Universität Hamburg
Department Informatik
Knowledge Technology, WTM

Deep reinforcement learning for mobile robot control task

Seminar Paper

Bio-inspired Artificial Intelligence

Marcus Rottschäfer
Matr.Nr. 7327475
9rottsch@informatik.uni-hamburg.de

13.10.2019

Abstract

Reinforcement Learning is a paradigm of Machine Learning that is about building competitive agents by letting them interact with an environment. This learning is inspired by how humans congregate knowledge about how to act in specific situations from feedback emitted from the environment. It appears natural to apply this kind of learning to robots that are to learn how to solve specific tasks in the real world. Recent advances in Reinforcement Learning combine classical Reinforcement Learning (RL) algorithms with Deep Learning, creating a subfield called Deep Reinforcement Learning. This approach provides a powerful framework for a variety of task like learning to play Atari games or learning to play the game of Go. Unfortunately, some of these algorithms, especially most value-based algorithms are not working very well with high-dimensional, continuous action spaces, as often found in robotic domains. This is why the class of Policy Gradient-based algorithms has been proposed, giving some advantages over classical value-based approaches.

In this survey, we investigate and analyse a specific field of Policy Gradient-based methods, called Deep Deterministic Policy Gradient algorithms. A selection of algorithms from this class will be evaluated and critically compared. The comparison is based on the use of these algorithms in the context of a mobile robot control task, whose implementation is documented in a paper parallel to this survey. The results focus on the theoretical aspects of the algorithms, supported by practical evaluation found in other papers. Findings suggest that in most cases, an algorithm called Twin Delayed Deep Deterministic Policy Gradient is to be preferred when applying Deep Deterministic Policy Gradient algorithms, for example in the context of a mobile robot.

Contents

Abbreviations

DDPG Deep Deterministic Policy Gradient

DNN Deep Neural Network

DPG Deterministic Policy Gradient

DQN Deep Q-Networks

RL Reinforcement Learning

TD3 Twin Delayed Deep Deterministic Policy Gradient

1 Introduction

The idea behind RL is to make an agent act in an environment and to use the feedback from that environment to improve the agents policy to reach a goal state. In other words, RL is about making an agent (for example a robot) learn a policy to reach a goal by interacting with an environment. This interaction part is a key concept in RL; it implies that an agent can sense some aspects of its environment and that it can take actions to change the state of the environment [17].

RL is a very active area of research in the field of Machine Learning, itself becoming more and more important for Artificial Intelligence. Recent advances in RL like learning to play Atari games [13] or beating the world champions in Go [14, 16] and StarCraft II [20] show a promising success in achieving top-level performance through the combination of RL with deep learning approaches. Next to applications like playing games, RL has been used in optimizing chemical reactions [24], web system configuration [3] or traffic light control [2], and has a huge impact on the field of robotics [8].

In this survey, we want to focus on the use of deep reinforcement learning methods in the domain of mobile robotics. More specifically, we focus on the domain of continuous action spaces and policy gradient RL algorithms. These algorithms are a special kind of RL algorithms that learn an approximation to the optimal agent's policy by using a parameterized policy representation, for example a Deep Neural Network (DNN) [17].

The reason behind choosing this special class of algorithms for the survey is that these methods appear to be better suited for the mobile robot control task. This is because agents like robots naturally interacting with the physical world are often required to take actions that come from a continuous domain. For example, an autonomous car must control continuous parameters like the current cars speed, the angle of the steering wheel or how much force to put on the brakes. This limits the application of many recent value-based deep RL algorithms like Deep Q-Networks (DQN) [13] for the use in continuous action domain problems, as those algorithms only work with discrete action spaces. A discretization of the continuous action space is often not feasible. It necessarily leads to less finer-grained actions being taken and to high-dimensional, discrete action spaces. For example, when discretizing a 7-degree of freedom system to have just three possible values per degree of freedom, the dimensionality of the action space becomes $3^7 = 2187$ [9]. This is why methods that work with continuous action spaces are sometimes preferred.

