Continuous Control with Deep Reinforcement Learning
H19 INF8225 Project

Olivier Samson[1] , **Jeremy Tschirner**[1] , **Pietro Cruciata**[1] and **Hussein Adel**[1]
[1]Département de génie informatique et génie logiciel, École Polytechnique de Montréal, Canada
olivier-2.samson@polymtl.ca, jeremy-eric.tschirner@polymtl.ca, pietro.cruciata@polymtl.ca, Hussein.Taha@polymtl.ca

Abstract

This paper presents our project for the course on artificial intelligence. Our subject covers the DDPG algorithm presented in the paper of continuous control with deep reinforcement learning [Lillicrap *et al.*, 2015]. Our implementation includes target networks for actor and critic and furthermore replay memory and proper action exploration. The evaluation of the DDPG algorithm is done on different continuous control environments including a comparison of different modifications of the algorithm. Experiments are executed ten times each and our results are shown with the average reward we got out of them. Moreover, we analyze our learning approach and suggest possible improvements on our algorithm at the end of this paper.

1 Introduction

This project aims to implement a deep deterministic policy gradient algorithm for continuous learning of tasks incorporated in the Open AI's Gym toolkit[1]. This approach aims at shortening calculations needed for continuous learning by using a deterministic policy and combining this with deep Q-learning networks (DQN algorithm).

1.1 Related Work

The concept of using reinforcement learning on a continuous action space is not new. [Baird III and Klopf, 1993] had already shown that a memory-based system combined with Q-learning could control a cart-pole balancing (or inverted pendulum) system, which could be represented by a function, with a new proposed method called *wire fitting*. This method offers a way of representing the problem's function that makes it possible to implement a reinforcement learning system in order to optimize it.

[Horiuchi *et al.*, 1996] proposes using fuzzy interpretation to adapt the Q-learning algorithm to continuous-valued pairs and continuous-valued states and action. The authors use fuzzy inference to calculate the value function, which evaluates the state/action pairs, to control, again, an inverted pendulum system. [Doya, 2000] shows that it is possible to use a continuous actor-critic algorithm to learn how to swing a pendulum up in a simulation environment. This was done by estimating the value function by minimizing the temporal difference (TD) error coupled with the use of a value-gradient-based greedy policy. [Doya *et al.*, 2002] proposes the use of a modular reinforcement learning architecture, which they call multiple model-based reinforcement learning (MMRL). With this, the authors were able to perform in both discrete and continuous tasks. Again, for demonstrating the performance in a continuous case, the swinging up of a pendulum was used.

[Pazis and Lagoudakis, 2009] proposes a method called Binary Action Search, which searches the entire action space, combined with the Least-Squares Policy Iteration and Fitted Q-Iteration reinforcement learning algorithms in order to control a continuous state-action system. The authors demonstrate their approach on the inverted pendulum, double integrator, and car on the hill systems. There are numerous works from recent years that cover the subject and it would be excessive to cite them all. In the next chapter we will take a closer look on some of them, explaining the mathematical foundations and techniques which led to the DDPG algorithm .

2 Theoretical Approach

The deep deterministic policy gradient (DDPG) algorithm is inspired by two algorithms previously developed: the deterministic policy gradient (DPG) algorithm [Silver *et al.*, 2014] and the DQN algorithm [Mnih *et al.*, 2015]. The DQN algorithm was developed to solve the problem of the unscalable action value function $Q(s,a)$ using non-linear function approximation $Q(s,a;\theta)$ through deep neural networks. Additionally, in the DQN presented in [Mnih *et al.*, 2013], the authors implemented the architecture TD-Gammon provided by [Tesauro, 1995] with a technique called *experience replay* [Lin, 1993]. This technique consists of storing the agent's experience (s_t, a_t, r_t, s_{t+1}) at each time-step in a sized cache (called *replay memory*). Then, random mini-batches are taken from the *replay memory* to select the action following a greedy policy. On the other hand, the DPG has an Actor-Critic architecture based on the policy gradient theorem [Sutton *et al.*, 1999, Peters *et al.*, 2005,

[1]https://gym.openai.com/

Bhatnagar *et al.*, 2007, Degris *et al.*, 2012]. The actor updates the policy parameters θ for the policy $\pi_\theta(a|s)$ using *gradient ascent* and the critic updates the value function parameters using an appropriate policy evaluation algorithm. The idea behind DPG is to choose an action according to a stochastic behavior (to ensure exploration) and also to learn about a deterministic target policy (exploiting the efficiency of the deterministic policy gradient) [Silver *et al.*, 2014].

