Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Autonomous Systems Lab
Prof. Roland Siegwart

Seminar in Robotics for CSE

State-of-the-art Techniques for Deep Q-learning

Spring Term 2017

Supervised by:
Fadri Furrer
Tonci Novkovic

Author:
Lorenzo Giacomel

Declaration of Originality

I hereby declare that the written work I have submitted entitled

State-of-the-art Techniques for Deep Q-learning

is original work which I alone have authored and which is written in my own words.[1]

Author(s)

Lorenzo	Giacomel

Student supervisor(s)

Fadri	Furrer
Tonci	Novkovic

Supervising lecturer

Roland	Siegwart

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/plagiarism-citationetiquette.pdf). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Place and date	Signature

[1] Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Contents

1 Introduction

One of the most fascinating challenges in the field of robotics has always been to build machines that are able to learn how to succeed in their given task. Through Reinforcement Learning (RL) techniques robots successfully learn how to behave with minimal human input.
Basically, in RL an agent is able to understand the optimal policy through repeated episodes of experience in the real world.
Thanks to advances in the theory of Markov Decision Processes, optimization (with Bellman Equations) and dynamic programming it has been possible to formulate RL problems conveniently and to prove their effectiveness.
Although, as often happens in robotics, purely theoretical approaches do not adapt well to the real world. In order to apply RL to concrete robotics tasks, one needs to take into account many issues. For instance, a perfect learning algorithm that requires long computations for a maximization step in a continuous domain would be totally useless. Moreover, it is necessary to make the agent find the optimal policy through the least possible number of trials. For this reason, nowadays, one of the biggest challenges in RL is to create fast and efficient algorithms that exploit all the possible knowledge from every single learning episode.
This report is divided into three sections. The first is a general introduction to RL, to the issues that one has to face to apply RL to robotic tasks and to the methods generally used to tackle those issues. In the second section, we analyse the application of RL Learning than robotic tasks because state and action spaces are discrete. Indeed, this intermediate step has been fundamental to be able to create more complicated algorithms for the continuous case. The Atari case has been studied mostly by the researchers of Google DeepMind. They achieved to apply deep neural networks to Q-learning, creating the Deep Q-learning algorithm. The third section is about the application of Deep Q-learning to the continuous case. The main achievement in this direction is the Continuous Q-learning with Normalized Advantage Functions algorithm, formulated by the research group led by Sergey Levine at Google Brain.

2 A concise Introduction to Reinforcement Learning

In this chapter we report the classical mathematical formulation of RL by means of Markov Decision Processes [1]. Moreover, we explain why it is not easy to apply already existing algorithms to RL, even though its formulation just makes use of some typical equations of mathematical optimization.

2.1 Mathematical Formulation of Reinforcement Learning

In RL an agent tries to maximize an accumulated reward over its life-time.
The agent can be modelled being in a state $\boldsymbol{s} \in S$ and can perform actions $\boldsymbol{a} \in A$ (A and S can be discrete or continuous and multi-dimensional). We consider only Markov states. A Markov state is a sufficient statistic of the future, i.e. it contains all the required information to predict the future.
Such a state has the so-called Markov property:

$$P(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{s}_{t-1}, \ldots, \boldsymbol{s}_1) = P(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t)$$

For every step the agent earns a scalar reward $r(\boldsymbol{s}, \boldsymbol{a})$, which is a function of a given state and action (it could also be a function of the current state only, $r(\boldsymbol{s})$, or a function of the current state, chosen action and future state, $r(\boldsymbol{s'}, \boldsymbol{a}, \boldsymbol{s})$).
On the other hand the environment $\mathcal{E}$ can also be stochastic, that means that it is modelled as a probability distribution on states and rewards.
The goal of RL is to find an optimal policy, which is a state-action mapping π that maximises the total reward. A policy can be deterministic or probabilistic. The former is a simple function $\boldsymbol{s} \longmapsto \boldsymbol{a} = \pi(\boldsymbol{s})$, the latter chooses the action according to a probability function, i.e. $\boldsymbol{a} \sim \pi(\boldsymbol{s}, \boldsymbol{a}) = P(\boldsymbol{a}|\boldsymbol{s})$.
Classical RL approaches are based on the assumption that we have a Markov Decision Process (MPD) $\langle S, A, R, T \rangle$ consisting of the set of states S, the set of actions A, the rewards R, and the transition probabilities $T(\boldsymbol{s'}, \boldsymbol{a}, \boldsymbol{s}) = P(\boldsymbol{s'}|\boldsymbol{s}, \boldsymbol{a})$ that describe the effect of the actions on a state.

2.1.1 Goals of Reinforcement Learning

The goal of RL is to discover an optimal policy π^* that maps states to actions so as to maximize the expected reward from the start distribution.
The expected future return at time t can be modelled in different ways:

- **Finite Horizon Model**: $R_t = \sum\limits_{i=t}^{T} r(\boldsymbol{s}_i, \boldsymbol{a}_i)$
- **Infinite Horizon Model with discounting factor**: $R_t = \sum\limits_{i=t}^{\infty} \gamma^{i-t} r(\boldsymbol{s}_t, \boldsymbol{a}_t)$
- **Average Reward Criterion**: $R_t = \lim\limits_{T \to \infty} \left\{ \frac{1}{T-t} \sum\limits_{i=t}^{T} r(\boldsymbol{s}_i, \boldsymbol{a}_i) \right\}$

After one of those models has been chosen, the total reward becomes $J = \mathbb{E}_{r_i, \boldsymbol{s}_i \sim E, \boldsymbol{a}_i \sim \pi}[R_0]$.

Two natural goals arise for the learner. In the first, we attempt to find an optimal strategy at the end of a phase of training or interaction. In the second the goal is to maximize the reward over the whole time the robot is interacting with the world. To gain information about the rewards and the behaviour of the system, the agent needs to decide whether to play it safe and stick to well known actions with high rewards or to discover new strategies with even higher reward (*exploration-exploitation trade off*).
In principle, algorithms to solve MDPs in polynomial time are known, but typical robotics state spaces are continuous or too large to allow direct solutions (*curse of dimensionality*).
There are two types RL methods:

- *Off-policy* methods learn independently of the employed policy, i.e. an explorative strategy that is different from the final desired policy can be employed during the learning process.
- *On-policy* methods collect sample information about the environment using the current policy.

2.1.2 Reinforcement Learning in the Average Reward Setting

In the following derivations the policy is assumed to be a conditional probability distribution $\pi(\boldsymbol{s}, \boldsymbol{a}) = P(\boldsymbol{a}|\boldsymbol{s})$. Reinforcement learning and optimal control aim at finding the optimal policy π^* that maximizes the average return J, which under all the assumptions we have made becomes $J(\pi) = \sum_{\boldsymbol{s},\boldsymbol{a}} \mu^\pi(\boldsymbol{s})\pi(\boldsymbol{s}, a)r(\boldsymbol{s}, \boldsymbol{a})$, where μ^π is the stationary state distribution generated by policy π acting in the environment.
The control problem can be formulated as:

$$\begin{aligned} \max_\pi \quad J(\pi) &= \sum_{\boldsymbol{s},\boldsymbol{a}} \mu^\pi(\boldsymbol{s})\pi(\boldsymbol{s},\boldsymbol{a})r(\boldsymbol{s},\boldsymbol{a}), \\ \text{s.t. } \mu^\pi(\boldsymbol{s}') &= \sum_{\boldsymbol{s},\boldsymbol{a}} \mu^\pi(\boldsymbol{s})\pi(\boldsymbol{s},\boldsymbol{a})T(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}'), \qquad \forall \boldsymbol{s}' \in S, \\ 1 &= \sum_{\boldsymbol{s},\boldsymbol{a}} \mu^\pi(\boldsymbol{s})\pi(\boldsymbol{s},\boldsymbol{a}), \\ \pi(\boldsymbol{s},\boldsymbol{a}) &\geq 0, \qquad \forall \boldsymbol{s} \in S,\ \forall \boldsymbol{a} \in A. \end{aligned}$$

There are two approaches to solve this problem:

- *policy search* optimizes the primal formulation
- *value function-based approach* optimizes the dual formulation