There are a few examples of the application of deep policy gradient-based algorithms in the mobile robot domain. In a recent implementation [4], the authors successfully applied a deep policy gradient-based algorithm called Deep Deterministic Policy Gradient (DDPG) to a TurtleBot3 robot [1], changing the default implementation which uses the discrete-action, value-based DQN. The TurtleBot3 robot learned to drive through an obstacle environment and learned to reach a specific region in the simulated map environment. The advantages of DDPG over DQN are, at least in theory, a better applicability to high-dimensional, continuous action spaces, due to the use of a policy gradient method for learning.

In another implementation, Ghouri et al. applied the DDPG algorithm and a distributed version D4PG to a quadcopter in a simulated environment [?]. Using those deep policy gradient learning algorithms, they were able to achieve flight control for the drone. Learned roll, pitch and yaw angles responses suggest that the distributed version D4PG is slightly better suited for the task.

Another implementation of DDPG in the mobile robot domain focuses on the problem of mapless navigation [?]. The mobile robot, build on the Turtlebot platform, is able to learn navigation in an obstacle domain in the real world, based on only 10-dimensional Laser range sensor readings and the location of the goal area. The transfer of the robot trained in a simulated environment onto a robot placed in the real world worked exceptionally well and enables the robot to seamlessly navigate an obstacle course in an office setting in the aim of navigating towards a specific area without a map of the environment.

As the DDPG algorithm achieved promising results for the robot and in a variety of physical control task [9], we compare the vanilla DDPG algorithm with a selection of DDPG-based methods like DDPG with Prioritized Experience Replay and Twin Delayed Deep Deterministic Policy Gradient (TD3). This survey gives a detailed introduction into the topic of RL (Section 2) and to (deterministic) policy gradient methods specifically (Section 2.3 - 2.4), followed by the evaluation and comparison of the three DDPG algorithms in the consecutive chapters 3 and 4.

2 Foundations

As mentioned previously, RL enables an agent to learn a goal-directed policy by interacting (e.g. taking actions) with an environment. Therefore, the environment has to provide some kind of measure (rewards) of how good the interaction is in the effort to reach the goal. In a more technical sense, the general objective of a RL algorithm is to maximize a numerical reward signal [17]. That reward, informally, gives feedback of how good the selection of a specific action in a specific state is. It helps the agent in shaping it's policy towards reaching the goal and is therefore essential in the learning and in defining the goal.

Next to the concept of reward, there is a second concept that is essential to a RL setup. This is the concept of a policy; a policy can be seen as a function $\pi : \mathbb{S} \mapsto \mathbb{A}$ that maps a perceived state to an action the agent should take. In case of stochastic policies, π maps to a probability distribution over actions. The policy is the core of an RL agent, as it alone is enough to fully determine the behaviour of the agent [17].

Having the reward signal available as an immediate numerical feedback to the agent, we are typically interested in developing a policy that maximizes rewards in the long run. To measure that reward of a state in the long run, the concept of the value function has been proposed. The idea is to define the value of a state, $V(s_t)$, to be the total amount of reward of all future states, starting from that state. As we are dealing with probabilistic environments, this is the expected cumulative reward over the future, starting from state s_t [17]. The normal reward only gives feedback concerning the current action in the current state. It gives no hint about an agents policy performance in the long run, this is why a measure like the value function is needed. The value function needs to regard all future states following a state s_t, so it is dependent on a policy π to experience all the future state-action pairs needed for the expected reward calculation [17]. In summary, the value function $V^\pi : \mathbb{S} \mapsto \mathbb{R}$ gives each state a numerical value determining its expected future reward following policy π.

