Learn to Move Through a Combination of Policy Gradient Algorithms: DDPG, D4PG, and TD3

Nicolas Bach*, Andrew Melnik*, Malte Schilling, Timo Korthals, and Helge Ritter

CITEC, Bielefeld University, Bielefeld, Germany
`nbach@techfak.uni-bielefeld.de`
`andrew.melnik.papers@gmail.com`

Abstract. Deep Reinforcement Learning has recently seen progress for continuous control tasks, driven by yearly challenges such as the *NeurIPS Competition Track*. This work combines complementary characteristics of two current state of the art methods, Twin-Delayed Deep Deterministic Policy Gradient and Distributed Distributional Deep Deterministic Policy Gradient, and applied this in the state-of-the-art *Learn to move — Walk Around* locomotion control challenge which was part of the *NeurIPS 2019 Competition Track*. The combined approach showed improved results and achieved the 4th place in this competition. The article presents this combination and evaluates the performance.

1 Introduction

The *NeurIPS 2019: Learn to Move — Walk Around*[1] challenge [1,2] poses a continuous control task for a physiologically plausible 3D walking agent in the physics-based OpenSim environment [3] that is to be controlled by activation of muscle fibers attached to the agent. The agent is supposed to follow a prescribed 2D velocity vector. The task became incrementally harder compared to the previous *NeurIPS 2018: AI for Prosthetics* challenge, in which the provided 1D velocity vector had always the same direction and only the absolute value was changing.

We solved the task by combining Twin-delayed Deep Deterministic Policy Gradient (TD3) [4] and Distributed Distributional Deep Deterministic Policy Gradient (D4PG) [5] algorithms. Both algorithms are extensions of the Deep Deterministic Policy Gradient (DDPG) [6] and implement several improvements (see table 1). This solution showed to score an improvement compared to the two algorithms individually and scored fourth place out of 310 teams in this competition. In this paper, we evaluate the feasibility and performance of combining these improvements and compare it to the performance of the two original algorithms in the *NeurIPS 2019: Learn to Move — Walk Around* challenge. The combined algorithm is tested against its components, TD3 and D4PG, in two

[1] `https://www.aicrowd.com/challenges/neurips-2019-learn-to-move-walk-around`
* Equal contribution.

experiments. Other top ranked solutions for this and previous years challenge variants are described in [7,8,5,2]. Deep Reinforcement Learning methods has been successfully applied in an increasing number of areas, ranging from computer games towards robotic control [9,10,11,12,13,14,15].

2 Methods

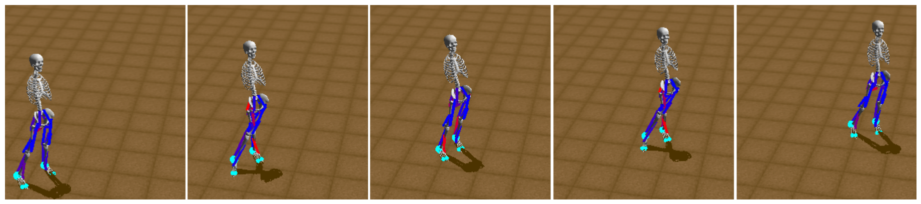

Fig. 1. The task of the competition: Developing a controller capable of locomotion for the skeleton, which can only be controlled via activation of its muscles on its legs. The figure shows a movement of the agent in a sequence of five time steps. Active muscles are shown in red, inactive muscles are shown in blue.

2.1 Combination of algorithms

Our algorithm is based on DDPG and combines all improvements (see table 1 for an overview) introduced by TD3 and D4PG. The implementation of TD3 and D4PG improvements is mostly straightforward (compartmentalization of both algorithms can be seen in algorithm 1; red highlights D4PG and green TD3). The improvements themselves do not intertwine with each other, except for clipped double Q-Learning of TD3 and the distributional value function of D4PG (in algorithm 1 computing the Q-value with twin critics and choosing their minimum is part of the distributional update steps). The combined algorithm should provide a more stable learning signal, while offering the same scalability as found in D4PG. Overall, we assume that it should lead to an improvement, first, compared to TD3 as the sample efficiency is better. Second, in comparison with D4PG, as improvements to stabilize the learning process are deployed.