Value Function Approaches
Using the Lagrangian function method with Lagrange multipliers $V^\pi(\boldsymbol{s}')$ and $\bar{R}$ it can be shown that for the optimal policy π^* it holds:

$$V^*(\boldsymbol{s}) = \max_{\boldsymbol{a}^*} \left[r(\boldsymbol{s},\boldsymbol{a}^*) - \bar{R} + \sum_{\boldsymbol{s}'} V^*(\boldsymbol{s}')T(\boldsymbol{s},\boldsymbol{a}^*,\boldsymbol{s}') \right] \tag{1}$$

Having $V^*(\boldsymbol{s}) = V^{\pi^*}(\boldsymbol{s})$. This statement is equivalent to the *Bellman principle of Optimality.* Thus, to maximize the average reward an agent has to perform an optimal action $\boldsymbol{a}^*$ and, subsequently, follow the optimal policy π^*.
$V^\pi(\boldsymbol{s})$ is the value function, it quantifies the amount of reward achievable from state $\boldsymbol{s}$ taking action a from policy π and, accordingly to Equation 1, it is defined as:

$$V^\pi(\boldsymbol{s}) = \sum_{\boldsymbol{a}} \pi(\boldsymbol{s},\boldsymbol{a})(r(\boldsymbol{s},\boldsymbol{a}) - (\bar{R} + \sum_{\boldsymbol{s}'} V^\pi(\boldsymbol{s}')T(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')).$$

Instead of the value function, many algorithms rely on the state-action value function, defined as:

$$Q^\pi(\boldsymbol{s},\boldsymbol{a}) = r(\boldsymbol{s},\boldsymbol{a}) - \bar{R} + \sum_{\boldsymbol{s}'} V^\pi(\boldsymbol{s}')T(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')$$

This function quantifies the reward achievable performing action $\boldsymbol{a}$ in state $\boldsymbol{s}$ and the following policy π. Equivalently to what has been shown before, the optimal state-action value function is:

$$\begin{aligned} Q^*(\boldsymbol{s},\boldsymbol{a}) &= r(\boldsymbol{s},\boldsymbol{a}) - \bar{R} + \sum_{\boldsymbol{s}'} V^*(\boldsymbol{s}')T(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}') \\ &= r(\boldsymbol{s},\boldsymbol{a}) - \bar{R} + \sum \boldsymbol{s}' \left(\max_{\boldsymbol{a}'} Q^*(\boldsymbol{s}',\boldsymbol{a}') \right) \end{aligned}$$

The optimal policy can be expressed through value function as

$$\pi^*(\boldsymbol{s}) = \arg\max_{\boldsymbol{a}} \left(r(\boldsymbol{s},\boldsymbol{a}) - \bar{R} + \sum_{\boldsymbol{s}'} V^*(\boldsymbol{s}')T(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}') \right) \tag{2}$$

or through the state-action value function as

$$\pi^*(\boldsymbol{s}) = \arg\max_{\boldsymbol{a}} Q^*(\boldsymbol{s},\boldsymbol{a}). \tag{3}$$

This second formulation has the advantage that it does not require the knowledge of transition probabilities.

In case of continuous state spaces determining an optimal solution is an optimization problem in itself. For discrete spaces π, $V^{\pi}(\boldsymbol{s})$ and $Q^{\pi}(\boldsymbol{s})$ could be represented by tables and the optimization is reduced to a table look-up. On the other hand, for large state spaces the table representation becomes intractable. Therefore it is necessary to use function approximation techniques.

It is now clear that, in order to find the optimal policy it is necessary to know $V^{*}(\boldsymbol{s})$ or $Q^{*}(\boldsymbol{s}, \boldsymbol{a})$ for every state. Three categories of methods can be used to estimate those functions:

- Dynamic Programming-Based Methods (*policy iteration* and *value iteration*)
- Monte Carlo Methods
- Temporal Difference Methods (TD(λ), SARSA, Q-learning)

Policy search
The other approach to the optimization problem is to optimize directly in the primal formulation. The general idea is to start with an initial guess for policy iteratively improve it until it converges to π^{*}.
This approach allows domain-appropriate prestructuring of the policy in an approximate form without changing the original problem. Moreover optimal policies have many fewer parameters than optimal value functions.
Policy search has been considered for long as a harder problem than value function based methods, but in robotics it has turned out to be more useful thanks to better scalability.
Most policy search methods optimize around existing policy π parametrized by a set of policy parameters $\boldsymbol{\theta}_i$, by computing changes in the policy parameters $\Delta\boldsymbol{\theta}_i$ that will increase the expected return and result in iterative updates of the form

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \Delta\boldsymbol{\theta}_i.$$

The computation of the policy update is the key step here and a variety of updates have been proposed. Usually policy search methods are categorized in:

- Black-box methods: those methods are general stochastic optimization algorithms using only the expected return of policies, estimated by sampling and do not leverage any of the internal structure of the RL problem.
- White-box methods: those methods, on the other hand, take advantage of some of some of additional structures within the RL domain.

Here we mainly focus on white-box methods.
In *gradient-based approaches* the update of the policy parameters are based on a hill-climb approach, that is following the gradient of the policy return J for a defined step-size α:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \alpha\nabla_{\boldsymbol{\theta}} J.$$

There are different options to compute $\nabla_{\boldsymbol{\theta}} J$:

- Estimation by linear regression. P perturbed policy parameters are evaluated to obtain an estimate of the gradient. We consider $\Delta\hat{J}_p \approx J(\boldsymbol{\theta}_i + \Delta\boldsymbol{\theta}_p) - J_{ref}$ where $p = [1 \dots P]$ are the individual perturbations and J_{ref} is a reference return.
  The gradient is now estimated from $\nabla_{\boldsymbol{\theta}} J \approx \left(\Delta\boldsymbol{\Theta}^T \Delta\boldsymbol{\Theta}\right)^{-1} \Delta\boldsymbol{\Theta}^T \Delta\hat{\mathbf{J}}$, where the matrix $\Delta\boldsymbol{\Theta}$ contains all stacked samples of the perturbations $\Delta\boldsymbol{\theta}_p$ and $\Delta\hat{\mathbf{J}}$ contains all the corresponding $\Delta\hat{J}_p$.
- REINFORCE methods. These methods rely on a gradient computation achieved through the formula $\nabla_{\boldsymbol{\theta}} J^{\boldsymbol{\theta}} = \mathbb{E}\left\{\sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi^{\boldsymbol{\theta}}(\boldsymbol{s}_t, \boldsymbol{a}_t) Q^{\pi}(\boldsymbol{s}_t, \boldsymbol{a}_t)\right\}$.

Besides gradient-based approaches there are different classes of methods which are based on expectation-maximization techniques, while some others rely on Dynamic Programming strategies.

2.1.3 Value Function Approaches vs Policy Search

Value Function:

+ If a complete optimal value function is known a global optimal solution follows by simply choosing actions to optimize it.

- Scales badly.
- Unstable under function approximations.

Policy Search:

+ Works well with continuous features.
+ Scales well.
+ May converge to local optima.

A class of algorithms called *actor-critic* methods attempt to incorporate advantages of both of the approaches.

2.1.4 Function Approximations

In RL function approximation techniques are useful when it is impossible to represent all states and actions, usually the approximation is based on sample data collected during the interaction with the environment. Function approximation can be employed to represent policies, value functions and forward models.
There are two kind of function approximation methods: *parametric* and *non-parametric*:

- A parametric function approximator uses a finite set of parameters with the goal of finding the parameters that make this approximation fit the data as close as possible. Examples include *linear basis functions* and *neural networks.*
- Non-parametric methods expand representational in relation to collected data and hence are not limited by the representation power of a chosen parametrization. An example is *Gaussian process regression.*

2.2 Challenges in Robot Reinforcement Learning

To apply RL to robotics we need to cope with some challenges. In this section, we briefly explain some of them.

2.2.1 Curse of Dimensionality

In high dimensional spaces an exponential explosion of states and actions occurs. For this reason the number of data and computations needed to cover the complete state-action space increases dramatically. Evaluating every state quickly becomes infeasible with growing dimensionality, even for discrete states. In robotics this problem is tackled by using a hierarchical task decomposition that shifts some complexity to a lower layer of functionality.

