Deep Deterministic Policy Gradients
Teaching a robot how to walk

Author: Jules Kreuer, Roman Machacek, Jonas Müller
Contact: jules.kreuer@student.uni-tuebingen.de
Date of submission: May 08, 2023
Project available on: Github

1 Introduction

The paper of [1] proposes a deep reinforcement learning algorithm called Deep Deterministic Policy Gradient (DDPG) for continuous control tasks in robotics and games. DDPG is an actor-critic method that uses two deep neural networks to learn the actor policy and critic Q-function. It is a model-free algorithm that uses off-policy data and is able to handle high-dimensional observation spaces. The authors demonstrate the effectiveness of DDPG on a variety of simulated robotics tasks, including block stacking, cartpole balancing, and humanoid locomotion. They also compare the performance of DDPG to other state-of-the-art methods and show that it achieves better results in terms of both learning speed and final performance.
The proposed method is first presented and then expanded with two new contributions trying to improve reward gaines in the optimization process of the proposed reinforcement learning pipline.

2 Preliminaries

Consider an Markov Decision Process $< S, A, R, P >$, with state space S, action space A, reward function R and state transition function P.
Return is defined as a cumulative reward discounted by γ: $R = \sum_{t=0}^{\infty} \gamma^t r_t$, where $\gamma \in [0, 1)$.
Agent actions are modeled through policy function $\pi : S \longrightarrow A$.
Further consider functions $V_\pi : S \longrightarrow \mathbb{R}$ and $Q_\pi : S \times A \longrightarrow \mathbb{R}$ representing expected returns by following policy π.

2.1 Actor Critic Approach

The question then is, how do we find an optimal policy π^*? For this there are two main approaches: Value and Policy based [2]. In Value based methods we try to approximate an optimal value functions V_π^*, Q_π^*. Therefore we are able to evaluate the quality of the current state (V_π) or state-action pair (Q_π) based on which we can update our policy π. Typical example is Q-learning and it's neural network version DQN. Disadvantage of using these methods is that they don't work well for large and continuous spaces.
Policy based methods on the other hand try to find optimal policy directly without using approximations of value functions. We model our policy with neural network parameterised by θ. We then optimise π_θ using gradients to maximise reward function. Advantage is that we can then use the policy π_θ for continuous spaces and for modelling probability distributions over actions. Main problem with this approach is typically high variance of the policy-gradients, consequently leading to slow convergence and efficiency.
What if we could combine both approaches? By learning the value function together with policy we could for example reduce gradient variance and improve the convergence.

1. Critic: Updates value function parameters
2. Actor: Updates policy function parameters based on critic

Actor–critic methods are the natural extension of the idea of gradient-based learning methods to temporal difference (TD) learning. TD learning is an technique used to predict an expected reward value in a sequence of states. Instead of calculating total future reward it uses TD error to predict the reward. TD error δ_t describes the difference between

the optimal V^* and predicted value V at step t:

$$\delta_t = V^*(S_t) - V(S_t) \tag{1}$$

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{2}$$

Using this error we can update our value estimation by: $V(S_t) = V(S_t) + \alpha\delta_t$, where α is a hyper-parameter.
Let's now use state-action values $Q(s, a)$ and look at the reinforcement learning problem from the perspective of neural network parameter optimization.
We can use the TD error as a loss function L in order to optimize actor (μ) and critic (Q) neural networks, given by the following equation:

$$L(Q', Q, \mu', \mu) = \sum_i (y_i - Q(s_i, a_i)|\Theta^Q)^2, \tag{3}$$

where we use separate target actor (μ') and critic (Q') networks in order to get to the used target values $y_i = r_t + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$.

3 Continuous Control with Deep RL

Algorithm 1: DDPG

Randomly initialise critic network $Q(s, a|\theta^Q)$
and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialise target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$.
Initialise replay buffer R.

for episode $= 1, M$ **do**
  Initialise a random process N for action exploration.
  Receive initial observation state s_1.
  for $t = 1, T$ **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to
    the current policy and exploration noise.
    Execute action a_t and observe reward r_t and observe new state s_{t+1}.
    Store transition (s_t, a_t, r_t, s_{t+1}) in R.
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R.
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$.

    Update critic by minimising the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$.

    Update the actor policy using the sampled policy gradient:
    $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|s = s_i, a = \mu(s_i)\nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$.