2.2 Comparison of Deterministic Policy Gradient algorithms

In the following, we describe experiments that demonstrate the viability of the combination of deterministic policy gradient algorithms for the *NeurIPS 2019: Learn to Move — Walk Around* challenge. We compare the combined approach to its' combinational parts, TD3 and D4PG. Further, we elaborate on the details regarding the challenge, especially the reward function, which had to be optimized, as this appeared detrimental to a reinforcement learning problem.

Algorithm 1 Combination of TD3 (green) and D4PG (red)

Input: batch size M, trajectory length N, number of actors K, replay size R, exploration constant ϵ, initial learning rates α_0 and β_0

```
 1: Initialize critic networks $Q_{\theta_1}$, $Q_{\theta_2}$ and actor network $\pi_\phi$ replicating network weights
    to each of the K actors with random network weights $\theta_1$, $\theta_2$, $\phi$
 2: Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
 3: for $t = 1, ..., T$ do
 4:     Sample mini-batch of $M$ transitions $(\mathrm{x}_{i:i+N}$, $\mathrm{a}_{i:i+N-1}$, $r_{i:i+N-1})$
        of length $N$ for replay buffer with priority $p_i$
 5:     Generate target action $\tilde{a} \leftarrow \pi'_\phi(s_{i:i+N-1}) + \epsilon, \quad \epsilon \sim clip(\mathcal{N}(0, \tilde{\delta}), -c, c)$
 6:     Construct target distributions
            $Y_i = (\sum_{n=0}^{N-1} \gamma^n r_{i+n}) + \gamma^N min_{j=1,2} Z^j_{w'}(\mathbf{x}_{i+N}, \pi_{\theta'}(\mathbf{x}_{i+N}))$
        for critics j = 1,2
 7:     Compute the actor and critics updates
            $\delta_{\theta_i} = \frac{1}{M} \sum_i \nabla_w (Rp_i)^{-1} d(Y_i, argmin_{Z_j} Z^j_w(\mathbf{x}_i, \mathbf{a}_i))\Big|_{j=1,2}$
            $\delta_\phi = \frac{1}{M} \sum_i \nabla_\theta \pi_\theta(\mathbf{x}_i) \mathbb{E}[\nabla_{\mathbf{a}} argmin_{Z_j} Z^j_w(\mathbf{x}_i, \mathbf{a})]\Big|_{\mathbf{a}=\pi_\theta(\mathbf{x}_i), j=1,2}$
 8:     Update critics $\theta_i \leftarrow \theta_i + \beta_t(\delta_{\theta_i})$
 9:     if $t$ mod $d$ then
10:         Update $\phi \leftarrow \phi + \alpha_t \delta_\phi$ and replicate network weights to the actors
11:         Update target networks $\theta'_i$ and $\phi'$
12:     end if
13: end for
14: return: policy parameters $\phi$
```

Actor

```
1: repeat
2:     Sample action $\mathbf{a} = \pi_\theta(\mathbf{x}) + \epsilon\mathcal{N}(0, 1)$
3:     Execute action $\mathbf{a}$, observe reward $r$ and state $\mathbf{x}'$
4:     Store $(\mathbf{x}, \mathbf{a}, r, \mathbf{x}')$
5: until learner finishes
```

Table 1. List of components of the deterministic policy gradients. A detailed description of each component can be found in the appendix. The components of our combined approach are shown in column four.

	Approach			
Component	DDPG	TD3	D4PG	**Ours**
Deterministic policy gradient	✓	✓	✓	✓
Target policy and value networks	✓	✓	✓	✓
Explorative noise	✓	✓	✓	✓
Experience replay buffer	✓	✓		
Clipped Double Q-Learning		✓		✓
Delayed update of policy networks		✓		✓
Target policy smoothing		✓		✓
Multiple sampler			✓	✓
Distributional critic			✓	✓
N-step returns			✓	✓
Prioritized experience replay buffer			✓	✓

The same configuration for all three algorithms was used to evaluate their characteristics. Actor and critic neural networks (TD3 and our algorithm operated with a pair of critics) were given three hidden layers with sizes $(512, 512, 256)$ in all three cases. We used Gaussian noise for exploration. Further, for each algorithm we deployed a trainer thread to perform update steps and 22 sampler threads to produce samples in parallel, which were necessary for the updates, and store them in a shared replay buffer. Although originally TD3 has no distributed training framework, we extended the algorithm for better evaluation of the other improvements. A sampler is a copy of the policy network, which acts on the environment, whereas the trainer contains the algorithm, which optimizes the policy and value function, and copies the weights of the policy functions to the sampler threads every 500 update steps.