2.1 Q-Function

The value function enables us to determine a states utility in the current environment reward setting, given a policy π. It could be convenient to extend that concept of the long-term, expected reward to a state-action pair (s_t, a_t). This is done with the so-called action-value function $Q^\pi : \mathbb{S} \times \mathbb{A} \mapsto \mathbb{R}$. Given a policy π, a reward $R(s_t, a_t)$ and a discount factor $\gamma \in [0, 1]$ that controls how farsighted the evaluation of future rewards is, this is [17]:

$$Q^\pi(s,a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k}) \mid s_t = s, a_t = a] \tag{1}$$

However, we are typically interested in finding a policy π that maximizes the expected discounted reward. That optimal policy π^* is rarely known in the beginning. Our goal is to find such optimal function, rather than only being able to evaluate $Q^\pi(s, a)$ for a fixed policy π. For this, the Bellman optimality equation provides a recursive relationship for the optimal action-value function, that can be used to find the optimal policy π^*:

$$Q^*(s,a) = \sum_{s',r} p(s', r \mid s, a)[r + \gamma \max_{a'} Q^*(s', a')], \quad \pi^*(s) = \arg\max_{a} Q^*(s,a) \tag{2}$$

$p(s', r \mid s, a)$ is the probability of transitioning into state s', receiving reward r when taking action a in state s and is also called the dynamics function [17]. A well-known RL algorithm to obtain an arbitrarily precise approximation of Q^* is called Q-learning [22, 21]. Q-learning uses the concept of Temporal-Difference learning of Q^* [17]. The details of how the algorithm obtains the optimal action-value function Q^* should not matter here; it is important to note that Q^* can be calculated off-policy. Off-policy learning refers to methods that learn an optimal target-policy π although following a possibly sub optimal, more exploratory behavior policy β in the learning process [17].

Q-learning can also be seen as a model-free RL algorithm, meaning that the algorithm does not require a model of the environment to begin with [17]. This can give an advantage regarding larger state and action spaces, as no model of the environment's dynamics must be stored to run the algorithm.

The Q-function and the derived optimality principle play an important role in many RL algorithms and lay the foundation for the following sections, introducing into DQN and a selection of policy gradient algorithms.

2.2 Deep-Q Networks

As we saw in the previous section 2.1, the Q-learning algorithm provides a way to learn the optimal Q-function Q^* without the need for a model of the environment. This makes it appealing for usage in e.g. a robotic setting. However, when applying the algorithm to real-world problems with a huge number of states and possible actions, problems occur due to the finite resources of a computer. For example, it becomes impossible to store $Q^*(s,a)$ for every possible state-action pair. This is why parameterized versions of the Q-function have been proposed, $Q(s,a;\theta) \approx Q^*(s,a)$ with θ being the parameter vector that is much smaller than the number of all state-actions pairs. The DQN algorithm proposed by Mnih et al. uses a deep convolutional neural network as a function approximator for the optimal Q-function [12, 13].

Previous approaches to approximate the optimal action-value function Q^* with nonlinear function approximators were shown to be unstable and diverging [13]. This is due to some issues arising when naively applying Q-function approximation [13]:

1. The sample that is experienced for a temporal difference update, the agent's experience $e_t = (s_t, a_t, r_t, s_{t+1})$ at time step t, is depending on previous experiences $(e_1, ..., e_{t-1})$. The sequence of observations is not independent and identically distributed.

2. The learning of the parameters θ of the state-action values $Q(s,a;\theta)$ depends on the target values $y_t = r + \gamma \max_{a'} Q(s', a'; \theta)$, itself depending on the state-action value estimates $Q(s,a;\theta)$ again. These correlations may cause instabilities and divergence when learning the parameters θ.

DQN overcomes these issues using two ideas: a bio-inspired mechanism called experience replay and the usage of a second neural network, updated only periodically, whose task is to approximate the Q-values for the target policy [13, 9].

The idea behind experience replay is to store the experiences e_t received by the agent in a replay memory $D_t = \{e_1, ..., e_t\}$. Instead of using each new experience e_t to update the Q-values, a sample or a minibatch of experiences is drawn uniformly from the replay memory and used for the update. This removes the temporal dependency of the sequence of experiences, if the replay memory is large enough. For example, Mnih et al. use a replay memory of the most recent 1 million frames for learning to play Atari 2600 games [13]. The replay buffer is sampled for minibatches of a constant size N, which makes learning better tractable for larger networks. The NFQCA algorithm [5] for example uses batch learning for stability reasons, but cannot be applied to large networks due to not making use of minibatches [9].