2.2.2 Curse of Real-World Samples

Robots inherently interact with the physical world. Hence, in robotics RL they suffers from most of the resulting real-world problems. For example, robot hardware is usually expensive, suffers from wear and tear, and requires careful maintenance. Thus, to apply RL in robotics, safe exploration becomes a key issue of the learning process. Furthermore, frequently, after a failure, human interaction is required in order to start a new episode.
For such reasons, real-world samples are expensive in terms of time, labour and, potentially, finances. In robotic RL, it is often considered to be more important to limit the real-world interaction time instead of limiting memory consumption or computational complexity. Thus, sample efficient algorithms that are able to learn from a small number of trials are essential.
Another problem that has to be faced in robotics is that often the effects taken actions are not instantaneous and they can be observed only several steps later. In this case Markov property is violated. One way of tackling this problem would be to include previous actions in the state, but this would increase even more the state space dimensionality. Another way could be to increase the duration of time steps even though in this way a lot of precision is lost.

2.2.3 Curse of Under-Modelling and Model Uncertainty

One way to offset the cost of real-world interaction is to use accurate models as simulators. In an ideal setting, this approach would render it possible to learn the behaviour in simulation and subsequently transfer it to the real robot. Unfortunately, creating a sufficiently accurate model of the robot and its environment is challenging and often requires very many data samples. As small model errors due to this under-modelling accumulate, the simulated robot can quickly diverge from the real-world system. For tasks where the system is self-stabilizing (that is, where the robot does not require active control to remain in a safe state or return to it), transferring policies often works well. In contrast, in unstable tasks small variations have drastic consequences.
In fact, tasks can often be learned better in the real world than in simulation due to complex mechanical interactions (including contacts and friction) that have proven difficult to model accurately.

2.2.4 Curse of Goal Specification

In RL, the desired behaviour is implicitly specified by the reward function. The learner must observe variance in the reward signal in order to be able to improve a policy.
In many domains, it seems natural to provide rewards only upon task achievement. This view results in an apparently simple, binary reward specification. However, a robot may receive such a reward so rarely that it is unlikely to ever succeed in the lifetime of a real-world system. Instead of relying on simpler binary rewards, we frequently need to include intermediate rewards in the scalar reward function to guide the learning process to a reasonable solution, a process known as *reward shaping.*
Reward shaping gives the system a notion of closeness to the desired behaviour instead of relying on a reward that only encodes success or failure.

2.3 Tractability through representation

Much of the success of RL methods has been due to the clever use of approximate representations. The need of such approximations is particularly pronounced in robotics, where table based representations are rarely scalable.

2.3.1 Smart State-Action Discretization

Reducing the dimensionality of states or actions by smart state-action discretization is a representational simplification that may enhance both policy search and value function-based methods. Different techniques are used to perform this discretization:

- Hand Crafted Discretization
- Learn from Data Discretization
- Meta-Actions
- Relational Representations

2.3.2 Value Function Approximation

A value function-based approach requires an accurate and robust but general function approximator that can capture the value function with sufficient precision while maintaining stability during learning. Function approximation has always been the key component that allowed value function methods to scale into interesting domains.
Unfortunately the max-operator used within the Bellman equation and temporal-difference updates can theoretically make most linear or non-linear approximation schemes unstable for either value iteration or policy iteration. Quite frequently such an unstable behaviour is also exhibited in practice. Linear function approximators are stable for policy evaluation, while non-linear function approximation (e.g., neural networks) can even diverge if just used for policy evaluation. In robot RL, the following function approximation schemes have been popular and successful:

- Physics inspired futures

- Neural Networks
- Generalize to Neighbouring Cells
- Local Models
- Gaussian Process Regression

2.3.3 Pre-structured policies

Policy search methods require a choice of policy representation that controls the complexity of representable policies to enhance learning speed. Policy search methods greatly benefit from employing an appropriate function approximation of the policy. As the next action picked by a policy depends on the current state and action, a policy can be seen as a closed-loop controller. However, especially for episodic RL tasks, sometimes open-loop policies (i.e., policies where the actions depend only on the time) can also be employed. Some popular methods for policy approximation are:

- Via Points & Splines
- Linear Models
- Motor Primitives
- Gaussian Mixture Models and Radial Basis Function Models
- Neural Networks
- Locally Linear Controllers
- Non-parametric Policies

2.4 Tractability Through Prior Knowledge

Prior knowledge can dramatically help to guide the learning process. Using prior knowledge it is possible to significantly reduce the search-space and, thus, speed up the learning process.

2.4.1 Prior Knowledge Through Demonstration

Frequently, it is useful to combine imitation learning and RL. This combination takes the name of *apprenticeship learning.*
The most dramatic benefit of demonstrations or a hand-crafted initial policy is they remove the need for global exploration of the policy or state-space of the RL problem. The student can improve by locally optimizing a policy knowing what states are important, making local optimization methods feasible. However, it has to be kept in mind that this way we can only find local optima close to the demonstration, that is, we rely on the demonstration to provide a good starting point.
A robot can be instructed through demonstration in two ways:

- Demonstration by a teacher (through mapping of human behaviour or through direct control by a human teacher)
- Hand-Crafted Policies

2.4.2 Prior Knowledge Through Task Structuring

Often a task can be decomposed hierarchically into basic components or into a sequence of increasingly difficult tasks. In both cases the complexity of the learning task is significantly reduced. Task structuring can be performed in two ways:

- Hierarchical RL
- Progressive Tasks

2.4.3 Direct Exploration with Prior Knowledge

As already discussed, balancing exploration and exploitation is an important consideration. Task knowledge can be employed to guide to robots curiosity to focus on regions that are novel and promising at the same time.

2.5 Tractability Through Models

Many robot RL problems can be made tractable by learning forward models, i.e., approximations of the transition dynamics based on data. Such model-based RL approaches jointly learn a model of the system with the value function or the policy and often allow for training with less interaction with the the real environment.
In robot RL, the learning step on the simulated system is often called *mental rehearsal*.

2.5.1 Core Issues and General Techniques in Mental Rehearsal

Experience collected in the real world can be used to learn a forward model from data. Such forward models allow training by interacting with a simulated environment. Only the resulting policy is subsequently transferred to the real environment. Model-based methods can make the learning process substantially more sample efficient.
Some key steps in Mental Rehearsal are:

- Dealing with simulation biases
- Distributions over Models for Simulation
- Sampling by Re-Using Random Numbers

2.5.2 Successful Learning Approaches with Forward Models

Model-based approaches rely on finding good policies using the learned model. Some methods directly obtain a new policy candidate from a forward model:

- Iterative Learning Control
- Locally Linear Quadratic Regulators
- Value Function Methods with Learned Models
- Policy Search with Learned Models

3 Playing Atari with Deep Reinforcement Learning

The rest of this essay will be focused on Q-learning algorithm with function approximation achieved through Deep Neural Networks.
Firstly, Q-learning has been applied to build an agent which is capable to achieve human-level performances when playing the Atari games.
The Atari domain is simpler to treat than real world robotic problems because the action space is discrete. When using neural network representation of the Q-function, it is a lot simpler to deal with discrete action spaces because at every step of the Q-learning algorithm a maximization over all the possible actions is required.
For the Atari task it is possible to learn an optimal policy using as input of the algorithm directly raw images from the Atari emulator without additional knowledge about the domain.

3.1 Recap of Q-learning

- E: environment (stochastic)
- $\boldsymbol{a}_t \in \mathcal{A} = \{1, \dots, K\}$: possible actions
- $\boldsymbol{x}_t \in \mathbb{R}^d$: raw image of the emulator observed by the agent at time t
- r_t: reward received at time t
- $\boldsymbol{s}_t$: state at time t defined as sequence of images and actions, $\boldsymbol{s}_t = (\boldsymbol{x}_1, \boldsymbol{a}_1, \boldsymbol{x}_2, \dots, \boldsymbol{a}_{t-1}, \boldsymbol{x}_t)$
- R_t: future discounted return at time t: $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$
- π: policy that maps sequences to actions.
- $Q^*(\boldsymbol{s}, \boldsymbol{a}) = \max_\pi \mathbb{E}[R_t | \boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a}, \pi]$: optimal action-value function, i.e. maximum reward achievable after seeing some sequence $\boldsymbol{s}$ and performing an action $\boldsymbol{a}$.
- *Bellman equation for Q^**: $Q^*(\boldsymbol{s}, \boldsymbol{a}) = \mathbb{E}_{\boldsymbol{s}' \sim \mathcal{E}}[r + \gamma \max_{\boldsymbol{a}'} Q^*(\boldsymbol{s}', \boldsymbol{a}') | \boldsymbol{s}, \boldsymbol{a}]$

Once Q^* is known the agent should pick at each step the action that maximizes it.