The implementation of prioritized experience replay [16] that is employed by D4PG and our approach uses the absolute TD-errors as sample-weighing-strategy and produces batches dependant on those weights, favoring more important samples. TD3 was implemented with a regular replay buffer, which is sampled uniformly. For realizing the distributional critic used in D4PG and our algorithm, we used a quantile distribution [17] consisting of 101 atoms. N-step returns were set to five. Delayed updates of policy networks were executed in a ratio of two critic updates for every actor update. For target policy smoothing we used Gaussian noise.

2.3 OpenSim environment

The *NeurIPS 2019: Learn to Move — Walk Around* challenge poses the task to control a physiologically plausible 3D walking agent in the physics-based OpenSim environment [3] only by activation of muscle fibers attached to the

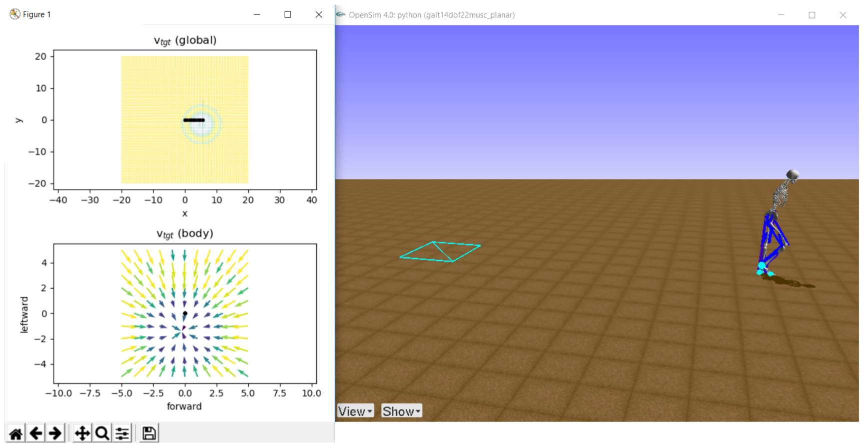

Fig. 2. The competition's environment. Based on OpenSim it provides a 3D environment, in which the agent should be controlled, and a velocity field to determine the trajectory the agent should follow. (source: [18])

agent. The activation range of the muscles spans the continuous space between 0 and 1. The agent has 22 muscles distributed over its lower body, so the action space amounted to twenty-two dimension. The agent is supposed to follow a provided 2D velocity vector field. This vector field V is a 2 x 11 x 11 tensor of 2D velocities in forward and leftward direction of the agent. It spans a 11 x 11 grid within 5 meters around the agent with the agent at its center. The distance between each discrete point in the grid amounts to 0.5 meters (as can be seen in figure 2). The vector field is one part of the observation space the agent could access. The second part of the provided observation space is a dictionary of 97 observations for pelvis state, ground reaction forces, joint angles, and velocities, as well as muscle states, such as their length. Therefore, the accessible observation space amounts to 341 dimensions. Our solution took only into account the actual target velocity in the agents position, as well as the difference between target velocity and real velocity, resulting in an observation space of 103 dimensions for our agent.

The environment provided two different reward functions for round one and two of the *NeurIPS 2019: Learn to Move — Walk Around* on which the agent was optimized. We used the reward function of the second round to conduct the experiments described in section 3, which was provided by the competition's environment[2]. It was not shaped in any form. The environment returned reward in each step (dense reward). The total reward $J(\pi)$ is described as a sum of reward for staying alive and reward for performing footsteps, where the latter was defined as bridging a minimum distance between contact with the ground, while traveling in the right direction and using minimal effort in terms of muscle

[2] `https://github.com/stanfordnmbl/osim-rl`

activation. The maximum number of steps in the first round was set to 1000 and in round two to 2500 steps per episode.

$$J(\pi) = R_{alive} + R_{step} \tag{1}$$

$$= \sum_{i} r_{alive} + \sum_{step_i} (w_{step} r_{step} - w_{vel} c_{vel} - w_{effort} c_{effort}) \tag{2}$$

In equation (2) w_{step}, w_{vel} and w_{effort} refers to a constant weight of the stepping reward as well as to the weights for effort and velocity costs. The costs and rewards are defined as

$$r_{alive} = 0.1 \tag{3}$$

$$r_{step} = \sum_{i \ in \ step_i} \Delta t_i = \Delta t_{step_i} \tag{4}$$

$$c_{vel} = || \sum_{i \ in \ step_i} (v_{pelvis} - v_{vectorfield}) \Delta t_{step_i} || \tag{5}$$

$$c_{effort} = || \sum_{i \ in \ step_i} \sum_{m}^{muscles} A_m^2 \Delta t_{step_i} || \tag{6}$$

with $\Delta t_i = 0.01$ seconds is the simulation time step. v_{pelvis} and $v_{vectorfield}$ are the velocity of the pelvis and the target velocity and A_m are muscle activations.

