Project Report: Project 2 - Continuous Control

Christian Michler

April 5, 2019

1 Introduction

In the first assignment of Udacity's Deep Reinforcement Learning Nanodegree [1], I studied *value-based methods* such as Deep Q Networks (DQN) [2] for training an agent to learn the value function that maps each state-action pair to a value. The optimal policy is then obtained from this action-value function estimate by taking the action for each state that has the biggest value.

Policy-based methods on the other hand directly learn the optimal policy and therefore do not have to maintain a separate value function estimate. Policy-based methods are attractive for continuous action spaces and, moreover, they can learn stochastic policies. A subclass of such policy-based methods are *policy gradient methods* that estimate the weights of an optimal policy by using gradient ascent.

Actor-Critic methods combine the respective strengths of policy-based methods and value-based methods. The actor uses a policy-based approach in which it learns to act by directly estimating the optimal policy and maximizing reward by gradient ascent. In contrast, the critic uses a value-based approach in which it learns to estimate the value of different state-action pairs. Actor-critic methods combine these respective approaches which makes them more stable than value-based methods and they require fewer training samples than policy-based methods. Deep Deterministic Policy Gradients (DDPG) [3] can be considered an actor-critic method and an extension of the DQN Method [2] to continuous action spaces.

In the first assignment [1], the setting was a *discrete* action space, i.e. an action space with a finite number of discrete action values. In contrast, in this second assignment, I consider a *continuous* action space, i.e. an action space that has an infinite number of possible action values. Using the *Reacher* Environment [1], the goal of this project is to train a Reinforcement Learning Agent to maintain the position of a double-jointed arm at the target location for as many time steps as possible. The exercise uses the Unity ML Agents Toolkit [4] to train an agent. I will use and study an agent based on DDPG [3] with various combinations of its hyperparameters.

The outline of this report is as follows: Section 2 describes the DDPG method. Section 3 first describes the problem setting of the continuous control environment in more detail. This section also contains the numerical results obtained using this environment and discusses the findings. Section 4 discusses some improvements and future work.

2 The DDPG method

Deep Deterministic Policy Gradients (DDPG) was introduced in [3]. DDPG can be considered an actor-critic method. Here, an Actor estimates the optimal policy and a Critic estimates the value of different state-action pairs. Both actor and critic use a deep neural network as nonlinear function approximator where the network weights are determined such that the neural network best approximates the respective function. In this project, the neural network is implemented in `model.py`. Both actor and critic have two hidden layers of 256 nodes[1].

[1] To be precise, Agent 2 that I am putting forward as solution has 256 nodes. Agent 1 has only 128 nodes, but below explanation of the network structure works analogously.

The actor (policy) network maps states to actions and has the following structure:

1. Input nodes (33 states)
2. First hidden layer (256 nodes) with ReLU activation
3. Batch normalization (if selected)
4. Second hidden layer (256 nodes) with ReLU activation
5. Output nodes (4 actions) with tanh activation[2]

The critic (value) network maps (state, action) pairs to Q-values and has the following structure:

1. Input nodes (33 states)
2. First hidden layer (256 nodes) with ReLU activation
3. Batch normalization (if selected)
4. Input the actions (4 nodes) at the subsequent hidden layer
5. Second hidden layer (256 nodes + 4 nodes from actions) with ReLU activation
6. Output node (1 Q-value), no activation

DDPG is an off-policy method and uses Experience Replay, i.e. rather than learning from sequential batches of experience tuples directly, these batches are stored in a replay buffer from which the agent then randomly samples. This mitigates the risk of potential correlations in a sequence of experience tuples and, moreover, enables the agent to learn from individual tuples multiple times, recall rare occurrences, and in general make better use of the generated samples. Experience Replay is implemented in `ddpg_agent.py`.

The DDPG implementation also invokes a target network besides the regular network, i.e. weights from a separate network are used in approximating the function value. While the local / regular network is updated after each time step, the target network is '*soft updated*'.