As we said above the DDPG uses the actor-critic architecture, this mean that there are two networks and two set of parameters to update. In the actor, the parameters are updated with the *policy gradient* to maximize the value function J while in the critic network, the parameters are updated in order to minimize the loss function L.

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

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

As in DQN, in the DDPG algorithm there is a need to use *experience replay* to guarantee that the samples are independent and identically distributed. Moreover, in DDPG, there are target networks (as used in the DQN) used only to calculate the TD error, adapted to the actor-critic algorithm. Additionally, in DDPG, a "soft" target update is applied, meaning that the weights are not copied directly to the target networks, but are added slowly, in correspondence with a predetermined factor, here τ, which greatly improves the stability of the algorithm.

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

Moreover, in the DDPG algorithm, an Ornstein-Uhlenbeck process (Uhlenbeck & Ornstein, 1930) is used to generate the noise that is applied to the actor target to ensure exploration of the action space.

$$a_t = \mu(s_t|\theta^{Q^\mu}) + \mathcal{N}$$

This noise process is temporally correlated and can be considered as a modification of the random walk in continuous time. Algorithm 1 shows the resulting pseudocode of the DDPG procedure.

Algorithm 1 Pseudocode DDPG

```
 1: Initialize Critic $Q(s,a|\theta^Q)$, Actor $\mu(s|\theta^\mu)$
 2: Initialize target networks $Q', \mu'$
 3: for each episode do
 4:     Receive initial observation $s_1$
 5:     for each C steps do
 6:         Execute action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$
 7:         Store transition
 8:         Sample minibatch of transitions $s_i, a_i, r_i, s_{i+1}$
 9:         Set targets $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
10:         Update Critic $L = \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
11:         $\nabla_{Q^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)\, \nabla_{Q^\mu}(\mu s|\theta^\mu)$
12:         update $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
13:         update $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
14:     end for each
15: end for each
```

3 Experiments

Before using reinforcement learning methods in any real world application like robotics, it is necessary to test and evaluate them extensively. For this purpose, simulated environments can be very useful as they provide a way to validate the methods without any high expenses or dangerous situations.

3.1 Implementation Details

Our project is implemented using Python 3.7.0. Equipped with efficient matrix calculation libraries like NumPy, it yields a reasonable trade off between efficiency and complexity. The neural networks were designed using the open source deep learning framework *PyTorch* [Paszke *et al.*, 2017]. The source code of our project including a trained model for each environment can be found at https://github.com/J93T/TP4-DDPG.

Following the original paper, our agent makes use of a *replay memory* to store the interaction samples during learning. For more complex environments, it turned out to be useful to run a certain amount of warm-up steps before the actual learning to fill up the replay memory. Furthermore, we also use target networks, which turned out to improve the performance in the majority of cases. The target networks are updated using a slow moving average. We will take a closer look on their importance in the results section.

3.2 Environments

To evaluate our algorithm, we chose three different environments from the Gym toolkit. It has to be said that we only covered training on a low dimensional state representation of the environments. Learning on raw pixel input like it was originally done requires huge amounts of computational capacities and training time. Still with our approach we show that DDPG is capable of mastering a range of different control problems.