A bonus of 500 was given by successfully standing near the target for a time period (two to four seconds). This bonus could be achieved twice, as after achieving the first reward a new velocity field spawned, which the agent also needed to solve to successfully end the episode.

3 Experiments

In the first experiment, we ran a test agent with no exploration noise every 500 updates of policy weights and collected reward values for that episode. For each algorithm, we repeated the training process with a different seed three times. The average result of three tests for each algorithm is plotted in figure 3. One repetition of the training process amounted to 1,000,000 update steps of the trainer thread. As described above, during training the episode length was constrained to 1000 steps per episode. This limits the reward, which could be achieved, by the number of steps.

For the second experiment, we tested the performance of the trained networks from the first experiment. We ran each of the 9 agents (3 algorithms x 3 seeds) on the same 50 episodes after the absolved training process (1,000,000 updates steps). The maximum length of an episode was set to 2500 steps. As mentioned above, this was established in the second round of the learning to run challenge, where the agent's task was to solve two velocity fields in one episode. Thereby, the reward was limited. The networks were compared in terms of reward earned and steps taken.

3.1 Results of the experiments

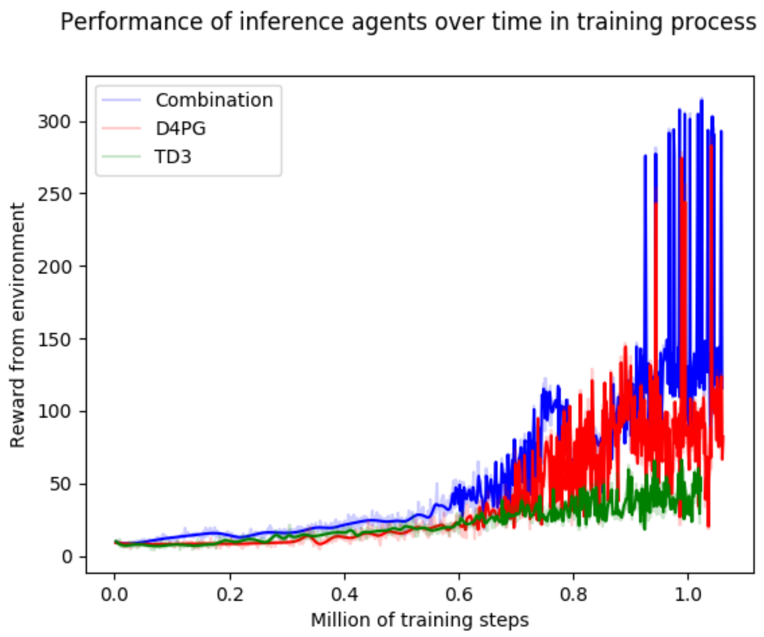

Fig. 3. Results of experiment 1. In later stages of training the agent was able to achieve a bonus reward of 500 by standing at its target for multiple seconds, resulting in spikes in the later stages of training. However, the rewards of the episodes only amount to about 350 in average because of their occurrence rate and difference in time during training runs.

The results for experiment 1, which are depicted in figure 3, show that a combination of algorithm converges faster to a well-performing policy, enabling the agent to achieve the second round's reward of 500. Due to restricted episode length of 1000 during training the agent was not able to solve whole episodes of the environment (default settings for difficulty 2 of the environment are 2500 steps per episode). Further, the reward is maxed out by around 350 for an episode. This relates to the spikes in figure 3. The agent was not able to achieve the second bonus, as the episode length was constrained to 1000 steps per episode during training. The spikes amount to a reward of around 350 as they are averaged over episodes, where the agent did not achieve bonus due to different occurrence rates in all seeds. Moreover, we can also observe a smoother trend of the curve until convergence (around 600,000 steps) for the combination of algorithms in comparison to D4PG. In later stages of the training process we could also observe, that our approach has less low-reward outliers as D4PG. TD3 was not able to produce a policy, which was able to score more than a reward of around 50 and therefore wasn't able to produce any high-reward outliers by scoring the bonus reward. D4PG was able to score the round two bonus, but at a later stage of the training than our proposed algorithm