3 Problem Setting, Results and Discussion

In the Reacher environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Hence, the goal of the agent is to maintain its position at the target location for as many time steps as possible.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1. The task is episodic, and in order to solve the environment, the agent must get an average score of +30 over 100 consecutive episodes (moving average).

The implementation is based on the DDPG agent implementation of the *pendulum* example of Udacity's Deep Reinforcement Learning Nanodegree; cf. [1]. While this implementation in itself is not sufficient to solve the task of the Reacher environment, I take it as a starting point. There are a number of hyperparameters that could potentially have an effect on the agent's performance. An exhaustive exploration of this parameter space by some form of grid search is computationally not feasible within the scope of this project. After some initial exploration of various hyperparameter settings by trial and error and gaining some insight what parameters make a difference, I have taken a more systematic approach by taking the pendulum example as a base case and investigating successive amendments, working my way towards further increasing the performance of the agent until it achieves the required average score.

[2]The tanh activation serves to scale the output action values to be a number between -1 and 1.

Table 1 shows the hyperparameters considered along with the final value chosen going forward. This list of hyperparameters and additions (such as batch normalization, gradient clipping, etc.) is based on discussions / suggestions in Udacity's Student Hub and Knowledge area as well as on the papers [3, 6] The corresponding numerical experiments are given in the jupyter notebooks `Parameter_Study.ipynb` for a selected subset of the parameter study and in `Continuous_Control.ipynb` for the final settings of two agents that succeed in performing the given task.

For the hyperparameter screening / tuning in `Parameter_Study.ipynb`, I have sometimes run the different cases only for 100 episodes to get an indication of the effect a given hyperparameter has on the agent's performance. I have set max_t = 1000 so that the agent has enough time steps to sample.

The random seed should not have a fundamental effect on the agent's performance - If it had, then the agent would not be sufficiently robust and would not easily generalize. Initial trials have shown that indeed the effect of the random seed is not fundamental to achieving the given task, and I have chosen a random seed default value of 2 throughout.

Below gives an overview of the various parameters I have played with in exploring the parameter space. I usually considered some reference case and then varied a parameter up and down to assess its effect, or switching a certain feature on / off. Depending on what yielded the biggest improvement, this setting was then adopted going forward, and taken as the new reference case against which to compare variations in the next parameter.

The order of the parameters in the table is therefore with the benefit of hindsight, i.e. after running numerous computations, first on my laptop and then in the Udacity workspace for faster turn around. For the parameter study, it is instructive to start from the pendulum solution [1] as a base / reference case, and subsequently vary one parameter after the other to assess its effect on the learning; see `Parameter_Study.ipynb`. However, this approach does not capture the effect of a variation in combinations of parameters.

I now realize that the most influential parameters are the standard deviation σ of the OU noise, batch normalization and the learning rates for actor and critic.

Ornstein-Uhlenbeck noise (OU noise) serves to encourage exploration of the action space; see also [3] and [5]. OU noise has several parameters such as the long-term mean μ, the speed of mean reversion θ and the volatility σ. While I also tried out some variations in μ and θ, it seemed that it is the volatility σ that has most of an effect on successful learning.

Batch normalization was mentioned in the Student Hub and is described in [3] and [6]. Batch normalization scales the input to be within the same range by fixing the means and variances of the layer inputs. It was upon using batch normalization in combination with suitable settings for volatility σ and the learning rates for actor and critic (see `Parameter_Study.ipynb`) that I saw the agent's learning improve significantly and successfully obtain the target average score of +30 after 216 episodes (Agent 1); see Fig. 1 and case 03.b in `Parameter_Study.ipynb`.

Let's still try to gauge the effect of a number of other parameters, and see if we can further improve the learning. Clipping did not make any difference for the single agent case, which is somewhat surprising. Up to this point, I have used mini-batch sizes of 128. With a mini-batch size of 64, the agent still attains the target average score of +30, but it requires more episodes. Using a mini-batch size of 256, the agent attains the target score slightly faster than in the 128 case requiring 206 episodes instead of 216; see case 05.c in `Parameter_Study.ipynb`. Moreover, increasing the number of nodes in the hidden layers from 128 to 256, the agent learns even faster (180 instead of 206 episodes), see Fig. 2 (Agent 2), but it also requires more computation time. The implementation is given in case 06.b of `Parameter_Study.ipynb` with the values of the hyperparameters given in the last column of Table 1. I am putting Agent 2 forward as the faster-learning agent, but have included both Agents 1 and 2 in `Continuous_Control.ipynb`.

