CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING

Jonathan Pearce
260672004
McGill University
Montreal, QC, Canada
jonathan.pearce@mail.mcgill.ca

ABSTRACT

Deep Deterministic Policy Gradient (DDPG) is currently one of the most popular deep reinforcement learning algorithms for continuous control. Inspired by the Deep Q-network algorithm (DQN) that works with discrete action spaces, DDPG uses a replay buffer to stabilize Q-learning. It has been demonstrated that prioritized experience replay (PER) can improve the performance of DQN. We investigate whether prioritized experience replay can have a similar effect with a continuous control algorithm such as DDPG. In this project we have reproduced the DDPG algorithm, integrated prioritized experience replay with DDPG and evaluated both algorithms on two popular benchmarking tasks for continuous control methods. Our experiments show that prioritized experience replay can improve the performance of the DDPG algorithm.

1 INTRODUCTION

One of the most popular deep reinforcement learning algorithms for continuous control is Deep Deterministic Policy Gradient (Lillicrap et al., 2016). DDPG is a model-free off-policy actor-critic algorithm that combines ideas from the Deterministic Policy Gradient algorithm (Silver et al., 2014) and the Deep Q-network algorithm (Mnih et al., 2015). DDPG extends DQN by working with continuous state and action spaces with the actor-critic framework. However, similarly to DQN, DDPG utilizes an experience replay and frozen target networks to stabilize Q-learning. Like DQN, DDPG samples from the replay buffer uniformly when performing network updates. Schaul et al. (2015) utilize the Deep Q-network algorithm to introduce prioritized experience replay, a mathematically motivated replay buffer sampling strategy that improves the performance of DQN. In prioritized experience replay, experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ are given a priority according to their absolute TD error, with the idea being to sample transitions that are 'surprising' (i.e. have a large TD error) more frequently. Prioritized experience replay is not dependent on the state or action space being discrete or continuous. Therefore it can be applied to continuous control algorithms that use a replay buffer, such as DDPG. Hou et al. (2017) have shown that DDPG with ranked prioritized experience replay can produce better results than standard DDPG on the Inverted Pendulum task. In this project we implement DDPG using the details of the original paper (Lillicrap et al., 2016). We then incorporate proportional prioritized experience replay into the DDPG algorithm. We evaluate and compare these two methods using two of the standard benchmarking environments for continuous control. Our results demonstrate that the performance of DDPG on continuous control tasks can be improved with prioritized experience replay

2 METHODS

We reproduce the DDPG algorithm using the pseudocode and experimental details from the original paper (Lillicrap et al., 2016). A summary of the DDPG algorithm is in appendix C of this report. Our reproduction follows the training procedure and hyperparameters exactly as specified. The only difference is that we do not sample correlated exploration noise using the Ornstein-Uhlenbeck process. Instead, we sample uncorrelated exploration noise from the normal distribution $\mathcal{N}(0, 0.1)$.

The work of Plappert et al. (2017) demonstrated that the performance of DDPG trained using these two methods for sampling exploration noise is not significantly different, in fact they showed that even with no exploration noise DDPG can still learn successful policies. Therefore for our DDPG implementation we choose to sample exploration noise from a normal distribution. This was the only significant difference between our DDPG implementation and the DDPG algorithm described by Lillicrap et al. (2016).

For prioritized experience replay we use the proportional prioritization method from Schaul et al. (2015), where the priority of transition i in the replay buffer is defined as $p_i = |\delta_i| + \epsilon$. We also experiment with the outlined bias correction method where each transition is associated with a importance-sampling weight w. When using importance-sampling weights the the loss function that the critic minimizes in the DDPG algorithm becomes,

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} w_i (y_i - Q(s_i, a_i \mid \theta^Q))^2$$

Where $y_i = r_i - \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \theta^{\mu'}) \mid \theta^{Q'})$. The only difference between this loss function and the loss function in the original DDPG algorithm is that for each experience i in the minibatch, the square of the bellman error is weighted by the importance sampling weight w_i. After the critic is trained via gradient descent on the minibatch of experiences, each experience has its' priority updated according to its latest bellman error using the equation above.