In experiment 2 (see table 2), we observed similar results. The trained policy of our combined approach was able to outperform both TD3 and D4PG. After the finished training, the TD3 algorithm scored worse compared to D4PG and our approach in terms of average steps and reward. Our proposed combined approach was able to produce a policy, which scores about 30% higher on average than D4PG in terms of average reward and average steps taken in episodes. We were also able to decrease the standard deviation in reward by about 20% and in steps by about 15%, which implies less proneness to the failure mode.

Table 2. Results of second experiment.

Approach	Reward - mean	Reward - std	Steps - mean	Steps - std
Ours	354.9	195.1	2070.5	703.8
D4PG	265.1	242.3	1448.0	847.3
TD3	25.6	1.9	259.4	6.0

4 Discussion

We found that combining the algorithms improved the results. TD3 couldn't solve the task at hand. The policy produced was not able to exhibit walking behavior and finished each episode abruptly with falling down in runtime with a frozen model. During training, TD3 was also not able to improve its behavior, such that the standing bonus could never be achieved. In general, TD3 fell off behind D4PG and our approach. This could be due to the fact, that certain improvements like prioritized sampling or n-step returns are helpful features for solving the challenge posed in this particular environment. D4PG was able to exhibit better performance compared to TD3. On runtime, a frozen D4PG policy was able to move around and in some episodes earn the bonus by standing in the middle of the target for an amount of time.

The combined approach was able to perform even better than D4PG. It maximized reward faster than D4PG and showed to be more stable, as the training curve of our algorithm has less low-reward outliers than D4PG. It also scored higher in the second experiment than TD3 and D4PG (see table 2), while having a smaller standard deviation of reward and steps than D4PG. However, it was not able to fully solve an environment of the second round. This might be due to the fact that we chose to reduce the episode length to 1000 steps during training.

All in all, in the comparison to its components, we could not find any unfavorable repercussions for the integrated approach in the two experiments by combining the here mentioned improvements of D4PG and TD3.

References

1. AIcrowd.com. Neurips 2019: Learn to move - walk around, 2019.
2. Lukasz Kidzinski, Sharada Prasanna Mohanty, Carmichael F. Ong, Zhewei Huang, Shuchang Zhou, Anton Pechenko, Adam Stelmaszczyk, Piotr Jarosik, Mikhail Pavlov, Sergey Kolesnikov, Sergey M. Plis, Zhibo Chen, Zhizheng Zhang, Jiale Chen, Jun Shi, Zhuobin Zheng, Chun Yuan, Zhihui Lin, Henryk Michalewski, Piotr Milos, Blazej Osinski, Andrew Melnik, Malte Schilling, Helge J. Ritter, Sean F. Carroll, Jennifer L. Hicks, Sergey Levine, Marcel Salathé, and Scott L. Delp. Learning to run challenge solutions: Adapting reinforcement learning methods for neuromusculoskeletal environments. *CoRR*, abs/1804.00361, 2018.
3. Ajay Seth, Michael Sherman, Jeffrey A Reinbolt, and Scott L Delp. Opensim: a musculoskeletal modeling and simulation framework for in silico investigations and exchange. *Procedia Iutam*, 2:212–232, 2011.
4. Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.
5. Gabriel Barth-Maron, Matthew W Hoffman, David Budden, Will Dabney, Dan Horgan, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. *arXiv preprint arXiv:1804.08617*, 2018.
6. 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 preprint arXiv:1509.02971*, 2015.
7. Sergey Kolesnikov and Valentin Khrulkov. Sample efficient ensemble learning with catalyst.rl. *arXiv preprint arXiv:[WIP]*, 2019.
8. Sergey Kolesnikov and Oleksii Hrinchuk. Catalyst.rl: A distributed framework for reproducible rl research. *arXiv preprint arXiv:1903.00027*, 2019.
9. Malte Schilling, Kai Konen, Frank W. Ohl, and Timo Korthals. Decentralized deep reinforcement learning for a distributed and adaptive locomotion controller of a hexapod robot, 2020.
10. Timo Korthals, Andrew Melnik, Jürgen Leitner, and Marc Hesse. Multisensory assisted in-hand manipulation of objects with a dexterous hand. *juxi. net*, 2019.
11. Kai Konen, Timo Korthals, Andrew Melnik, and Malte Schilling. Biologically-inspired deep reinforcement learning of modular control for a six-legged robot. In *2019 IEEE International Conference on Robotics and Automation Workshop on Learning Legged Locomotion Workshop,(ICRA) 2019, Montreal, CA, May 20-25, 2019*, 2019.
12. Andrew Melnik, Sascha Fleer, Malte Schilling, and Helge Ritter. Modularization of end-to-end learning: Case study in arcade games. *arXiv preprint arXiv:1901.09895*, 2019.
13. Malte Schilling and Andrew Melnik. An approach to hierarchical deep reinforcement learning for a decentralized walking control architecture. In *Biologically Inspired Cognitive Architectures Meeting*, pages 272–282. Springer, 2018.
14. Timo Korthals, Marc Hesse, Jürgen Leitner, Andrew Melnik, and Ulrich Rückert. Jointly trained variational autoencoder for multi-modal sensor fusion. In *2019 22th International Conference on Information Fusion (FUSION)*, pages 1–8. IEEE, 2019.
15. Andrew Melnik, Lennart Bramlage, Hendric Voss, Federico Rossetto, and Helge Ritter. Combining causal modelling and deep reinforcement learning for autonomous agents in minecraft. 2019.

16. Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.
17. Will Dabney, Mark Rowland, Marc G Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
18. Neurips 2019 learn to move-environment. `http://osim-rl.stanford.edu/docs/nips2019/`, 2019.
19. David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.
20. George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.
21. Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 449–458. JMLR. org, 2017.

A Appendix

A.1 Deterministic Policy Gradient Algorithms

Environments with continuous action spaces come a bit closer to the reality. Although, they are often more difficult to solve, it is necessary to be able to solve problems in this group, when we try to make progress towards algorithms we can deploy in the real world to e.g. build controllers for robots. In the *NeurIPS 2019: Learn to Move — Walk Around* challenge using a continuous action space as design choice is justified by imitating a humanoid 3D model more realistically. This humanoid is controlled by activation of the muscles on his legs. The action space consists of activation intervals for different muscles, from which a value can be sampled from. The correct sampling of a value to produce a behaviour pattern is the goal of the environment provided by the challenge. One of the current state-of-the-art algorithms to solve such types of environments is the model-free off-policy algorithm Deep Deterministic Policy Gradient (DDPG,[6]) as well as its improved versions, twin-delayed DDPG (TD3, [4]) and distributed, distributional DDPG (D4PG, [5]). In the following we describe these algorithms in more detail.

Deep Deterministic Policy Gradient (DDPG) In this work we use DDPG as baseline algorithm for solving locomotion in reinforcement learning problems. DDPG is an off-policy, model-free algorithm and it is able to solve problems in environments with continuous action spaces. It can be seen as a variant of Deep Q-networks, as it combines Deterministic Policy Gradients (DPG, [19]) with Q-Learning and other extensions, namely experience replay and target value and policy networks. DDPG is furthermore an actor-critic-algorithm and consists of 4 different neural networks in total: The actor, the critic and both target actor and target critic. The actor is also called the policy function $\pi(s) = a$, which computes an action for a given state. The critic, also refered to as Q-value function, computes the Q-value, a numerical value that represents the discounted future reward for a state-action-pair. The critic is also the main objective to be optimized, such that we find the maximal, real Q-value for given state-action-pairs. We can derive the optimal Q-Value function $Q^*(s, a)$ by minimizing the loss between the output of the function approximator and the bellman-equation:

$$Q^*(s_t, a_t) = \mathbb{E}[r(s_t, a_t) + \gamma max_{a_{t+1}} Q'^*(s_{t+1}, \pi'(s_t))] \tag{7}$$

This computes the Q-value for a given time step t. The discount rate γ diminishes additional reward of steps into the future. This has the effect, that immediate reward is given a preference over future reward. The value and policy function on the right-hand site of the equation are the target value and policy function. Their functions are discussed in section A.1.

Given the target function and the neural networks as function approximators (given by actor and critic networks) we now can derive the loss function L:

$$L(\theta^Q) = \mathbb{E}[(Q(s_t, a_t|\theta^Q) - Y_t)^2] \tag{8}$$

where Y_t is the target in a supervised learning sense and is computed by using the Bellman-equation as intermediate optimum:

$$Y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1}|\theta^Q) \tag{9}$$

θ^Q are the function parameters for policy μ and value function Q. In the next sections we discuss how to update the parameters of a policy by the optimization step of the value function and the components experience replay buffer, exploration noise and target policy and value networks.