3.2 Deep Q-learning

The idea of Deep Q-learning [2] is to build an artificial neural network that approximates the optimal action-value function as $Q^*(\boldsymbol{s}, \boldsymbol{a}) \approx Q(\boldsymbol{s}, \boldsymbol{a} | \boldsymbol{\theta})$ and to train it while playing some episodes of the game.

3.2.1 Q-network

The neural network is trained performing stochastic gradient descent to find the parameters $\boldsymbol{\theta}$ that minimize the loss function $L_i(\boldsymbol{\theta}_i) = \mathbb{E}_{\boldsymbol{s},\boldsymbol{a}\ \rho(\cdot)}[(y_i - Q(\boldsymbol{s}, \boldsymbol{a} | \boldsymbol{\theta}_i))^2]$, where

- $y_i = \mathbb{E}_{\boldsymbol{s}' \sim E}[r + \gamma \max_{\boldsymbol{a}'} Q(\boldsymbol{s}', \boldsymbol{a}' | \boldsymbol{\theta}_{i-1}) | \boldsymbol{s}, \boldsymbol{a}]$ is the target for the i-th iteration
- $\rho(\boldsymbol{s}, \boldsymbol{a})$ is the behaviour distribution, a probability distribution over sequences and actions

3.2.2 Experience Replay

Experience replay allows to take into account transitions experienced many time steps before. Agent's experience $\boldsymbol{e}_t = (\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$ is stored in a dataset $\mathcal{D} = (\boldsymbol{e}_1, \dots, \boldsymbol{e}_N)$.
During the inner loop of the algorithm it is then possible to apply minibatch updates to samples of experience $\boldsymbol{e} \sim D$ drawn at random from the stored samples.
After performing experience replay, the agent selects and executes an action according to an ϵ-greedy policy that means that the next action is

$$\boldsymbol{a} = \begin{cases} \pi(s) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

Since using sequences of arbitrary length as inputs for a neural network can be difficult, the Q-function instead works on fixed length representations of histories produced by a function $\boldsymbol{\phi}$.

This approach has several advantages:

- each step of the approach is potentially used in many weight updates, which allows for greater data efficiency
- randomizing the samples breaks correlations between samples and therefore reduces the variance of updates
- the behaviour distribution is averaged over many of its previous stages, smoothing out learning and avoiding oscillations or divergence in the parameters.

The complete pseudo-code for the algorithm is the following:

Algorithm 1 Deep Q-learning with Experience Replay [2]

```
Initialise replay memory $\mathcal{D}$ to capacity $N$
Initialise action-value function Q
for episode = 1, M do
    Initialise sequence $\boldsymbol{s}_1 = \{\boldsymbol{x}_1\}$ and preprocessed sequence $\boldsymbol{\phi_1} = \boldsymbol{\phi}(\boldsymbol{s}_1)$
    for t=1,T do
        With probability $\epsilon$ select a random action $\boldsymbol{a}_t$
        Otherwise select $\boldsymbol{a}_t = \arg\max_{\boldsymbol{a}} Q(\boldsymbol{\phi}(\boldsymbol{s}_t), \boldsymbol{a}|\boldsymbol{\theta})$
        Execute action $\boldsymbol{a}_t$ in emulator and observe reward $r_t$ and image $\boldsymbol{x}_{t+1}$
        Set $\boldsymbol{s}_{t+1} = (\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{x}_{t+1})$ and preprocess $\boldsymbol{\phi}_{t+1} = \boldsymbol{\phi}(\boldsymbol{s}_{t+1})$
        Store transitions $(\boldsymbol{\phi}_t, \boldsymbol{a}_t, r_t, \boldsymbol{\phi}_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\boldsymbol{\phi}_j, \boldsymbol{a}_j, r_j, \boldsymbol{\phi}_{j+1})$ from $\mathcal{D}$
        set $y_j = \begin{cases} r_j & \text{for terminal } \boldsymbol{\phi_{j+1}} \\ r_j + \max_{\boldsymbol{a}'} Q(\boldsymbol{\phi}_{j+1}, \boldsymbol{a}'|\boldsymbol{\theta}) & \text{for terminal } \boldsymbol{\phi_{j+1}} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\boldsymbol{\phi}_j, \boldsymbol{a}_j|\boldsymbol{\theta}))$
    end for
end for
```

3.2.3 Model Architecture

Atari frames, which are 210 x 164 pixel images with a 128 color palette, are reduced and cropped to 84 x 84 greyscale images, in order to reduce the computational cost.
The function ϕ applies this preprocessing to the last four frames of a history and stacks them to produce the input for the Q-function.

The ANN built for this algorithm takes states as inputs and produces Q value for every action (in the experiments the number of actions varies from 4 to 18).
The exact architecture is the following:

- input: 84 x 84 x 4 image produced by ϕ
- the first hidden layer convolves 16 8 x 8 filters with stride 4 with the input image and applies a rectifier nonlinearity
- the second hidden layer convolves 32 4 x 4 filters with stride 2 again followed by a rectifier nonlinearity.
- the final hidden layer is fully connected and consists of 256 rectifier units
- the output layer is a fully-connected linear layer with a single output for each valid action

	B.Rider	**Breakout**	**Pong**	**Seaquest**	**S.Invaders**
Random	354	1.2	-20.4	110	179
Sarsa	996	5.2	-19	665	271
Contingency	1743	6	-17	723	268
DQN	**4092**	**168**	**20**	**1705**	**581**
Human	7456	31	-3	28010	3690

Tab. 1: Average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps.[2]

3.2.4 Target Network

The target network, with parameters $\boldsymbol{\theta}^{-}$, is the same as the on-line network except that its parameters are copied every τ steps from the on-line network, so that then $\boldsymbol{\theta}_t^{-} = \boldsymbol{\theta}_t$, and kept fixed on all other steps. The target used by Deep Q-learning is then

$$y_i = \mathbb{E}_{s' \sim E}[r + \gamma \max_{\boldsymbol{a}'} Q(\boldsymbol{s}', \boldsymbol{a}' | \boldsymbol{\theta}_{i-1}^{-}) | \boldsymbol{s}, \boldsymbol{a}].$$

3.2.5 Results

Table 1 compares average total reward obtained [2] for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. Deep Q-learning always overperforms all of the previous methods and in the cases of Breakout, Enduro and Pong it obtains higher rewards than an expert human player. On the other hand in the other cases Deep Q-learning does not reach human performances.

3.2.6 Overoptimism Due to Estimation Errors

Q-learning is one of the most popular Reinforcement Learning algorithms, but it is known to sometimes learn unrealistically high action values because it includes a maximization step over estimated action values, which tends to prefer overestimated to underestimated values. This overoptimism can result in non-optimal or unstable policies, which do not lead to high rewards.

3.3 Double Deep Q-learning

The purpose of Double Deep Q-learning is to eliminate the overoptimisms in the approximated Q function. It seems clear that a more precise estimate of Q will generate higher rewards.