The second idea is to use a second neural network $Q^\pi(s,a;\theta')$ to calculate the Q-values for the target policy, and to update that second network only every C steps of parameter updates in the first network $Q^\beta(s,a;\theta)$ [13]. After the C steps, the parameters of the target network are simply copied from the first network, $\theta' = \theta$.

This architecture partly overcomes the correlation between the target values and the updates to the first network, which have been known to cause oscillations or divergence of the policy [13]. When fixing the parameters of the target network for C time steps, a delay between the updates to Q^β and their effect on the target network is implemented, improving the stability of learning [13].

DQN is an algorithm to apply nonlinear Q-function approximation in a robust way, taking the Q-learning algorithm to neural network methods. The DDPG algorithm and many of its enhancements use parts of DQN for deep RL in the continuous action domain.

2.3 Policy Gradient

As mentioned previously, the method to select the optimal action in the algorithms presented above is to choose the action that maximizes the state-action value function for a given state. This means that in order to obtain a policy π, it is essential to have estimated the action values.

Policy gradient methods follow a different approach. They are driven by the aim to develop a parameterized policy that best approximates the optimal policy, not necessarily relying on the estimates of a value function $V(s)$ or $Q(s,a)$ for the action selection. Although the value function is not needed for the action selection anymore, learning of the policy parameters could still somehow make use of it (see some examples for algorithms in Section 3). Equation 3 shows the parameterization of a stochastic policy π with a parameter vector $\boldsymbol{\omega}$, which for a given state s outputs a probability distribution over the action space [17].

$$\pi(a \mid s, \boldsymbol{\omega}) = Pr\{A_t = a \mid S_t = s, \boldsymbol{\omega}_t = \boldsymbol{\omega}\} \tag{3}$$

The reason these methods are called Policy Gradient methods is that the gradient of a policy performance measure $\nabla J(\boldsymbol{\omega})$ will be used to learn the policy parameters [17]. The idea is to use gradient ascent so that the parameters $\boldsymbol{\omega}$ maximize the policy performance measure. This performance measure $J(\boldsymbol{\omega}) = v_{\pi_\omega}(s_0)$ is defined to be the value of the start state of an episode [17].

Learning a parameterized policy with gradient ascent allows the action probabilities to change smoothly during training, in contrast to the greedy action selecting of the action-value methods $\pi(s) = \arg\max_a Q(s,a)$, and causes policy gradient methods to have stronger convergence guarantees [17].

There is also another advantage to policy gradient methods that make them especially appealing to mobile robot control tasks. Naturally, the actions are now learned as a function of states and parameters, so $\mathbb{A} \subseteq \mathbb{R}^N$ can be a real-valued, continuous action space. This makes learning in high-dimensional action spaces feasible. It allows the selection of continuous actions like speed and angles, rather than depending on action discretization as needed in action-value methods like DQN. These methods failed in high-dimensional, continuous action spaces, as one is often confronted with in robot tasks (e.g. [?]). The reason is that action-value methods cannot select continuous actions ($\pi(s) = \arg\max_a Q(s,a)$ would require an optimization over a each time) and the discretization needed for finer grained actions would lead to an intractable amount of discrete actions [9].

Many approaches that use the policy gradient like DDPG [9] or Deterministic Policy Gradient (DPG) [15] are referred to as actor-critic methods, meaning that these methods learn a parameterized policy (the actor), as well as a parameterized value function (the critic).

2.4 Deterministic Policy Gradient Methods

While policy gradient methods in general refer to parameterized, stochastic policies like shown in eq. (3), deterministic policy gradient methods aim to learn parameters for a deterministic policy, $\pi_{\boldsymbol{\omega}}(s) = a$. The learning of the parameters $\boldsymbol{\omega}$ is also achieved with gradient ascent, this time using the deterministic policy gradient. Silver et al. showed that the deterministic policy gradient exists and that it can be estimated more efficiently than the stochastic policy gradient [15]. The intuition is here that the stochastic policy gradient requires an integration over both the state and action space, while the deterministic version only requires an integration over the state space [15], making the deterministic policy gradient in practice better suited for high-dimensional action spaces, as less samples are required.