Furthermore, in Table 1, there are a number of other hyperparameters that I have also considered but not studied in great detail (so not shown in `Parameter_Study.ipynb`): `UPDATE_EVERY` and `UPDATE_M_TIMES`, i.e. how often to update the network and how many times, seem more relevant for training the 20 agent environment. For BUFFER_SIZE, WEIGHT_DECAY, TAU and GAMMA the default parameters from the pendulum example worked just fine.

Finally, from Figs. 1 and 2 it is apparent that the score curve exhibits large variations / spikes, i.e. the score occasionally drops quite a bit and then recovers. A method with less variation / more

monotone learning and, hence, more robust behaviour would be preferable; see Section 4 below for future work.

Parameter	Range	Agent 1	Agent 2
`sigma` OU noise Std. Deviation	0.3, 0.2, 0.1, 0.05	0.05	0.05
`LR_ACTOR, LR_CRITIC` learning rate actor, critic	1e-4, 1e-3 1e-4, 1e-4 3e-4, 3e-4 5e-4, 5e-4	5e-4, 5e-4	5e-4, 5e-4
`use_BatchNorm` Batch normalization	on/off	on	on
`use_clipping` gradient clipping for critic	on/off	off	off
`BATCH_SIZE` size of mini-batches	64, 128, 256	128	256
`fc_unit` #nodes in hidden layers of Actor/Critic network	128, 256	128	256
`UPDATE_EVERY` how often update network	1	1	1
`UPDATE_M_TIMES` how many times update network	1	1	1
BUFFER_SIZE Size of Replay buffer	1e5	1e5	1e5
WEIGHT_DECAY L2 weight decay	0	0	0
TAU for soft update of target parameters	1e-3	1e-3	1e-3
GAMMA discount factor	0.99	0.99	0.99

Table 1: Hyperparameters considered in tuning the DDPG Agent: Parameters that I have found most significant for improving the agent's learning (with a selected value range studied in `Parameter_Study.ipynb`) *(top)*, and parameters that I have also considered but not studied in great detail here (so not included in `Parameter_Study.ipynb`) *(bottom)*. The last two columns give the final parameter values for the Agents 1 and 2 that successfully obtain the target average score of +30 after 216 and 180 episodes, respectively; see also `Continuous_Control.ipynb` and Figs. 1 and 2.

4 Ideas for Future Work

Below I discuss some ideas for future work. To properly assess variations in different hyperparameters settings, I would do a more extensive grid search. However, this is computationally expensive. Moreover, it seems worthwhile to implement *Prioritized* Experience Replay [7] rather than the plain vanilla Replay Buffer only.

As observed in discussing Figs. 1 and 2, the score curve exhibits large variations and spikes, i.e. the score occasionally drops significantly and then recovers. A method with less variation and more monotonous learning behaviour would be preferable. To this end, I would consider investigating the performance of Distributed Distributional Deep Deterministic Policy Gradients (D4PG) [8] and Proximal Policy Optimization (PPO) [9, 10].

Finally, I would redo this exercise using the Reacher environment with 20 agents.