    Update the target networks:
    $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$.
  end for
end for

3.1 Experiments

The authors tested the DDPG algorithm on a multitude of different reinforcement learning problems (Figure 3.1)

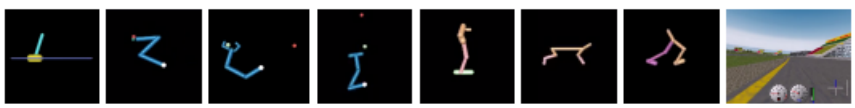

Figure 1: Example screenshots of a sample of environments we attempt to solve with DDPG. In order from the left: the cartpole swing-up task, a reaching task, a gasp and move task, a puck-hitting task, a monoped balancing task, two locomotion tasks and Torcs (driving simulator). We tackle all tasks using both low-dimensional feature vector and high-dimensional pixel inputs. Detailed descriptions of the environments are provided in the supplementary. Movies of some of the learned policies are available at `https://goo.gl/J4PIAz`.

The researchers tested their algorithm in a variety of simulated physical environments, including classic reinforcement learning tasks like cartpole and high-dimensional tasks like gripper (Figure 3.1). They ran experiments using both low-dimensional state descriptions and high-dimensional renderings of the environment. Performance was evaluated periodically during training by testing the policy without exploration noise.
DDPG's performance across all environments was measured, with results averaged over 5 replica. This method was able to learn good policies on many tasks, and in some cases, it learned policies superior to previous known methods.
The algorithm's value estimates were examined by comparing the estimated values after training with the true returns seen on test episodes. In simple tasks, DDPG estimated returns accurately without systematic biases. For harder tasks, the estimates were worse, but DDPG still learned good policies (Figure 3.1). The algorithm showed good performance on all tasks, while the variant with target network and batch normalization in the forward pass of the networks suffered from instability.

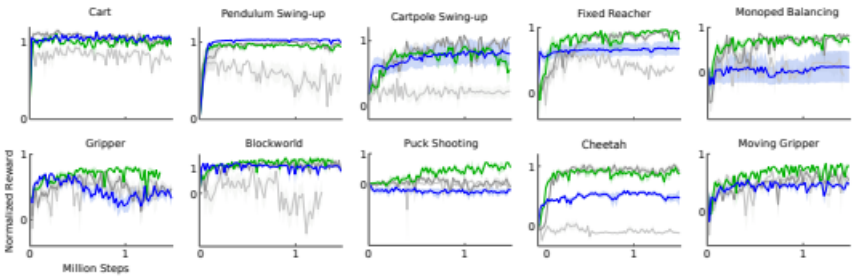

Figure 2: Performance curves for a selection of domains using variants of DPG: original DPG algorithm (minibatch NFQCA) with batch normalization (light grey), with target network (dark grey), with target networks and batch normalization (green), with target networks from pixel-only inputs (blue). Target networks are crucial.

3.2 Own Contributions

Two strategies were investigated to enhance the performance of the algorithm. Given that the algorithm samples random elements from the buffer, we sought to optimize it using two approaches. Initially, we computed a distribution across the elements in the buffer, which subsequently allowed us to sample the buffer according to the distribution weights. The weights are determined by the following equation:

$$p_i = softmax(|r_i - \mu_R|), \tag{4}$$

where $r_i \in R$ is the reward for element i in the buffer, μ_R is the mean of the rewards and $softmax$ is the softmax function with temperature parameter β given by:

$$softmax(z_i) = \frac{e^{z_i/\beta}}{\sum_{j=1}^{n} e^{z_j/\beta}}, \tag{5}$$

Therefore replay elements are sampled based on their absolute deviation from the mean of the rewards. The underlying idea is that larger deviations correspond to more informative experiences, which are then sampled with greater frequency. The β parameter controls the magnitude of the influence of the sampling process, where β controls the smoothness of the distribution as the distribution becomes more uniform with increasing β and more concentrated at different elements towards 0. Because of the sampling of the buffer with a given distribution we also maintain exploration in the training loop.

In the second method, we employed two distinct buffers derived from the complete replay buffer. The concept involves adding episodes exceeding a specific return threshold t to the "good" buffer, while those falling below are allocated to the "bad" buffer. The threshold parameter is updated dynamically according to the agent's experiences, as shown in the following equation:

$$t_i = \mu_{i,\text{roll}} + x * \sigma_{i,\text{roll}} \tag{6}$$

In this equation, the new threshold is determined by adding the mean μ_{roll} to x times the standard deviation of rewards σ_{roll} from the last 50 episodes. We utilised $x \in -0.5, 0.5$ during our experiments. This approach aims

to leverage the variability of experiences to perform upper and lower confidence bound weighted sampling of the replay buffer. During sampling, priority can be given either to trajectories from the good or from the bad buffer. This allows to tune the composition of the mini-batch from which the networks learn. We used ratios of 0.7 and 0.5 when sampling.