$$\nabla_{\boldsymbol{\omega}} J(\boldsymbol{\omega}) = \mathbb{E}_{s_t \sim p^{\beta}}[\nabla_a Q(s, a; \theta)|_{s=s_t, a=\pi_{\boldsymbol{\omega}}(s_t)} \nabla_{\boldsymbol{\omega}} \pi_{\boldsymbol{\omega}}(s)|_{s=s_t}] \tag{4}$$

Equation 4 shows the deterministic policy gradient found by Silver et al. [15]. Here, policy $\pi_{\boldsymbol{\omega}}(s_t)$ refers to the deterministic target policy, p^{β} describes a probability distribution over the states s_t visited by following the exploratory behaviour policy β.

3 Evaluation of Deterministic Policy Gradient Algorithms

The purpose of this survey is to compare the DDPG algorithm and other methods based on the DPG in the context of a mobile robot control task. This chapter introduces to the algorithms used in the comparison, based on the RL and policy gradient foundations presented in the last chapter.

The comparison of different DPG-based algorithms focuses on the off-policy, DNN-based variants like DDPG [9], DDPG with Prioritized Experience Replay [7] and TD3 [?]. There are more DNN-based approaches like Distributed Distributional Deep Deterministic Policy Gradient (D4PG) [?] or Multi-agent DDPG (MADDPG) [10], but these will be excluded as they focus more on the distributed training (D4PG) or on environments with multiple agents in a Markov game (MADDPG).

We further exclude all methods not relying on DPG and DNN function approximators specifically. Other policy gradient methods that learn on-policy, like REINFORCE [23] or A3C [11], are not regarded as with these, it is not possible to learn from the history of an exploratory policy β. This leads to the selection of DDPG variants presented and compared in the following sections, starting with the vanilla DDPG.

3.1 Deep Deterministic Policy Gradient (DDPG)

The DQN algorithm presented in section 2.2 is capable of end-to-end learning, meaning: for the Atari game playing task solved in the corresponding paper [13], the DNN function approximator is able to estimate the action-value function directly from raw pixel values. This is an important step in building competitive RL agents that operate on high-dimensional, unprocessed input. However, a drawback of the DQN algorithm is its inability to deal with high-dimensional, continuous action spaces.

This is where policy gradient methods, due to their policy parameterization, naturally offer an advantage over action-value methods. The idea of DDPG is to combine the ability of DQN to learn the action-value function using deep function approximators with DPG's ability to deal with high-dimensional, continuous action spaces. The resulting model-free, off-policy, actor-critic algorithm can learn policies in high-dimensional, continuous action spaces end-to-end [9]. This makes it especially appealing to physical control problems like often faced in robotics.

Algorithm 1 shows the DDPG procedure in pseudo code. Note how the different concepts presented in the previous chapter emerge in the algorithm: Just like DQN, DDPG maintains two neural networks Q and Q' for the action-value function approximation and the target action-value prediction. In the actor-critic framework, these work as the critic while the two policy neural networks π and π' work as the actor.

The first real difference to DQN appears in the exploratory policy action selection phase. While action-value methods could e.g. use ϵ-greedy action selection to choose a random action with a certain percentage for exploration preservation, policy gradient methods with continuous action spaces need to follow a different approach. Here, a noise vector $\mathcal{N}_t$ generated randomly from e.g. an Ornstein-Uhlenbeck process [19] is added to the action chosen by the actor network, $a_t = \pi_{\boldsymbol{\omega}}(s_t) + \mathcal{N}_t$. This step is essential to the learning as it ensure continuous exploration of the action space.

Same as DQN, the algorithm makes use of a replay buffer R to achieve training in a more state-visitation independent manner. The training of the critic and actor networks is based on minibatch sampling from R. While the critic is updated the same way as in DQN, by temporal-difference learning from the predictions of the target network, the actor networks update must also be accounted for. This is where the deterministic policy gradient proposed by Silver et al. [15] is used. The parameters $\boldsymbol{\omega}$ of the actor network π can be updated by applying gradient ascent with the estimation of the deterministic policy gradient $\nabla_{\boldsymbol{\omega}} J$.