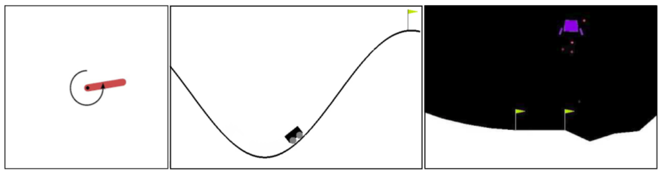
Figure 1: Gym Environments: Pendulum Swing up, Mountain Car and Lunar Lander

The first environment is a comparably simple text book problem. The goal is to swing and keep up a pendulum attached to a pivot point by applying a continuous bounded torque on it. The state $s \in [\sin(\theta), \cos(\theta), \dot{\theta}]$ encodes information about the pendulum's angle and angular velocity

Table 1: DDPG Hyperparameter for three different environments

Hyperparameter	**Pendulum Swingup**	**Mountain Car**	**Lunar Lander**
Critic Units	200/100	128/64	256/128
Actor Units	200/100	128/64	256/128
Discount factor γ	0.95	0.99	0.995
Batch Size	64	128	128
Learning Rate Critic	1e-3	1e-3	1e-3
Learning Rate Actor	1e-4	1e-4	1e-4
Target Update τ	1e-2	0.005	1e-2
Max Buffer Size	1e6	1e6	1e6

while the reward function penalizes the deviation from the desired angle $\theta_{des} = 0$ and angular velocity $\dot{\theta}_{des} = 0$.

The second environment is somewhat more challenging. Its goal is to drive an underpowered car up a hill from a lower position. Due to the constraint on the cars power, it is not possible to reach the goal directly by just going forward. The trick is rather to gain potential energy by climbing up the other hill and use this momentum to reach the goal. However, this is a scenario with sparse rewards, as the agent constantly loses a certain amount proportional to its energy consumption. Only by reaching a goal it receives the final positive reward. To solve the problem, the algorithm needs a good exploration behavior to prevent getting stuck in the local optimum of just doing nothing.

The last environment we evaluated on is the most complex. Its state space has eight dimensions which describe the position, velocity and rotation of a space ship in two dimensions. The goal is to land the ship in a desired target zone by applying thrust in one of three possible directions. The reward function is complex and considers fuel consumption as well as a successful landing. The difficulty with this environment is that the agent can easily get stuck in a local optimum where it learned to hover without landing. Figure 1 shows all three different environments used in our experiments.

3.3 Hyperparameter

Although deep reinforcement learning methods strive to be robust towards different environments and setups, the choice of hyperparameters can be a crucial factor for their success. Following our intention of reproducing the results achieved in the original paper we tried to keep our hyperparameters as close as possible to the original ones. However, some modifications were inevitable due to constraints on computing resources and the fact that we did not train the agent on raw pixel inputs.

Table 1 shows our choice of hyperparameters for different environments. The number of hidden neurons is shown for both first and second hidden layer of the networks. For the activation, we used ReLU functions and a tanh output layer for the actor to bound the actions. The network optimization was done using the state of the art optimizer Adam [Kingma and Ba, 2015] using adaptive moment estimation with default settings.

It is noticeable that our parameter τ, which controls the target networks update speed, is larger than in the original paper. For our experiments, this choice turned out to give better results. Besides that, the hyperparameter are similar to the original ones except for slight changes in the number of hidden neurons.

3.4 Results

We evaluated the DDPG algorithm on the three different continuous task environments already mentioned above. The results are shown in figure 2, figure 3 and 4. The training was done over 200 episodes for the first two environments and 300 episodes for the more complex Lunar Lander. To get a sense of the algorithms reliability, we averaged the results over ten completely independent experiments. For all plots, the thick line shows the corresponding mean, while the colored area displays the 95% confidence interval around it.

For each environment we trained three different versions of the algorithm, one without target networks, one with target networks and the last with both target networks and batch normalization, as proposed in the original paper. The reward was taken directly during training where the agent was following its explorational policy. To be able to compare the different environments, we normalized the reward between the score of a random agent (0) and a perfect planning algorithm (1).

We can see that throughout all environments the agent reached a near optimal policy at the end. The DDPG version with target network (orange) and target network with batch normalization (green) seems to achieve higher mean rewards and more stability relative to the DDPG without target networks (blue). This is particularly the case for the *Inverted Pendulum* and the *Lunar Lander* environment. For the *Mountain Car*, it seems that both target network and batch normalization did not have a considerable impact. This might be related to the fact that the environment only provides a sparse reward. If the agent finds the goal through exploration once, it often directly applied the right policy again for the rest of the episodes. In figure 4 we can see this phenomenon, because the learning process seems not to be as smooth ascending as in the other two environments.