3.3.1 Double Q-learning

The max operator in standard Q-learning and Deep Q-learning uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, we can decouple the selection from the evaluation.
In the original Double Q-learning algorithm [3], two value functions are learned by assigning each experience randomly to update one of the two value functions, such that there are two sets of weights, $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. For a clear comparison, we can first untangle the selection and evaluation in Q-learning and rewrite its target as

$$y_i = \mathbb{E}_{\boldsymbol{s}' \sim \mathcal{E}}[r + \gamma Q(\boldsymbol{s}', \arg\max_{\boldsymbol{a}'} Q(\boldsymbol{s}', \boldsymbol{a} | \boldsymbol{\theta}_{i-1}) | \boldsymbol{\theta}_{t-1}) | \boldsymbol{s}, \boldsymbol{a}].$$

Now the target can be rewritten as

$$y_i = \mathbb{E}_{\boldsymbol{s}' \sim \mathcal{E}}[r + \gamma Q(\boldsymbol{s}', \arg\max_{\boldsymbol{a}'} Q(\boldsymbol{s}', \boldsymbol{a} | \boldsymbol{\theta}_{i-1}) | \boldsymbol{\theta}_{t-1}') | \boldsymbol{s}, \boldsymbol{a}].$$

Notice that the selection of the action, in the argmax, is still due to the on-line weights $\boldsymbol{\theta}_t$. This means that, as in Q- learning, we are still estimating the value of the greedy policy according to the current values, as defined by $\boldsymbol{\theta}_t$. However, we use the second set of weights $\boldsymbol{\theta}_t'$ to fairly evaluate the value of this policy. This second set of weights can be updated symmetrically by switching the roles of $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$.

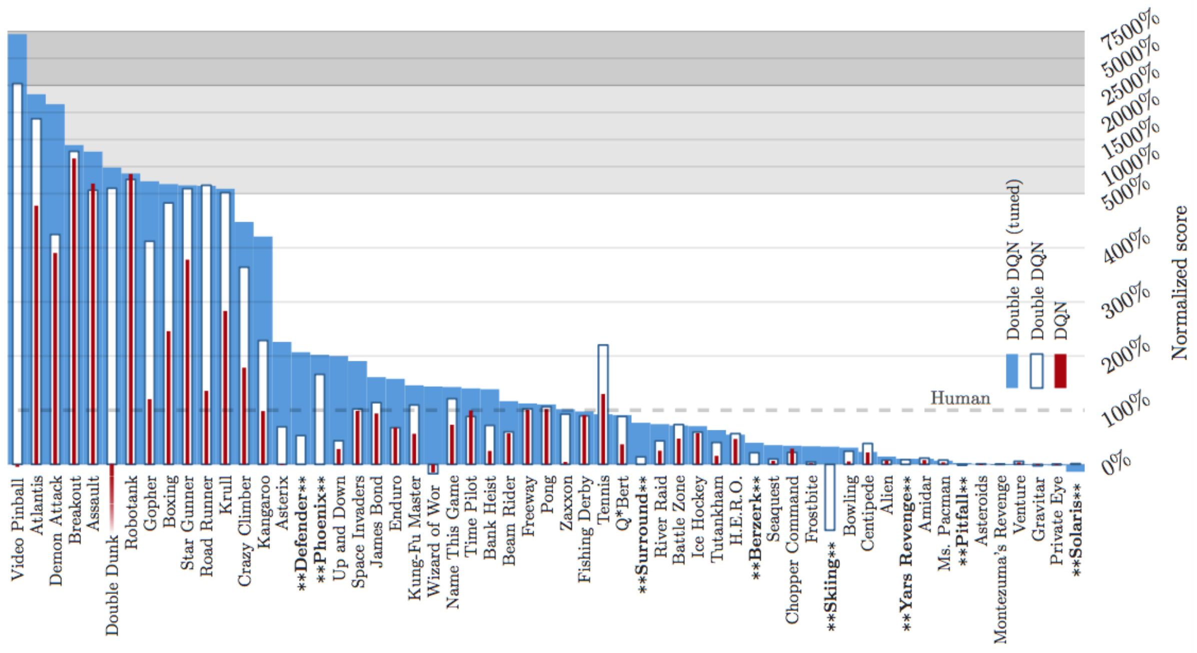


Fig. 1: Normalized scores on 57 Atari games, tested for 100 episodes per game with human starts.[3]

3.3.2 Double Deep Q-learning

An efficient approach is to evaluate the greedy policy according to the on-line network, but using the target network to estimate its value. The resulting algorithm is known as Double Deep Q-learning [3]. Its update is the same as for Deep Q-learning, but replacing the target with

$$y_i = \mathbb{E}_{\boldsymbol{s}' \sim \mathcal{E}}[r + \gamma Q(\boldsymbol{s}', \arg\max_{\boldsymbol{a}'} Q(\boldsymbol{s}', \boldsymbol{a}|\boldsymbol{\theta}_{i-1})|\boldsymbol{\theta}_{i-1}^{-})|\boldsymbol{s}, \boldsymbol{a}].$$

In comparison to Double Q-learning, the weights of the second network $\boldsymbol{\theta}_i$ are replaced with the weights of the target network $\boldsymbol{\theta}_i^{-}$ for the evaluation of the current greedy policy. The update to the target network stays unchanged from Deep Q-learning, and remains a periodic copy of the on-line network.

3.3.3 Results

Double Deep Q-learning has been tested on several Atari games [3]. It has been shown that in all of the games it reduces the overestimations and in almost all of them it outperforms simple Deep Q-learning as shown in figure 1.

3.4 Prioritized Experience Replay

In simple *experience replay* the minibatch is built by sampling the transitions $\boldsymbol{e}_t$ uniformly at random. On the other hand some transitions are more useful for learning than others. The learning process can be made more efficient and effective by picking more often important transitions. Thus it is useful to weight each transition in memory differently so that it is possible to sample experiences from the weights distribution.

3.4.1 Prioritizing with TD Error

The central component of prioritized replay [4] is the criterion by which the importance of each transition is measured. One idealised criterion would be the amount the RL agent can learn from a transition in its current state (expected learning progress). While this measure is not directly accessible, a reasonable proxy is the magnitude of a transition's TD error $\delta_i = y_i - Q(\boldsymbol{\phi}_i, \boldsymbol{a}_i|\boldsymbol{\theta})$, which indicates how "surprising" or unexpected the transition is: specifically, how far the value is from its next-step bootstrap estimate. This is particularly suitable for incremental, online RL algorithms, such as Q-learning, that already compute the TD-error and update the parameters in proportion to δ.

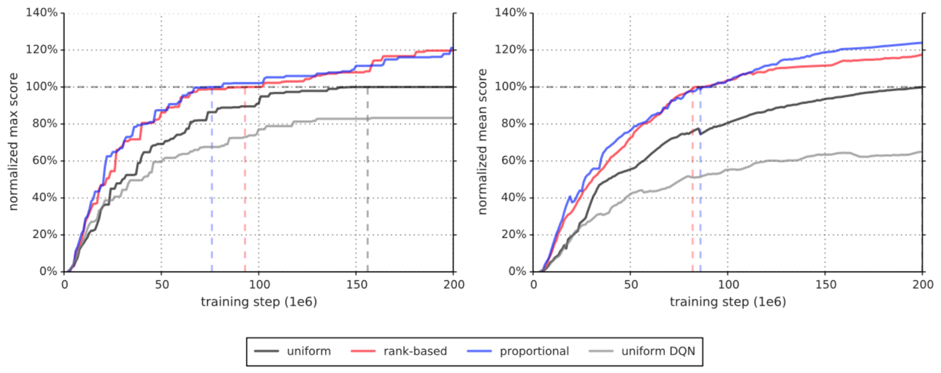


Fig. 2: Summary plots of learning speeds. Left: median over 57 games of the maximum baseline-normalized score achieved so far. Right: Similar to the left, but using the mean instead of maximum, which captures cumulative performance rather than peak performance.[4]

3.4.2 Stochastic Prioritization

It is possible introduce a stochastic sampling method that interpolates between purely greedy prioritization and uniform random sampling.
The probability of sampling a transition i is defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where $p_i > 0$ is the priority of transition i. The exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case.
Two variants for the priorities are proposed:

- $p_i = |\delta_i| + \epsilon$
- $p_i = \frac{1}{rank(i)}$ where $rank(i)$ is the rank of the transition i when the replay memory is sorted according to $|\delta_i|$

Both of the variants give speed-up in the learning process and depending on the application one can be better than the other.