Deterministic policy gradients In an environment which provides a continuous action space we can derive a deterministic policy by using Deterministic policy gradient (DPG). Rather than returning a probability distribution over actions $\mathcal{A}$ given a state, a deterministic policy $\mu(s) = a$ returns a single action in a deterministic way. The main objective $J(\theta)$ in an off-policy actor-critic algorithm, which mainly optimizes the value function is defined as:

$$J(\theta) = \int_{\mathcal{S}} \rho^{\mu}(s) Q(s, \mu_\theta(s)) ds \tag{10}$$

where θ are the parameters and $\mathcal{S}$ is the state space.

$$\rho^{\mu}(s') = \int_{\mathcal{S}} \sum_{k=1}^{\inf} \gamma^{k-1} \rho_0(s) \rho^{\mu}(s \rightarrow s', k) ds \tag{11}$$

is defined as the discounted sum of state visitation probability density at state s'. $\rho^{\mu}(s \rightarrow s', k)$ gives us the probability density from state s to state s' after moving k steps by using policy μ. $\rho_0(s)$ is the initial distribution over states.

We can now compute the gradient of $J(\theta)$ using the Deterministic policy gradient theorem.

$$\nabla_\theta J(\theta) = \int_{\mathcal{S}} \rho^{\mu}(s) \nabla_a Q^{\mu}(s, a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)} ds \tag{12}$$

$$= \mathbb{E}_{s \approx \rho^{\mu}} [\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu}(s, a)|_{a=\mu_\theta(s)}] \tag{13}$$

First, the chainrule yields the gradient of Q $\nabla_a Q^{\mu}(s, a)$ with respect to a. Second, we derive the gradient of the deterministic policy $\nabla_\theta \mu_\theta(s)$ with respect to $theta$, which optimizes our policy. As an example to show how to compute updates, consider DPG in combination with on-policy actor-critic policy $SARSA$. First, we compute the TD-error in $SARSA$:

$$\delta_t = R_t + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t) \tag{14}$$

The parameter update of the value function is defined as:

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w Q_w(s_t, a_t) \tag{15}$$

Then, we can use the Deterministic policy gradient theorem to compute policy parameter updates of θ using equation 12:

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_a Q^\mu(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)} \tag{16}$$

One problem of using DPG is exploration because of the deterministic nature of the policy we optimize. On way to prevent this is to add noise to the parameter space or action space, which in this case would result in an off-policy non-deterministic policy.

Exploration noise As mentioned in A.1, DPG updates could inhibit exploration depending on the environment. To ensure exploration in the continuous action space, DDPG uses an exploration policy μ', in which noise is added to the actions of the policy network μ.

$$\mu'(s_t) = \mu(s_t) + \mathcal{N} \tag{17}$$

$\mathcal{N}$ denotes noise sampled from a noise generating process, such as Gaussian noise. The authors of the DDPG paper suggest using the Ornstein-Uhlenbeck process [20] for exploring physical environments, as it allows temporally correlated exploration.

Target value and policy networks DDPG utilizes frozen copies of value and policy function to compute the target Y_t (equation 9). More specifically, they are used to compute the right-hand site of the bellman-equation, as it was found that the learn process gets less stable, when not using copies due to the change of weights during optimization. Thus, the learning process consists of the following steps: first, a batch of training data is sampled from the experience buffer. Second, the loss L (equation 8) is computed using the target value and policy networks to generate Y_t. After update steps of value and policy networks, the target networks get softupdated by:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \tag{18}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \tag{19}$$

θ^Q and θ^μ are the parameters of the value network and the policy network, $\theta^{Q'}$ and $\theta^{\mu'}$ are the parameters of the target value and the policy network. The constant $\tau \ll 1$ is a hyperparameter, that realizes the soft update by scaling down the update step, so that the parameters of the target networks change slower than those of the actor and critic networks.

Experience replay buffer An experience sample typically consists of the tuple $s = \{s_n, a, r, d, s_{n+1}\}$, where s_n is the current and s_{n+1} the next state, a is the action, r is the reward and d is a boolean indicating, whether an episode is over or not. DDPG makes use of an experience replay buffer, in which samples generated by the interaction of policy and environment are stored and from which batches are sampled to perform updates using the value function, the bellman-equation and DPG.

