1889–1897, 2015.

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

[12] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*, 15(1):1929–1958, 2014.

[13] E. Todorov, T. Erez, Y. Tassa. MuJoCo: A physics engine for model-based control. *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, IEEE. 2012:5026–5033, 2012.

[14] H. Van Hasselt, A. Guez, D. Silver. Deep reinforcement learning with double q-learning. *Advances in Neural Information Processing Systems*, pp. 2613–2621, 2010.

[15] H. Van Hasselt, A. Guez, D. Silver. Deep reinforcement learning with double q-learning. *AAAI*, pp. 2094–2100, 2016.

[16] C.J.C.H. Watkins. Learning with delayed rewards. Ph.D. thesis, King's College, Cambridge, 1989.