A big difference between the environments is their complexity. The plots show that the agent takes more training episodes for the more complex environment, *Lunar Lander*, to derive to a reasonable policy. Also this deviation seems to be larger. For the *Inverted Pendulum* in figure 2, it is noticeable that the average reward is stabilizing after only 80 episodes using the target networks. This difference is mainly caused on the different dimensions of observation and action space respectively. The *Lunar Lander* has a an eight dimensional state space and a four dimensional action space while the *Inverted Pendulum* only allows one action while observing a three dimensional state space.

For the *Mountain Car Continuous* environment shown in figure 4, the rewards of the DDPG algorithm with and without target network and batch normalization seem to be nearly equal with a bit of fluctuation throughout the

Table 2: DDPG normalized performance for 10 experiments each training 200 episodes on the Pendulum Swingup and Mountain car and 300 episodes on Lunar Lander

Reward	**Pendulum Swingup**	**Mountain Car**	**Lunar Lander**
Max (T+BN)	1.07	0.98	1.02
Max (T)	0.99	0.97	1.01
Max	0.80	0.97	0.99
Avg (T+BN)	0.64	0.73	0.65
Avg (T)	0.62	0.71	0.63
Avg	0.38	0.63	0.35
Min (T+BN)	-0.39	-0.21	-0.84
Min (T)	-0.39	-0.23	-0.83
Min	-0.41	-0.23	-8.25

episodes.

Finally, table 2 shows a summary of the performance for all three different DDPG variants. We selected the maximum, average and minimum normalized reward over 10 training experiments. These results confirm the impression that target networks are important in most cases. However, for our comparably low dimensional environments, batch normalization did not have a strong impact.