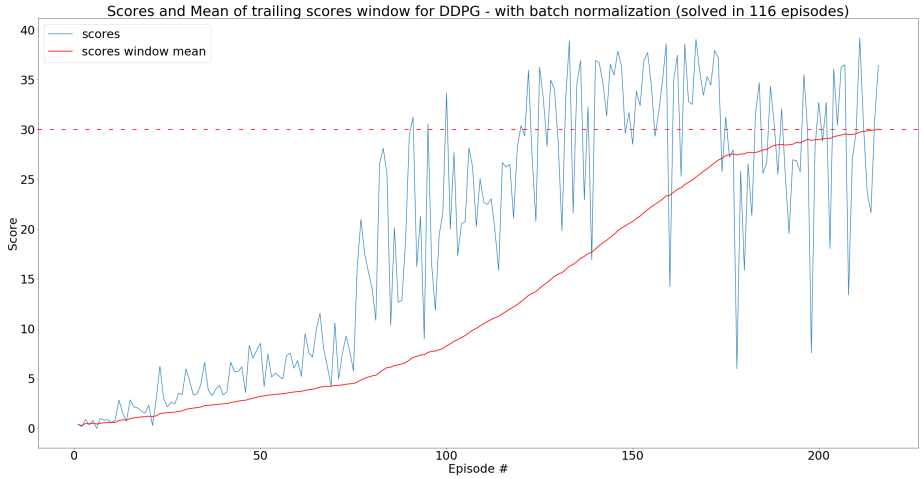

Figure 1: Agent 1: Scores and mean score over a trailing-score-window of the last 100 episodes for the DDPG Agent with parameters set according to Table 1. Achieved a trailing-score-window mean of +30.0 after 216 episodes. If solved is counted as number of episodes - 100, then this corresponds to 116 episodes as indicated in the figure.

References

[1] https://eu.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

[2] 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), 529–533, 2015. https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

[3] 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, ICLR Conference paper (2016). https://arxiv.org/pdf/1509.02971.pdf

[4] https://github.com/Unity-Technologies/ml-agents

[5] https://en.wikipedia.org/wiki/Ornstein-Uhlenbeck_process

[6] Sergey Ioffe and Christian Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift. https://arxiv.org/pdf/1502.03167.pdf

[7] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, Prioritized Experience Replay, ICLR Conference paper (2016). https://arxiv.org/pdf/1511.05952.pdf

[8] Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess and Timothy Lillicrap, Distributed Distributional Deterministic Policy Gradients, ICLR Conference paper (2018). https://arxiv.org/pdf/1804.08617.pdf

[9] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov, Proximal Policy Optimization Algorithms. https://arxiv.org/pdf/1707.06347.pdf

[10] https://openai.com/blog/openai-baselines-ppo/

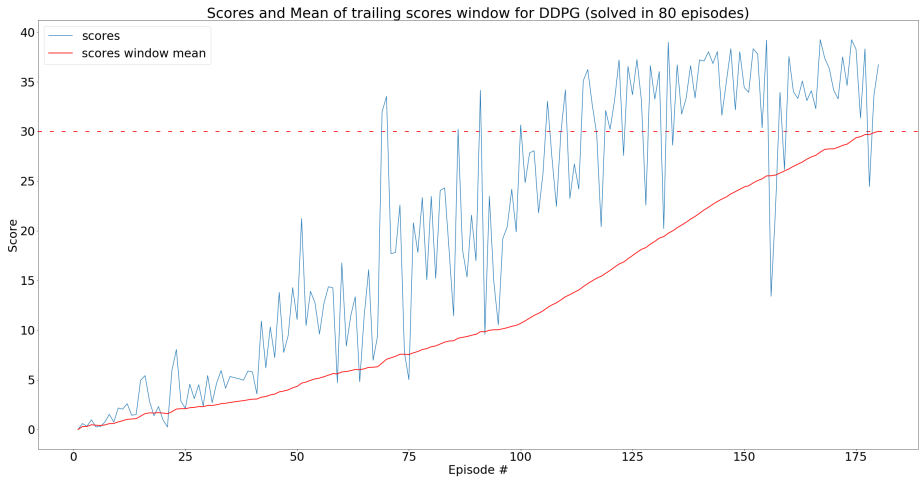

Figure 2: Agent 2: Scores and mean score over a trailing-score-window of the last 100 episodes for the DDPG Agent with parameters set according to Table 1. Achieved a trailing-score-window mean of +30.0 after 180 episodes. If solved is counted as number of episodes - 100, then this corresponds to 80 episodes as indicated in the figure.