A.2 Twin-delayed Deep Deterministic Policy Gradient (TD3)

One common problem of DDPG is the overestimation of the Q-value, which in turn results in policy-breaking. Twin-delayed Deep Deterministic Policy Gradient is able to diminish this effect by extending DDPG algorithm with three additional improvements. The first improvement is introducing a second value function network (as in twin-critics) to learn two q-functions. Second, it updates the policy network less frequently than the value networks. The third extension consists of target policy smoothing, i.e. adding a small amount of noise to the output of the target policy network. All these mentioned extensions provide more stability for approximating the optimal policy.

Clipped double Q-Learning Addressing overestimation of the Q-value, i.e. a state-action-pair is incorrectly valued too high, the first improvement of TD3 over DDPG is implemented by using two critics or value function networks instead of one (which also means two target critics). The two value functions are optimized with one target Q-function, which uses the minimum of the Q-values estimated by both target functions:

$$Y_t = r(s_t, a_t) + \gamma \min_{i=1,2} Q_{\theta_i}(s_{t+1}, \mu_{\theta_i'}(s_{t+1})) \tag{20}$$

By always choosing the minimum Q-value, it is more difficult for the value functions to develop an overestimation of Q-value for certain inputs.

Delayed policy network updates Less frequent updates of the policy network ensures, that the value function has a harder time converging on the failure mode, where it overestimates actions incorrectly. In a scenario, where the value function would start overestimating the outputs of a poor policy, additional updates of the value network while keeping the same policy could lead to overcoming the incorrect estimation of the poor performing policy.

Target policy smoothing The third improvement of TD3 is also an improvement of the target Y_t. The action produced by the target policy network, which is utilized in the target Q-function, gets modified by adding a small amount of noise, which is also clipped into an interval. This has the effect of covering a clipped area around the action in the action space, instead of predicting a deterministic action. In case, that the value-function produces a Q-value incorrectly to large for a certain action, adding a clipped amount of noise to the action acts as a regularizer, as the high-valued action gets smoothed by the noise.

A.3 Distributed Distributional Deep Deterministic Policy Gradient (D4PG)

D4PG, similar to TD3, is an extended version of DDPG. It implements 4 additional improvements, which overall address stability and scalability of DDPG. The first improvement, a distributional value function, provides a more stable estimation of the Q-value. Second, the process of gathering experiences is distributed over a number of in parallel acting policy networks, which store their experiences in a shared experience replay buffer. The third improvement, prioritized experience replay, weighs the produced experiences, so that important experiences are more often sampled than others. The last improvement is n-step returns. When computing the TD-error n-step-returns allows a more confident estimation of a state-action-pair by producing a reward over n steps into the future.

Multiple sampler To address the sample-inefficiency problem of model-free reinforcement learning, multiple copys of the policy network run in parallel to produce samples and store them in a shared experience buffer. The copys are updated at the same time and the number of sampler can be chosen as required.

Distributional value function D4PG uses a distributional version of critic updates. This means, that expected Q-value is modeled as a random variable, thus the value function maps the input, a state-action-pair, to a distribution Z_w, which is distributed over w. Given $Q_w(s,a) = \mathbb{E}Z_w(x,a)$, the loss for the distributional function is given by minimizing the distance between two distributions $L(w) = \mathbb{E}[d(\mathcal{T}_{\mu_\theta}, Z_{w'}(s,a), Z_w(s,a)]$, where $\mathcal{T}_{\mu_\theta}$ is the Bellman operator. As [21] show, this improvement results in a more stable learning signal.

N-step returns When constructing the target and doing the forward step of the value network for computing the loss, this improvement incorporates computing the sum of rewards of n-steps instead of having a one-step reward. The target incorporating n-step returns is computed by:

$$Y_t = \sum_{n=0}^{N-1} \gamma^N r_{t+n} + \gamma^N Q_{\theta'}(s_{t+N}, \mu_{\theta'}(s_{t+N})) \tag{21}$$

This estimates future reward more accurately.

Prioritized experience replay Instead of sampling uniformly from the replay buffer, the samples stored in the prioritized experience replay buffer are weighted with an importance weight and are sampled with a non-uniform probability p_i. The weight, which adjust the probability can, e.g. be realized by the TD-error. This would have the effect, that samples with high TD-error get sampled more often than others.