The last major idea that DDPG uses based on DQN is the delayed update of the target networks Q' and π'. While in DQN the target network parameters will simply be copied from the critic network after C training steps, achieving a delayed updated leading to more stable learning, DDPG implements what Lillicrap et al. call soft target updates. The target networks are updated at each step, but the amount to which the newly learned parameters of critic Q and actor π influence the target update is controlled by a constant τ, typically $\tau \ll 1$.

Algorithm 1: DDPG algorithm

```
Randomly initialize critic network $Q(s, a; \theta)$ and actor $\pi_{\boldsymbol{\omega}}(s)$ with weights
  $\theta$ and $\boldsymbol{\omega}$
Initialize target network $Q'(s, a; \theta')$ and actor $\pi'_{\boldsymbol{\omega}'}(s)$ with weights $\theta' \leftarrow \theta$,
  $\boldsymbol{\omega}' \leftarrow \boldsymbol{\omega}$
Initialize replay buffer $R$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for t=1,T do
        Select action $a_t = \pi_{\boldsymbol{\omega}}(s_t) + \mathcal{N}_t$ 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 $y_i = r_i + \gamma Q'(s_{i+1}, \pi'_{\boldsymbol{\omega}'}(s_{i+1}); \theta')$
        Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i; \theta))^2$
        Update the actor policy using the sampled policy gradient:
          $\nabla_{\boldsymbol{\omega}} J(\boldsymbol{\omega}) \approx \frac{1}{N} \sum_i \nabla_a Q(s, a; \theta)|_{s=s_i, a=\pi_{\boldsymbol{\omega}}(s_i)} \nabla_{\boldsymbol{\omega}} \pi_{\boldsymbol{\omega}}(s)|_{s=s_i}]$
        Update the target networks: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$,
          $\boldsymbol{\omega}' \leftarrow \tau\boldsymbol{\omega} + (1-\tau)\boldsymbol{\omega}'$
    end
end
```

Furthermore, DDPG uses batch normalization [?] for the training of the DNN, making the training more robust to differently scaled inputs like occurring with different physical units.

3.2 DDPG with Prioritized Experience Replay

Hou et al. [7] propose an enhancement to DDPG by applying a mechanism called prioritized experience replay. They propose to change the probability distribution of sampling from the replay buffer to be a non-uniform distribution, prioritizing some samples (experiences) over other samples. The idea is inspired by how humans learn from interaction. That is, experiences that are more painful or more fruitful than others are more important and are kept in the memory for a longer time span to learn from [7]. This idea gives rise to the principle of prioritized experience sampling, in empirical evaluations achieving faster training times and more stable training than the vanilla DDPG algorithm.

To implement the prioritization of successful and painful attempts over more uninformative attempts, Hou et al. propose to use the absolute temporal difference errors (TD-errors) of the training as a metric [7]. The idea is that experiences with a high magnitude of absolute TD-error provide more information for the training and must therefor be replayed more often and more frequent. The experiences stored in the replay buffer $e_1 ... e_{|R|}, e_i = (s_i, a_i, r_i, s_{i+1})$ are sorted according to their absolute TD-error, $\delta_i = r_i + \gamma Q'(s_{i+1}, a_{i+1}; \theta') - Q(s_i, a_i; \theta)$. Now when sampling a minibatch from the replay buffer, each experience j is sampled with probability

$$P(j) = \frac{D_j^\alpha}{\sum_k D_k^\alpha} \tag{5}$$

where $D_j = \frac{1}{rank(j)}$ weights the ranking of the experiences in the replay buffer. α is a hyperparameter that controls the importance of the prioritization. Equation 5 ensures that although the replay buffer has a definitive ranking, all experiences can be sampled, higher ranked experiences are just more likely [7].

This prioritized experience replay introduces a bias over the state visitation distribution, as some experiences are more frequently replayed than others. To overcome this bias, the authors propose to use a weighting scheme for each experience in the minibatch,

$$W_j = \frac{1}{|R|^\beta P(j)^\beta} \tag{6}$$

where $|R|$ is the size of the replay buffer and β is another importance-controlling hyperparameter. The W_j's weight the TD-errors in the critic update step and are used to overcome the bias of the state visitation frequency [7].