3.4.3 Annealing the Bias

The estimation of the expected value with stochastic updates relies on those updates corresponding to the same distribution as its expectation. Prioritized replay introduces bias because it changes this distribution. This bias can be corrected using importance-sampling (IS) weights

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

that fully compensate for the non-uniform probabilities $P(i)$ if $\beta = 1$. These weights can be folded into the Q-learning update by using $\omega_i\delta_i$ instead of δ_i. For stability reasons the weights are always normalized by $\frac{1}{\max_i \omega_i}$.
In typical RL scenarios, the unbiased nature of the updates is most important near convergence at the end of training. We therefore exploit the flexibility of annealing the amount of importance-sampling correction over time, by defining a schedule on the exponent β that reaches 1 only at the end of learning.

3.4.4 Results

As shown in the figure 2 Double Deep Q-learning with Prioritized Experience Replay performs better than both Deep Q-learning and simple Double Deep Q-learning algorithms. The plots show the median of mean and maximum score over 57 different Atari games [4].

4 Deep Q-learning for Robotic Tasks

Robotics applications are more complicated than Atari games because state and action spaces become continuous. For this reason feature and policy engineering are always required in order to apply model-free RL. On the other hand it would be preferable to have a version of Q-learning able to deal with continuous states. Q-learning is interesting because in principle it is able to solve tasks without any knowledge of the domain (as seen in Atari application).
On the other hand, model-based RL could be very useful in robotics in to improve data efficiency. The biggest drawback of model-based RL is that inaccuracies on the model will reflect in accuracy of the final optimal policy.

One of the main issues to deal with in order to apply Q-learning to the continuous case is that at each iteration an optimization of a complicated non-linear function is needed to find the optimal action $\boldsymbol{\mu} = \arg\max_{\mathbf{a}'} Q(\mathbf{s}, \mathbf{a}', \boldsymbol{\theta})$.
In the discrete case the output of the Q-network was a real value for each possible action, therefore this maximization problem was easy to solve.

4.1 Deep Deterministic Policy Gradient

The first algorithm that we analyse to deal with robotics task is actually not a Q-learning algorithm. DDPG [5] is, instead, an actor-critic method. The algorithms that belong to this class usually use two function approximators, one for the policy $\boldsymbol{\mu}(\boldsymbol{s}|\boldsymbol{\theta}^{\mu})$ (the actor) and one for the value function $Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^{Q})$ (the critic). The actor chooses which action to take when being on a particular state, while the critic evaluates the goodness of this action.
DDPG uses 2 neural networks to approximate the two functions. At each time step of every episode the parameters of the critic $\boldsymbol{\theta}^{Q}$ are updated in the usual way minimizing the loss

$$L(\boldsymbol{\theta}^{Q}) = \mathbb{E}_{\boldsymbol{s}_t \sim \rho^{\beta}, \boldsymbol{a}_t \sim \beta, r_t \sim E}[\big(Q(\boldsymbol{s}_t, \boldsymbol{a}_t|\boldsymbol{\theta}^{Q}) - y_t\big)^2]$$

with

$$y_t = r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma Q(\boldsymbol{s}_{t+1}, \boldsymbol{\mu}(\boldsymbol{s}_{t+1})|\boldsymbol{\theta}^{\boldsymbol{Q}})$$

Instead, the parameters of the actor $\boldsymbol{\theta}^{\mu}$ are updated following the gradient of the total reward $J = \mathbb{E}_{r_i, \boldsymbol{s}_i \sim E, \boldsymbol{a}_{i>t} \sim \pi}[R_1]$ that is:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}^{\mu}} J &\approx \mathbb{E}_{\boldsymbol{s}_t \sim \rho^{\beta}}[\nabla_{\boldsymbol{\theta}^{\mu}} Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^{\mu})|_{\boldsymbol{s}=\boldsymbol{s}_t, \boldsymbol{a}=\mu(\boldsymbol{s}_t|\boldsymbol{\theta}^{\mu})}] \\ &= \mathbb{E}_{\boldsymbol{s}_t \sim \rho^{\beta}}[\nabla_{\boldsymbol{a}} Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^{\mu})|_{\boldsymbol{s}=\boldsymbol{s}_t, \boldsymbol{a}=\mu(\boldsymbol{s}_t)} \nabla_{\boldsymbol{\theta}^{\mu}} \mu(\boldsymbol{s}|\boldsymbol{\theta}^{\mu})|_{\boldsymbol{s}=\boldsymbol{s}_t}] \end{aligned}$$

Similarly to DQN transitions are saved in a *replay memory* and minibatch of experience are sampled at random from this memory buffer.
Moreover, this algorithm uses target networks to make the updates of $\boldsymbol{\theta}^{Q}$ and $\boldsymbol{\theta}^{\mu}$ more smooth.
In the algorithm 2 the pseudocode for DDPG method is shown.

4.2 Continuous Q-learning with Normalized Advantage Functions

The idea of NAF algorithm [6] is to give a representation of the Q-function from which it is simple to retrieve $\boldsymbol{\mu}$. In order to achieve this representation we first define the advantage function

$$A^{\pi}(\boldsymbol{s}_t, \boldsymbol{a}_t) = Q^{\pi}(\boldsymbol{s}_t, \boldsymbol{a}_t) - V^{\pi}(\boldsymbol{s}_t),$$

which essentially quantifies the advantage (or disadvantage) in reward when performing the action $\boldsymbol{u}_t$ with respect to the average reward achievable from state $\boldsymbol{x}_t$. Thus, through this decomposition the parameters of the Q-network can be split as follows:

$$\begin{aligned} Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^{Q}) &= A(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^{A}) + V(\boldsymbol{s}|\boldsymbol{\theta}^{\boldsymbol{V}}) \\ A(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^{A}) &= \frac{1}{2}(\boldsymbol{a} - \boldsymbol{\mu}(\boldsymbol{s}|\boldsymbol{\theta}^{\mu})^T \boldsymbol{P}(\boldsymbol{s}|\boldsymbol{\theta}^{P})(\boldsymbol{a} - \boldsymbol{\mu}(\boldsymbol{s}|\boldsymbol{\theta}^{\mu})), \end{aligned}$$

with

$$\boldsymbol{P}(\boldsymbol{s}|\boldsymbol{\theta}^{P}) = \boldsymbol{L}(\boldsymbol{s}|\boldsymbol{\theta}^{P})\boldsymbol{L}(\boldsymbol{s}|\boldsymbol{\theta}^{P})^T,$$

Algorithm 2 Deep Deterministic Policy Gradient [5]

```
Randomly initialize critic network $Q(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta}^Q)$ and actor $\boldsymbol{\mu}(\boldsymbol{s}|\theta^\mu)$ with weights $\boldsymbol{\theta}^Q$ and $\boldsymbol{\theta}^\mu$
Initialize target networks $Q'$ and $\boldsymbol{\mu}'$ with weights $\boldsymbol{\theta}^{Q'} \leftarrow \boldsymbol{\theta}^Q$ and $\boldsymbol{\theta}^{\mu'} \leftarrow \boldsymbol{\theta}^\mu$
Initialize replay buffer $R \leftarrow \emptyset$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $\boldsymbol{s}_1$
    for t=1,T do
        Select action $\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t|\boldsymbol{\theta}^\mu) + \mathcal{N}_t$ according to current policy and exploration noise
        Execute action $\boldsymbol{a}_t$ and observe reward $r_t$ and new state $\boldsymbol{s}_{t+1}$
        Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$ in $R$
        Sample random minibatch of transitions $(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{\phi}_{i+1})$ from $\mathcal{R}$
         set $y_i = r_i + Q(\boldsymbol{s}_{i+1}, \boldsymbol{\mu}'(\boldsymbol{s}_{i+1}|\boldsymbol{\theta}^\mu))$
        Update the critic by minimizing the loss $L = \sum_i((Q(\boldsymbol{s}_i, \boldsymbol{a}_i|\boldsymbol{\theta}^Q) - y_i)^2)$
        Update the actor policy using the sampled policy gradient
            $$\nabla_{\boldsymbol{\theta}^\mu} J = \frac{1}{N}\sum_i \nabla_{\boldsymbol{a}} Q(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta}^\mu)|_{\boldsymbol{s}=\boldsymbol{s}_i,\boldsymbol{a}=\mu(\boldsymbol{s}_i)} \nabla_{\boldsymbol{\theta}^\mu} \mu(\boldsymbol{s}|\boldsymbol{\theta}^\mu)|_{\boldsymbol{s}=\boldsymbol{s}_i}$$
        Update the target networks: $\boldsymbol{\theta}^{Q'} \leftarrow \tau\boldsymbol{\theta}^Q + (1-\tau)\boldsymbol{\theta}^{Q'}$ and $\boldsymbol{\theta}^{\mu'} \leftarrow \tau\boldsymbol{\theta}^\mu + (1-\tau)\boldsymbol{\theta}^{\mu'}$
    end for
end for
```

where $\boldsymbol{L}(\boldsymbol{x}|\boldsymbol{\theta}^P)$ is a lower-triangular matrix whose entries come from a linear output layer of a neural network, with the diagonal terms exponentiated.
While this representation is more restrictive than a general neural network function, since the Q-function is quadratic in $\boldsymbol{u}$, the action that maximizes the Q-function is always $\boldsymbol{\mu}(\boldsymbol{x}|\boldsymbol{\theta}^\mu)$. Thus we can now use an approach similar to Deep Q-learning and formulate algorithm 3:

Algorithm 3 Continuous Q-learning with NAF [6]

```
Randomly initialize $Q$ network $Q(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta}^Q)$ with weights $\boldsymbol{\theta}^Q$
Initialize target network $Q'$with weights $\boldsymbol{\theta}^{Q'} \leftarrow \boldsymbol{\theta}^Q$
Initialize replay buffer $R \leftarrow \emptyset$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $\boldsymbol{s}_1 \sim p(\boldsymbol{s}_1)$
    for t=1,T do
        Select action $\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t|\boldsymbol{\theta}^\mu) + \mathcal{N}_t$ according to current policy and exploration noise
        Execute action $\boldsymbol{a}_t$ and observe reward $r_t$ and new state $\boldsymbol{s}_{t+1}$
        Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$ in $R$
        for iteration= $1, I$ do
            Sample random minibatch of $m$ transitions from $(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{\phi}_{i+1})$ from $R$
             set $y_i = r_i + \gamma V'(\boldsymbol{s}_{i+1}|\boldsymbol{\theta}^{Q'})$
            Update $\boldsymbol{\theta}^Q$ by minimizing the loss $L = \frac{1}{N}\sum_i \left(Q(\boldsymbol{s}_i, \boldsymbol{a}_i|\boldsymbol{\theta}^Q) - y_i\right)^2$
            Update the target network: $\boldsymbol{\theta}^{Q'} \leftarrow \tau\boldsymbol{\theta}^Q + (1-\tau)\boldsymbol{\theta}^{Q'}$
        end for
    end for
end for
```

In general A, does not need to be quadratic, and exploring other parametric forms such as multimodal distributions is an interesting avenue for future work.

4.2.1 Results

To evaluate the results of NAF comparing it to DDPG, they have been run on a set of robotic tasks using the MuJoCo simulator [6]. As shown in figure 3 for three-joint reacher and peg insertion tasks,

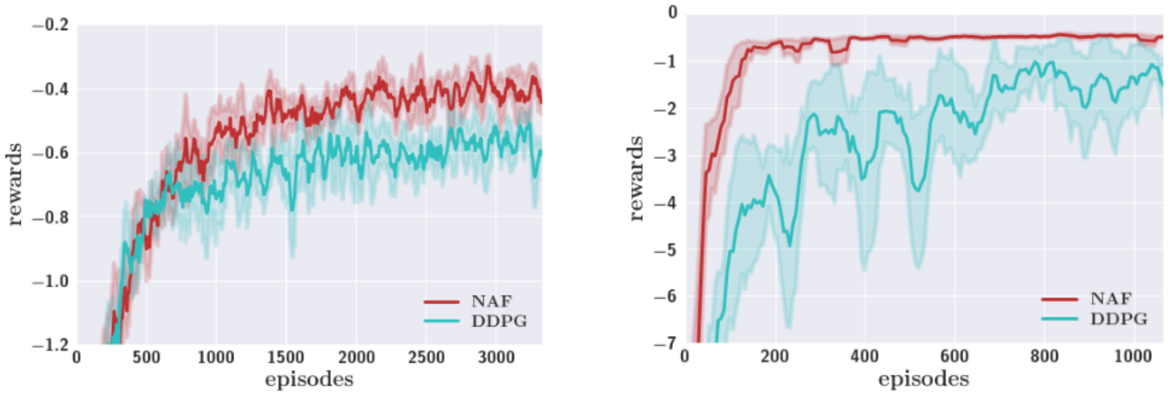


Fig. 3: NAF vs DDPG results on three-joint reacher and peg insertion. On reacher, the DDPG policy continuously fluctuates the tip around the target, while NAF stabilizes well at the target.[6]

NAF leads the agent to earn higher rewards and it needs less training episodes to converge than DDPG.

4.3 Accelerating Learning with Imagination Rollouts

On physical robots it is very important to reduce the number of episodes needed to learn a good Q function. This is because off-policy methods like Q-learning choose the new action maximizing Q values and inaccurate Q values can lead the robot to catastrophic behaviour causing failures and hardware damages. A first approach for better exploitation of the episodes could be to generate good exploratory behaviours using planning or trajectory optimization. To this end one can employ the iLQG algorithm.

4.3.1 The iLQG algorithm

When the transition probabilities $P(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$ are not known, they can be approximated with some learned model $\hat{P}(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$. The iLQG algorithm optimizes trajectories by iteratively constructing locally optimal linear feedback controllers under a local linearization of the dynamics $\hat{P}(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t) = \mathcal{N}(\boldsymbol{f}_{\boldsymbol{s}t}\boldsymbol{s}_t + \boldsymbol{f}_{\boldsymbol{a}t}\boldsymbol{a}_t, \boldsymbol{F}_t)$ and a quadratic expansion of the reward $r(\boldsymbol{s}_t, \boldsymbol{a}_t)$.
Under linear dynamics and quadratic rewards, the action-value function $Q(\boldsymbol{s}_t, \boldsymbol{a}_t)$ and the value function $V(\boldsymbol{s}_t)$ are locally quadratic and can be computed by linear programming.
The optimal policy can be derived analytically from the quadratic $Q(\boldsymbol{s}_t, \boldsymbol{a}_t)$ and $V(\boldsymbol{s}_t)$ functions and corresponds to a linear feedback controller $\boldsymbol{g}(\boldsymbol{s}_t) = \hat{\boldsymbol{a}}_t + \boldsymbol{k}_t + \boldsymbol{K}_t(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t)$ where $\boldsymbol{k}_t$ is an open-loop term, $\boldsymbol{K}_t$ is the closed loop feedback matrix and $\hat{\boldsymbol{x}}_t$ and $\hat{\boldsymbol{u}}_t$ are the states and actions of the nominal trajectory, which is the average trajectory of the controller.
When the dynamics are not known a particularly effective way to use iLQG is to combine it whith learned time-varying linear models $\hat{P}(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$. In this variant of the algorithm trajectories are sampled from the controller previously derived and used to fit time-varying linear dynamics with linear regression. These dynamics are then used with with iLQG to obtain a new controller.
This approach requires a few additional assumptions: namely, it requires the initial state to be either deterministic or low-variance Gaussian, and it requires the states and actions to be all continuous.
The model itself is given by $\hat{P}(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t) = \mathcal{N}(\boldsymbol{F}_t[\boldsymbol{s}_t; \boldsymbol{a}_t] + \boldsymbol{f}_t, \boldsymbol{N}_t)$. Every n episodes, we refit the parameters $\boldsymbol{F_t}$, $\boldsymbol{f_t}$ and $\boldsymbol{N_t}$ by fitting a Gaussian distribution at each time step to the vectors $[\boldsymbol{x}_t^i, \boldsymbol{u}_t^i, \boldsymbol{x}_{t+1}^i]$ where i indicates the sample index, and conditioning this Gaussian on $[\boldsymbol{x}_t; \boldsymbol{u}_t]$ to obtain the parameters of the linear-Gaussian dynamics at the step.
Although this approach introduces additional assumptions beyond the standard model-free RL setting, it produces impressive gains in sample efficiency on tasks where it can be applied.