4 Results and Discussion

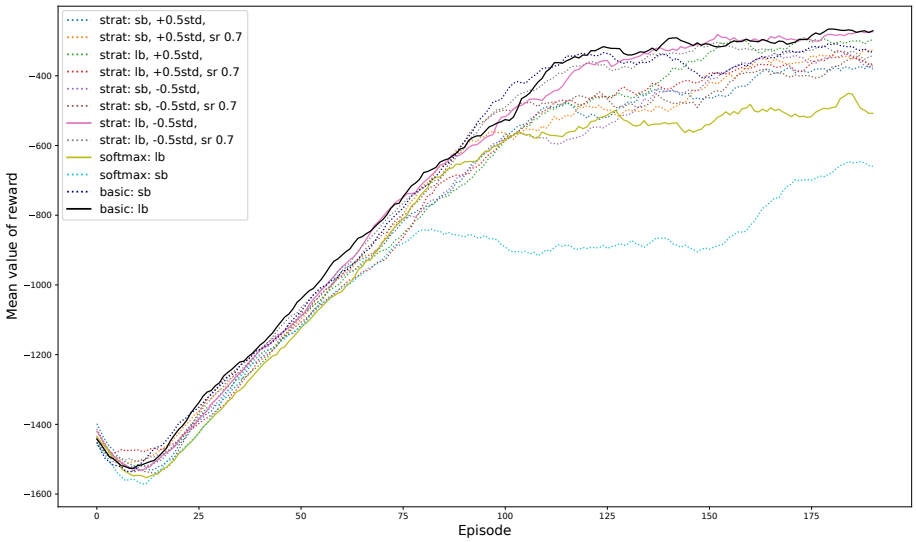

Figure 1: Average reward (30 runs, rolling average with window size 10) on Pendulum-v1, all 12 configurations. lb: large buffer (10^6), sb: small buffer (10^4), sr: sampling ratio of "good" buffer.

In our implementation, we examined two strategies to enhance the performance of the DDPG algorithm: the stratification approach and the softmax method. Both approaches aimed to optimise the sampling process from the replay buffer. However, our results were inconclusive, and we encountered several challenges during the implementation and evaluation of the methods. The reward of 30 runs on Pendulum-v1 with a moving average with window size 10 of all 12 configurations can be seen in Figure 1).

4.1 Stratification and Softmax Approach

The stratification approach carries the risk of the algorithm falling into local maxima depending on the environment's characteristics. For instance, in the BipedalWalker-v3 environment, the agent may become stuck where it remains stationary without falling, avoiding significant penalties. Such a local maxima is not present in the simpler Pendulum-v1 environment where the performance matches the baseline.
The softmax implementation proved to be computationally heavy, taking approximately twice as long to execute compared to the stratification or basic approaches. This increased complexity hindered us from exploring different hyperparameters.

4.2 Hyperparameter Optimisation

Both the stratification and softmax methods require hyperparameter optimisation, which adds complexity to the implementation. For example, the performance largely depends on the buffer size, as illustrated in Figure 1. Here, runs with larger buffer (10^6) perform in general better than runs with smaller (10^4) ones. Additionally the reward threshold and sampling ratio did impact the performance. The stratification method performed best when using a combination of a large buffer (10^6), low threshold ($x = -0.5$) and balanced sampling rate (0.5 / 0.5). This may indicate that episodes with a low reward are still important.

4.3 Inconclusive Results

Our inconclusive results can be partially attributed to the small-scale environments and low episode count we used for testing. Due to the computational limitations of a single HPC node, we were unable to thoroughly evaluate the performance of the algorithm on more complex environments.
Interestingly, similar modifications to the buffer were proposed in prioritised experience replay paper [3], where they achieved significant improvements using modified importance sampling from the buffer.

5 Conclusion

In this study, we reproduced the performance of the training pipeline presented in [1] for the bipedal walker and pendulum tasks. The employed models in addition to the training procedure can be considered as a valuable research avenue for solving continuous control problems with reinforcement learning. In addition we concluded that the proposed replay buffer sampling methods did not yield improvements. Nonetheless this conclusion shows that the randomness in buffer sampling is necessary to explore the reward space more fully and secure training success without convergence to local minima in the loss landscape.

References

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

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

[3] Y. Hou and Y. Zhang. *Improving DDPG via Prioritized Experience Replay (RL course report)*. 2019. DOI: 10.13140/RG.2.2.23694.41287.