3.3 Twin Delayed Deep Deterministic Policy Gradient (TD3)

Fujimoto et al. showed that actor-critic methods suffer from an overestimation bias of the Q-function. Just like in Q-learning with discrete action spaces, policy updates achieved with the DPG cause an overestimation of the Q-values [?]. This problem affects methods like DDPG, as shown practically and theoretically in [?].

The authors propose TD3 as an advancement of DDPG, eradicating some of the known issues with DDPG. The overestimation bias is one of the main problems overcome in the TD3 algorithm. This is achieved by using a clipped variant of the Double Q-learning algorithm [6].

The general idea of the Clipped Double Q-learning is to replace the critic network with two separate networks. So instead of one critic and one target network like in DDPG, TD3 has two critic and two target networks: $Q(s, a; \theta_1), Q(s, a; \theta_2)$ and $Q'(s, a; \theta'_1), Q'(s, a; \theta'_2)$. The target network with the smaller Q-value prediction will be chosen as the learning target in each time step and the critic that provides the lower mean-squared error (MSE) loss will give the updates to the two critics parameters.

Clipped Double Q-learning may introduce an underestimation bias of the Q-function, that will however not affect the algorithm negatively. This is because the policy updates do not explicitly propagate underestimated actions, in contrast to overestimated actions [?]. Using two critic networks (as twins) gives the first part of the algorithm's name.

Another problem seems to appear when using actor-critic methods with target networks, like done in DDPG. It could occur that the training of the agent diverges due to the overestimation of a poor policy [?]. The intuition here is that an inaccurate critic value prediction could lead to a poor policy performance, which then leads to diverging Q-value estimates again. This can be interpreted by that the interaction between the actor and the critic updates could get to entangled. Fujimoto et al. propose to overcome this issue by updating the policy network less frequently than the critic networks, for example every d training steps, in contrast to every training step as done in DDPG. This delayed update of the actor provides additional stability and increases the quality of the policy updates [?].

The last change used by TD3 is a regularization of the target policy. As pointed out by [?], DPG methods tend to lead to high-variance in target values. This variance can be problematic and TD3 accounts for that by adding a small random noise to the action predicted by the target policy, averaged over the whole minibatch.

The TD3 algorithm clearly outperforms DDPG and other recent actor-critic algorithms in continuous control MuJoCo environment tasks [?].

4 Discussion

In this chapter we discuss and compare the deterministic policy gradient algorithms presented in chapter 3. The discussion will highlight advantages and general drawbacks regarding theoretical and practical aspects of the three algorithms.

4.1 Continuous Action Spaces

As mentioned previously, DDPG should be preferred over DQN when handling RL domains with continuous and high-dimensional action spaces. This is due to the strength of policy gradient methods, especially the deterministic policy gradient [15], to learn a parameterized policy. All three algorithms presented in chapter 3 are using the deterministic policy gradient in combination with deep learning methods and are therefore, in principle, equally equipped to handle high-dimensional, continuous action spaces. Just the timing of the actor updates achieved with DPG differ, for example is TD3 updating the actor less frequently than the critic networks, achieving a better entanglement between the actor and the critic. This is part of the reason why TD3 outperforms the DDPG algorithm in more complicated action domains like the OpenAI gym tasks HalfCheetah-v1 and Ant-v1 [?].

4.2 Q-value Overestimation Bias

Q-learning algorithms using function approximation to represent the Q-function suffer from an overestimation bias, as shown by Thrun et al. [18]. This means that most of the Q-values are overestimated in regard to the true Q-values. Even worse; with imprecise estimation necessarily occurring due to the function approximation, approximation errors accumulate, giving higher values to arbitrary bad state-action pairs [?].