3 EXPERIMENTS

To evaluate our two algorithms, we measure their performance on a pair of continuous control environments, Inverted Pendulum and Hopper (Erez et al., 2011). We run these environments using the open source PyBullet physics simulator (Coumans et al., 2013). PyBullet's main advantage over MuJoCo (Todorov et al., 2012) is that it is not license based and can therefore be run in cloud environments, such as Google Colab notebooks. An important note is that PyBullet's locomotion tasks (Hopper, HalfCheetah, Walker, etc.) are based on the roboschool environments (OpenAI, 2017) which are harder versions of the standard MuJoCo tasks. Therefore the results from PyBullet experiments with these tasks are not directly comparable to the same tasks run with MuJoCo. In the DDPG paper, performance statistics are only reported after at least 1 million training steps. In our experiments we only train for 100,000 steps and therefore we are not able to compare our results with the original paper regardless.

In PyBullet, the Inverted Pendulum task has a state space dimension of 5 and an action space dimension of 1. The dynamical system consists of a cart that slides on a rail, and a pole connected through an unactuated joint to the cart. The only actuator applies force on the cart along the rail The reward penalizes the angular deviation of the pole from the upright position. The pole starts each episode hanging upside-down. The Hopper task involves controlling an agent with a state space dimension of 15 and action space dimension of 3. Hopper is a 2D "robot leg" with 4 rigid links, including the torso, thigh, leg and foot. There are 3 actuators, located at the three joints connecting the links. The intended goal is to hop forward as fast as possible, while approximately maintaining the standing height, and with the smallest control input possible.

For DDPG we use the same hyperparmeters and actor and critic network structures as specified in the original paper. For prioritized experience replay we set $\epsilon =$ 1e-6 and $\alpha = 0.6$. Schaul et al. (2015) use $\alpha = 0.6$ for proportional prioritized experience replay on the Atari games, also Horgan et al. (2018) use $\alpha = 0.6$ in their experiments with continuous control tasks. Training for 100,000 time steps with regular DDPG takes approximately 90 minutes on Inverted Pendulum and 120 minutes on Hopper. Therefore an efficient implementation of prioritized experience replay was important to prevent significant delays in data collection. Schaul et al. (2015) explain how to implement proportional prioritization efficiently with the 'sum-tree' data structure. In my opinion implementing proportional prioritization that can be sampled from quickly and accurately is more related to data structure knowledge and practice then reinforcement learning. Because of this I decided to use the proportional prioritization implementation from OpenAI baselines (Dhariwal et al., 2017), I provide references to the exact code we have used in the Google Colab file linked in Section 5.

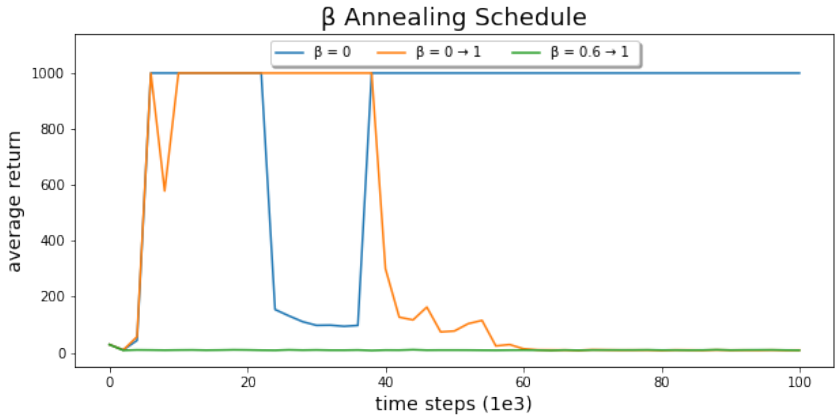

Figure 1: Hyperparameter search for β. Data collected from Inverted Pendulum task. We considered two linear annealing schedules for β, one from 0 to 1 (orange) and another from 0.6 to 1 (green). We also consider a fixed β value of 0 (blue)

For our first experiment we performed a small hyperparameter search for the optimal annealing schedule of the parameter β which is used in calculating importance sampling weights in prioritized experience replay. We considered two linear annealing schedules for β, one from 0 to 1 and another from 0.6 to 1. Schaul et al. (2015) use the schedule 0.6 to 1 with proportional prioritization on the Atari environments. We also consider the option of not using importance sampling weights (i.e. $\beta = 0$ and therefore $w = 1$). For this hyperparameter search we run DDPG with prioritized experience replay on the Inverted Pendulum environment for 100,000 time steps. We evaluate the policy every 2000 time steps, where each evaluation runs 10 episodes with no exploration noise and returns the average reward. The results of this hyperparameter search are in Figure 1. We only ran this search across 1 random seed because β fixed at 0 proved to be successful while the two annealing schedules failed.