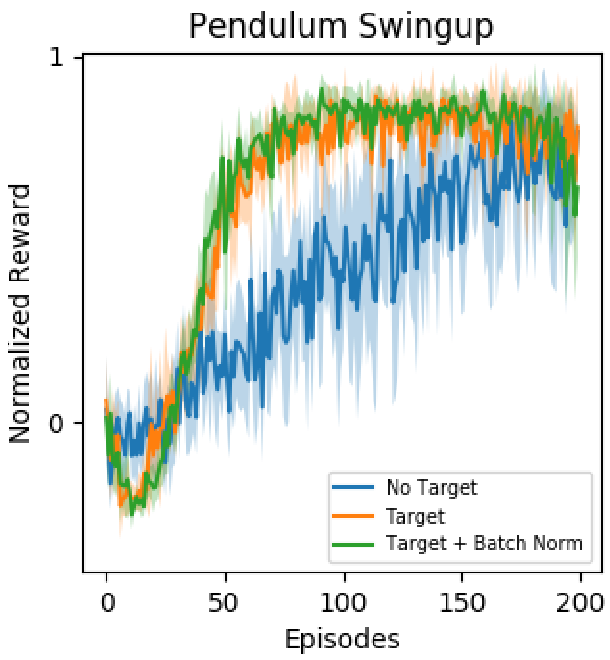

Figure 2: Swinging-Up Pendulum Results for three different variants of DDPG

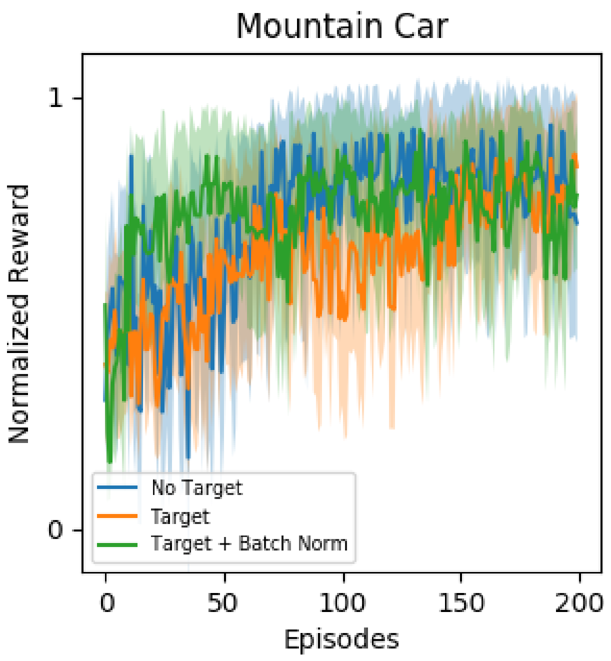

Figure 3: Mountain-Car Continuous Results for three different variants of DDPG

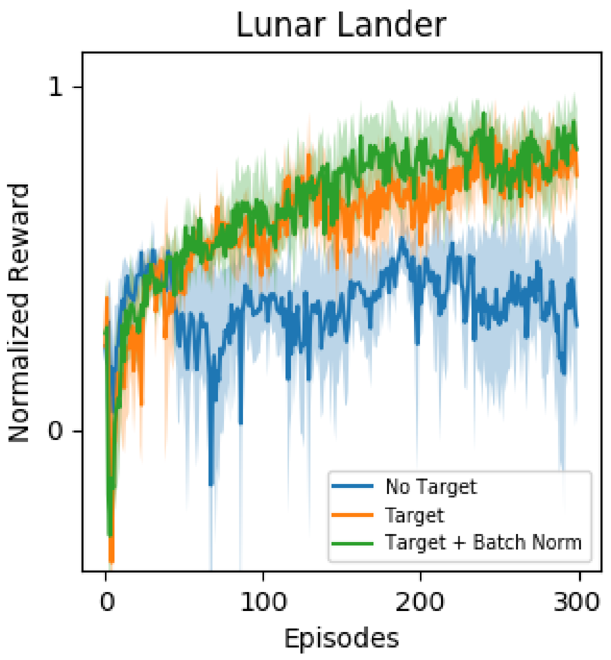

Figure 4: Continuous Lunar Lander Results for three different variants of DDPG

4 Learning Approach Analysis

As this project represents a practical implementation of our presentation topic, we already had most of the necessary theoretical background knowledge when starting the implementation. The algorithm itself in general is straight forward and rather easy to follow. To really grasp the idea behind the policy gradient we also used different sources from the internet and tried to understand the derivation of the respective theorem to get a better understanding of its use.

The next step was the actual implementation. As we all, to this point, were not proficient with PyTorch, we first implemented numerous small tutorials and samples to get used to it.

As in every research, the part that was more time consuming in the project was researching adequate sources and documentation for our subject. The fact that, on the Internet, there are plenty of tutorials and explanations is useful, but, at the same time, you can lose a lot of time looking at the different point of views. Being able to select the right sources was a main factor in our research. Also we take many experiences to stabilize our learning code by adding random noise, target network and doing many experiments for each environment.As above mentioned, implementing the algorithm on a smaller scale before getting to the more complicated environments represented another key point in our understanding of DDPG with *PyTorch*.

5 Conclusion

Our project is an implementation of the DDPG algorithm. We showed that it is capable of solving continuous control tasks, such as the *InvertedPendulum*, *MountainCarContinuous* and *LunarLanderContinuous* from the Open AI's gym toolkit. To achieve that, our algorithm makes use of *replay memory* and target networks for both actor and critic. In most cases, the implementation of the target networks showed an amelioration of both stability and results. Furthermore we showed that batch normalization can help stabilize the learning and give the algorithm the capability to handle differently scaled state spaces in a robust way. However, both of these aren't as crucial as initially thought, as shown by our test results. Proper action exploration was always ensured using a random noise process.

On the other hand, really high dimensional environments such as locomotion tasks turned out to be really complex and time consuming to evaluate so we could not show any results in this area. It can still be evaluated with our project code, but a high end PC or a high performance cluster is required.