This bias leads to sub-optimal policies and is a problem effecting DDPG and DDPG-based algorithms like DDPG with Prioritized Experience Replay. TD3 is one of the few deep actor-critic algorithms to overcome this issue due to the use of Clipped Double Q-learning and is in this regard superior to the other two algorithms. Again, this is part of the reason why even in complicated domains like the OpenAI gym tasks HalfCheetah-v1 and Ant-v1, TD3 achieves better results than the basic DDPG algorithm and a selection of other policy gradient methods [?].

5 Conclusion

We introduced into the topic of RL, concentrating on Deep RL methods using the (Deterministic) Policy Gradient for learning. The evaluation and comparison of three different DDPG-based algorithms presented strengths and weaknesses of the different approaches. We compared the algorithms based on a variety of factors like the ability to deal with high-dimensional, continuous action spaces and their affinity to overestimate the Q-function in the learning. The comparison suggests that from the three algorithms DDPG, DDPG with Prioritized Experience Replay and TD3, the latest one provides the most advantages over the other two algorithms. This is not surprising since TD3 was built to overcome several minor issues known to DDPG and, as a successor to DDPG, is expected to have a better general performance. Theoretical justification of the advantages of DDPG with Prioritized Experience Replay over DDPG and TD3 over the other two have been given.

What could be included in future work is a practical comparison of the three algorithms. Fujimoto et al. provide a practical comparison of TD3 with DDPG, but in this setting the DDPG with Prioritized Experience Replay algorithm is missing [?]. Although one can expect a superior performance of TD3, based on the theoretical justifications made in this paper and on Fujimoto et al.'s findings, this is a hypothesis still to be confirmed practically.

Another future research objective could be to extend the selection of DDPG-based algorithms that have been used in this survey. As mentioned earlier, algorithms like Distributed Distributional Deep Deterministic Policy Gradient (D4PG) [?] or Multi-agent DDPG (MADDPG) [10] are based on DDPG and fit into the framework of this theoretical comparison.

In another comparison, it could be interesting to compare this narrow class of Deep DPG algorithms presented in this paper with other state-of-the-art Policy

Gradient methods. For the context of the mobile robot control task where a continuous action domain is required, it could be interesting to see the different performances with non-DDPG-based algorithms like A3C [11].

References

[1] TurtleBot3.

[2] Itamar Arel, Cong Liu, Tom Urbanik, and Airton G Kohls. Reinforcement learning-based multi-agent system for network traffic signal control. *IET Intelligent Transport Systems*, 4(2):128–135, 2010.

[3] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. A reinforcement learning approach to online web systems auto-configuration. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 2–11. IEEE, 2009.

[4] Jan-Gerrit Habekost and Nicolar Frick. Deep Reinforcement Learning for Mobile Robot Control Task, 2020.

[5] Roland Hafner and Martin Riedmiller. Reinforcement learning in feedback control : Challenges and benchmarks from technical process control. *Machine Learning*, 84(1-2):137–169, 2011.

[6] Hado V Hasselt. Double Q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[7] Yuenan Hou, Lifeng Liu, Qing Wei, Xudong Xu, and Chunlin Chen. A novel DDPG method with prioritized experience replay. *2017 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2017*, 2017-Janua:316–321, 2017.

[8] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[9] 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. 9 2015.

[10] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in Neural Information Processing Systems*, 2017-Decem:6380–6391, 2017.

[11] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. pages 1–9, 2013.

[13] 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, 2015.

[14] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and others. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[15] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. *31st International Conference on Machine Learning, ICML 2014*, 1:605–619, 2014.

[16] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, and others. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[17] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[18] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, 1993.

[19] George E Uhlenbeck and Leonard S Ornstein. On the theory of the Brownian motion. *Physical review*, 36(5):823, 1930.

[20] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, and others. AlphaStar: Mastering the real-time strategy game StarCraft II. *DeepMind Blog*, 2019.

[21] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[22] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[23] Lilian Weng. Policy Gradient Algorithms. *lilianweng.github.io/lil-log*, 2018.

[24] Zhenpeng Zhou, Xiaocheng Li, and Richard N Zare. Optimizing chemical reactions with deep reinforcement learning. *ACS central science*, 3(12):1337–1344, 2017.