To compare DDPG and DDPG with prioritized experience replay we utilize the Inverted Pendulum and Hopper environments. Each task is run for 100,000 time steps. We evaluate the policy every 2000 time steps, where each evaluation runs 10 episodes with no exploration noise and returns the average reward. Our results are reported over 3 random seeds and are presented in Table 1 and Figure 2 shows the learning curves for each environment.

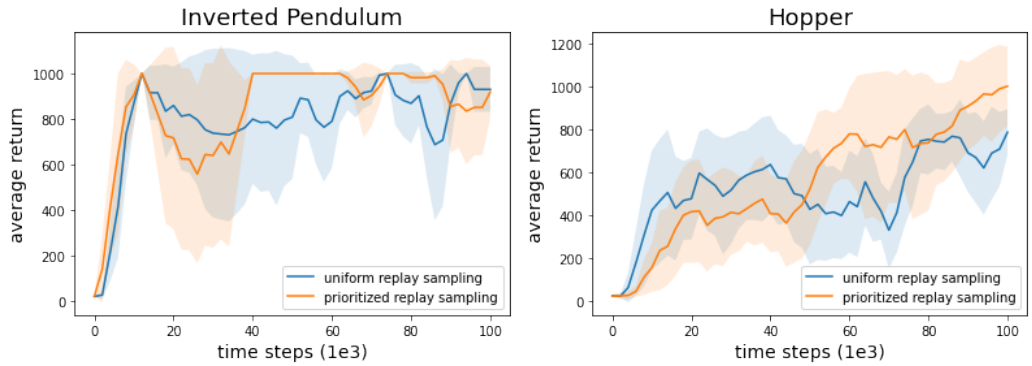

Figure 2: Comparison of DDPG with uniform replay sampling and prioritized replay sampling in the Inverted Pendulum and Hopper environments. Data is averaged over three independent runs.

Environment Name	DDPG	DDPG + PER
Inverted Pendulum	1000.0 ± 0.0	1000.0 ± 0.0
Hopper	826.9 ± 88.3	1062.4 ± 175.6

Table 1: Maximum average reward over 10 evaluation trials of 100,000 time steps. Mean and standard deviation reported from three independent runs

4 DISCUSSION

Our experiments have demonstrated that similarly to DQN, the performance of DDPG can be improved with prioritized experience replay. On the inverted pendulum task both DDPG and DDPG with prioritized experience replay were able to achieve perfect scores of 1000 reward in each of the three independent runs. Beyond 40,000 time steps DDPG with prioritized experience replay is able to obtain perfect scores more frequently and its average reward is higher than standard DDPG. The inverted pendulum environment was also helpful validating that our implementations were correct as they were both able to quickly learn and achieve a perfect score on the task. On the more complex Hopper task DDPG with prioritized experience replay also outperforms regular DDPG. Although learning slightly slower at the beginning, after 50,000 time steps DDPG with prioritized experience replay performs just as well and usually much better than DDPG with uniform experience replay. The results in Table 1 show that on the Hopper environment DDPG with PER significantly outperforms DDPG, its maximum average reward over 10 evaluation trials is more than one standard deviation above DDPG's. Even without importance sampling weights correcting the bias in the updates, prioritized experience replay still improves the performance of DDPG on multiple continuous control tasks.

Reviewing the hyperparameter search for the annealing schedule of β, we have come to realize that our choices of linear schedules were not appropriate given our experimental setup and this led to both of them failing. In the prioritized experience replay paper, the authors discuss that having unbiased updates from using importance sampling weights is most important near convergence at the end of training, this is what allows them to justify an annealing of β that goes from $\beta_0 \in [0, 1)$ to 1 over time. In most deep reinforcement learning research with continuous control, algorithms are trained on environments for at least 1 million time steps. Therefore in an experiment where DDPG is trained for 1 million time steps an appropriate linear annealing schedule would be to take β from $\beta_0 \in [0, 1)$ to 1. However, in our experiments we have only trained DDPG for 100,000 time steps. Therefore a more appropriate linear annealing schedule for our experiments in this project would be to take β from $\beta_0 \in [0, 0.1)$ to 0.1, since we are only performing a partial training procedure on DDPG. The annealing schedules we tested in our hyperparameter search annealed β too quickly.