There are still many things that can be improved in our implementation. We do not train our networks on raw pixel data like [Lillicrap *et al.*, 2015] did, even though it showed better results. It is also possible to run the actor and critic updates in parallel with distributed actors for a faster learning rate [Sogabe *et al.*, 2019, Espeholt *et al.*, 2018, Adamski *et al.*, 2018]. Adding noise in the parameter space instead of the action space can also enhance the actor's performances and enable it to give the same result for a given state [Plappert *et al.*, 2017]. In addition, a better management of the replay buffer could improve performances performance in our tasks [de Bruin *et al.*, 2018].

References

[Adamski *et al.*, 2018] Igor Adamski, Robert Adamski, Tomasz Grel, Adam Jedrych, Kamil Kaczmarek, and Henryk Michalewski. Distributed deep reinforcement learning: Learn how to play atari games in 21 minutes. *CoRR*, abs/1801.02852, 2018.

[Baird III and Klopf, 1993] Leemon C. Baird III and A Harry Klopf. Reinforcement learning with high-dimensional, continuous actions, Nov. 1993.

[Bhatnagar *et al.*, 2007] Shalabh Bhatnagar, Richard S. Sutton, Mohammad Ghavamzadeh, and Mark Lee. Incremental natural actor-critic algorithms. In John C. Platt, Daphne Koller, Yoram Singer, and Sam T. Roweis, editors, *NNeural Information Processing SystemsIPS*, pages 105–112. Curran Associates, Inc., 2007.

[de Bruin *et al.*, 2018] Tim de Bruin, Jens Kober, Karl Tuyls, and Robert Babuška. Experience selection in deep reinforcement learning for control. *Journal of Machine Learning Research*, 19(9):1–56, 2018.

[Degris *et al.*, 2012] Thomas Degris, Patrick M. Pilarski, and Richard S. Sutton. Model-free reinforcement learning with continuous action in practice. In *American Control Conference*, pages 2177–2182. IEEE, 2012.

[Doya *et al.*, 2002] Kenji Doya, Kazuyuki Samejima, Kenichi Katagiri, and Mitsuo Kawato. Multiple model-based reinforcement learning. *Neural Computation*, 14(6):1347–1369, 2002.

[Doya, 2000] Kenji Doya. Reinforcement learning in continuous time and space. *Neural Computation*, 12(1):219–245, 2000.

[Espeholt *et al.*, 2018] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1407–1416, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[Horiuchi *et al.*, 1996] T. Horiuchi, A. Fujino, O. Katai, and T. Sawaragi. Fuzzy interpolation-based q-learning with continuous states and actions. In *Proceedings of IEEE 5th International Fuzzy Systems*, volume 1, pages 594–600 vol.1, Sep. 1996.

[Kingma and Ba, 2015] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[Lillicrap *et al.*, 2015] 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. *arXiv e-prints*, arXiv:1509.02971, Sep. 2015.

[Lin, 1993] Long J. Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, CMU, Pittsburg, 1993.

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

[Mnih *et al.*, 2015] 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, February 2015.

[Paszke *et al.*, 2017] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[Pazis and Lagoudakis, 2009] Jason Pazis and Michail G. Lagoudakis. Binary action search for learning continuous-action control policies. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 793–800, 2009.

[Peters *et al.*, 2005] J. Peters, S. Vijayakumar, and S. Schaal. Natural actor-critic. In *Proceedings of the 16th European Conference on Machine Learning*, pages 280–291, 2005.

[Plappert *et al.*, 2017] 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. *CoRR*, abs/1706.01905, 2017.

[Silver *et al.*, 2014] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin A. Riedmiller. Deterministic policy gradient algorithms. In *ICML*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 387–395. JMLR.org, 2014.

[Sogabe *et al.*, 2019] Tomah Sogabe, Malla Bahadur, Katsuyohi Sakamoto, Masaru Sogabe, Koichi Yamaguchi, and Shinji Yokogawa. Hybrid policy gradient for deep reinforcement learning. *Bulletin of Networking, Computing, Systems, and Software*, 8(1), 2019.

[Sutton *et al.*, 1999] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller, editors, *Neural Information Processing Systems*, pages 1057–1063. The MIT Press, 1999.

[Tesauro, 1995] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.