In order to make our analysis stronger it would have been ideal to run our experiments on more locomotion tasks such as HalfCheetah or Walker2D. Training DDPG for longer would also have made our analysis more interesting. The standard deviation statistics reported in Table 1 and Figure 2 are not out of control, however running our experiments over more seeds in order to reduce the standard deviation would make our conclusions more concrete. Even with the limited scale of our experiments, the conclusion we have reached, that prioritized experience replay improves the performance of DDPG on continuous control tasks is reasonable.

We have also demonstrated that the standard benchmarking environments for continuous control algorithms can be experimented with in Google Colab notebooks by using the PyBullet physics simulator. This is a great way for people looking to run small to medium scale experiments to become familiar with these environments and be able to test their code quickly.

One interesting extension to this work would be to experiment with a much more complicated environment such as Humanoid (state space dimension of 44 and action space dimension of 17 in PyBullet) and to study how well prioritized experience replay scales with state and action space dimension. Another idea for future work would be try other methods that have been proven to increase DQN performance on DDPG, such as multi-step learning.

5 LINKS

Below are links to my code and project spotlight video.

https://colab.research.google.com/drive/1By6nzouRomRx5uz4vIoi_GH58En2v2p4

https://youtu.be/JyffR9AG8nw

REFERENCES

Erwin Coumans et al. Bullet physics library. *Open source: bulletphysics. org*, 15(49):5, 2013.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

Tom Erez, Yuval Tassa, and Emanuel Todorov. Infinite-horizon model predictive control for periodic tasks with contacts. 06 2011. doi: 10.15607/RSS.2011.VII.010.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay, 2018.

Yuenan Hou, Lifeng Liu, Qing Wei, Xudong Xu, and Chunlin Chen. A novel ddpg method with prioritized experience replay. pp. 316–321, 10 2017. doi: 10.1109/SMC.2017.8122622.

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. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1509.02971.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. URL http://dx.doi.org/10.1038/nature14236.

OpenAI. Roboschool. https://github.com/openai/roboschool, 2017.

Joelle Pineau. The machine learning reproducibility checklist (version 1.0). 2018.

Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration, 2017.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2015.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pp. I–387–I–395. JMLR.org, 2014.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. pp. 5026–5033, 10 2012. ISBN 978-1-4673-1737-5. doi: 10.1109/IROS.2012.6386109.

A REPRODUCIBILITY CHECKLIST

We follow the reproducibility checklist (Pineau, 2018) and point to relevant sections explaining them here.

For all algorithms presented, check if you include:

- **A clear description of the algorithm(s).** See Appendix C for a DDPG summary. See Method section for details about the changes made to DDPG to work with prioritized experience replay.
- **An analysis of the complexity (time, space, sample size) of the algorithm.** No complexity analysis was done.
- **A link to a downloadable source code, including all dependencies.** `https://colab.research.google.com/drive/1By6nzouRomRx5uz4vIoi_GH58En2v2p4`
- **A link to a project spotlight video.** `https://youtu.be/JyffR9AG8nw`

For any theoretical claim, check if you include:

- **A statement of the result.** This project was experimental, no new theoretical claims were made.

For all figures and tables that present empirical results, check if you include:

- **A complete description of the data collection process, including sample size.** We use standard continuous control benchmarks (Inverted Pendulum and Hopper) from Erez et al. (2011). We run these environments using the open source PyBullet physics simulator (Coumans et al., 2013).
- **A link to downloadable version of the dataset or simulation environment.** See: `https://github.com/bulletphysics/bullet3` for PyBullet physics simulator
- **An explanation of how samples were allocated for training / validation / testing** For each trial we train for 100,000 time steps, evaluating the policy every 2000 time steps. The reported evaluation statistic is the average return over 10 episodes with no exploration noise. We run 3 random seeds where the randomness stems from the initialization of the actor and critic networks and the exploration noise.
- **An explanation of any data that were excluded.** No data was excluded.
- **The range of hyper-parameters considered, method to select the best hyper-parameter configuration, and specification of all hyper-parameters used to generate results.** Hyperparameters for DDPG are the same as reported in the original paper (Lillicrap et al., 2016) except for the exploration noise, this is discussed in the Methods section. The hyperparameters for proportional prioritized experience replay follow the original work of Schaul et al. (2015), we perform a hyperparameter search over schedules for the value β, this is discussed in the Experiments section.
- **The exact number of evaluation runs.** 3 seeds for both Inverted Pendulum and Hopper. 100,000 training steps each, with testing every 2000 steps.
- **A description of how experiments were run.** See Experiments section.
- **A clear definition of the specific measure or statistics used to report results.** All reported results are average return statistics with standard deviation calculated from three random seeds.
- **Clearly defined error bars.** Standard deviation used in all cases.
- **A description of results with central tendency (e.g. mean) and variation (e.g. stddev).** We use standard deviation, see Experiments section.
- **A description of the computing infrastructure used.** We ran the experiments in a Google Colab Notebook. Inverted Pendulum takes approximately 90 minutes per random seed for 100,000 training steps. Hopper takes approximately 120 minutes per random seed for 100,000 training steps.

B REPRODUCIBILITY PROCEDURE

Note: This same information can be found in my Google Colab notebook.

To reproduce the results of my report. Open the Google Colab notebook linked in section 5. Go to the code block below the heading 'execute experiment', here you can specify the algorithm to run (DDPG or DDPG + Prioritized Experience Replay), the environment (Inverted Pendulum or Hopper) and the seed value (0,1 or 2). Then run the notebook up until the 'Data Processing' text block, this will run the selected algorithm on the specified environment for 100,000 training steps. This process will take approximetly 90 minutes for Inverted Pendulum experiments, and 120 minutes for Hopper experiments. After the code has exectued there will be a .npy file in the Google Colab 'Files' tab, the naming of the file will reflect the algorithm, environment and seed that was just run. Due to Google colab timeout concerns, download the .npy file immediately to your machine. Repeat this procedure for each algorithm and environment pair (4 combinations) with seeds 0,1 and 2 (12 total trials). After you have executed these experiments and saved the .npy files locally, upload all of the .npy files and run all the code below the 'Data Processing' header near the bottom of the notebook. This will produce the two learning curves from Figure 2 and the data in Table 1.

C DDPG SUMMARY

The first method we use in our experiments is Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2016). DDPG is a model-free off-policy actor-critic algorithm, combining DPG (Silver et al., 2014) with DQN (Mnih et al., 2015). Similarly to DQN, DDPG utilizes an experience replay and a frozen target network to stabilize the learning of the Q function. DDPG extends DQN by working with continuous state and action spaces with the actor-critic framework while learning a deterministic policy μ_θ. In order to ensure sufficient exploration, an exploration policy μ' is constructed by adding noise $\mathcal{N}$:

$$\mu'(s) = \mu_\theta(s) + \mathcal{N}$$

DDPG does soft updates on the parameters of the actor and critic target networks θ^μ and θ^Q respectively:

$$\theta' = \tau\theta + (1-\tau)\theta'$$

Where $\tau << 1$. This ensures the target network parameters change slowly. The critic Q is trained by gradient descent on the ℓ_2 loss of the Bellman error:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - Q(s_i, a_i \mid \theta^Q))^2$$

Where $y_i = r_i - \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \theta^{\mu'}) \mid \theta^{Q'})$ and N is the batch size. The actor μ is trained by gradient descent on the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a \mid \theta^Q) \mid_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s \mid \theta^\mu) \mid_{s_i}$$

C.1 ACTOR STRUCTURE

Fully connected neural network with two hidden layers. The input is the state vector. The first hidden layer has 400 units, second layer has 300 units, both layers use the ReLU activation function. The output layer produces the policy and has the number of hidden units equal to the dimension of the action space, and uses a tanh activation function to bound the actions.

C.2 CRITIC STRUCTURE

Fully connected neural network with two hidden layers. The input to the first layer is the state vector. The action vector is input to the second hidden layer. The first hidden layer has 400 units plus the dimension of the action space, second layer has 300 units, both layers use the ReLU activation function. The output layer outputs the Q value.