4.3.2 Imagination Rollouts

Unfortunately applying directly iLQG exploration to NAF does not give any benefit to the learning process. It can be inferred that this method is not effective because the agent needs to explore both good and bad actions to understand the true Q-values, while iLQG makes the agent explore only good

actions.
One way to overcome this problem could be to use noisy on-policy actions, but this is inconvenient form a practical point of view because it could lead to undesirable behaviours that lead to robot damage.
Nevertheless iLQG can still be employed but it is necessary to change the NAF algorithm. One good way to do this is to generate synthetic on-policy trajectories under a linear model. Adding these synthetic samples, called *imagination rollouts*, to the replay buffer effectively augments the amount of experience available for Q-learning. The particular approach used is to perform rollouts in the real world using a mixture of planned iLQG trajectories and off-policy trajectories, and then generate additional synthetic on-policy rollouts using the learned model from each state visited along the real-world rollouts. The resulting algorithm is the following:

Algorithm 4 Imagination Rollouts with Fitted Dynamics and optional iLQG Exploration [6]

```
Randomly initialize $Q$ network $Q(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta}^Q)$ with weights $\boldsymbol{\theta}^Q$
Initialize target network $Q'$with weights $\boldsymbol{\theta}^{Q'} \leftarrow \boldsymbol{\theta}^Q$
Initialize replay buffer $R \leftarrow$ and fictional buffer $R_f \leftarrow \emptyset$
Initialize additional buffers $B \leftarrow$ and $B_{old} \leftarrow \emptyset$ with size $nT$
Initialize fitted dynamics model $\mathcal{M} \leftarrow \emptyset$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $\boldsymbol{s}_1$
    Select $\boldsymbol{\mu}'(\boldsymbol{s},t)$ from $\{\boldsymbol{\mu}(\boldsymbol{s}|\boldsymbol{\theta}^\mu), \pi_t^{iLQG}(\boldsymbol{s}_t|\boldsymbol{a}_t)\}$ with probabilities $\{p, 1-p\}$
    for t=1,T do
        Select action $\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t|\boldsymbol{\theta}^\mu) + \mathcal{N}_t$
        Execute action $\boldsymbol{a}_t$ and observe reward $r_t$ and new state $\boldsymbol{s}_{t+1}$
        Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$ in $R$ and $B$
        if  mod $(episode \cdot T + t, m) = 0$ and $\mathcal{M} \neq \emptyset$ then
            sample $m(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}_{i+1}, i)$ from $B_{old}$
            Use $\mathcal{M}$ to simulate l steps from each sample
            Store all fictional transitions in $R_f$
        end if
        Sample random minibatch of $m$ transitions $I \cdot l$ times from $R_f$ and $I$ times from $R$
        and update $\boldsymbol{\theta}^Q$,$\boldsymbol{\theta}^{Q'}$ as in algorithm 3 per minibatch
    end for
    if $B_f$ is full then then
        $\mathcal{M} \leftarrow$ FitLocalLinearDynamics($B_f$)
        $\pi^{iLQG} \leftarrow$ iLQG_OneStep($B_f, \mathcal{M}$)
        $B_{old} \leftarrow B_f$, $B_f \leftarrow \emptyset$
    end if
end for
```

As the Q-function becomes more accurate, on-policy behavior tends to outperform model-based controllers. We therefore propose to switch off imagination rollouts after a given number of iterations. In this framework, the imagination rollouts can be thought of as an inexpensive way to pretrain the Q-function, such that fine-tuning using real world experience can quickly converge to an optimal solution.

4.3.3 Results

The version of NAF algorithm with Imagination Rollouts has been tested again on some benchmark tasks on the MuJoCo simulator[6]. In figure 4 it's possible to see three different situations. The first image shows the results of the algorithm in terms of reward when it has been applied to solve the single-target reacher task but instead of the fitted model it used the real model of the robot for the dynamics. In this case the algorithm was also run without Imagination Rollouts, but simply with the iLQG trajectories. The results show that Imagination Rollouts from the dynamical model improve a lot the results. The reason is probably that the iLQG trajectories alone do not lead the agent to try also bad actions, so the experiences are less effective. The middle image in figure 4 image shows the results when the full algorithm, with fitted dynamics, has been applied to the same task as before. The same considerations made before apply to this case. Furthermore it shows how trying to fit the dynamical model with a neural network does not work. This failure is probably due to the fact that the sampled transitions are not enough to fit such a complicated model as a neural network. The last image shows the results when the algorithm has been applied to the single target gripping task. Also in this case we can confirm that iLQG trajectories themselves are not very effective, while Imagination Rollouts improve the solution.

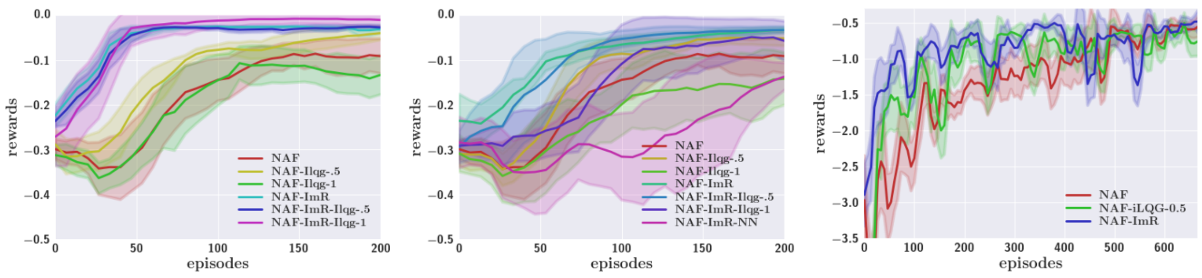


Fig. 4: Results on NAF with iLQG-guided exploration and imagination rollouts (first) using true dynamics (second and third) using fitted dynamics. “ImR”denotes using the imagination rollout with l = 10 steps on the reacher and l = 5 steps on the gripper. “iLQG-x”indicates mixing x fraction of iLQG episodes. Fitted dynamics uses time-varying linear models with sample size n = 5, except “-NN”which fits a neural network to global dynamics.[6]

4.4 Accelerating Learning with Asynchronous Off-Policy Updates

A different approach to speed up the learning process in NAF algorithm is to parallelize the learning process. This approach is particularly suitable for pratical robotic systems where experience is more time consuming than neural network backward passes.

4.4.1 Parallelization of NAF

In asynchronous NAF[7], the learner thread is separated from the experience collecting worker threads. The learner thread uses the replay buffer to perform asynchronous updates to the deep neural network Q-function approximator. This thread runs on a central server, and dispatches updated policy parameters to each of the worker threads. The experience collecting worker threads run on the individual robots, and send the observation, action, and reward for each time step to the central server to append to the replay buffer. This decoupling between the training and the collecting threads allows the controllers on each of the robots to run in real time, without experiencing delays due to the computational cost of backpropagation through the network. Furthermore, it makes it straightforward to parallelize experience collection across multiple robots simply by adding additional worker threads. While the trainer thread keeps training from the centralized replay buffer, the collector threads sync their policy parameters with the trainer thread at the beginning of each episode, execute commands on the robots, and push experience into the buffer. The asynchronous learning procedure is summarized in algorithm 5:

4.4.2 Results

As expected Asynchronous Off-Policy Updates improves a lot both the learning time and the quality of the solution. In the literature [7] many results that show this effectiveness both in real world and in simulated experiments can be found. Among all of those the ones that seem more relevant are the ones made in real world on a target-reaching task and on a door opening task. The robots used in this case is the 7 DOF arm shown in figure 5. Figure 6 shows the learning results for these two task. The left plot shows the curves for success rate using 1, 2 or 4 workers for the target reaching plot. In this case it is easy to see how the success rate after 500’000 is improved to almost 100%. On the right plot, instead it is possible to see the learning curves for real-world door opening. Learning with two workers significantly outperforms the single worker, and achieves a 100% success rate in under 500,000 update steps, corresponding to about 2.5 hours of real time.

Algorithm 5 Asynchronous NAF - N collector threads and 1 trainer thread [7]

```
// trainer thread
Randomly initialize $Q$ network $Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^Q)$ with weights $\boldsymbol{\theta}^Q = \{\boldsymbol{\theta}^\mu, \boldsymbol{\theta}^P, \boldsymbol{\theta}^V\}$
Initialize target network $Q'$with weights $\boldsymbol{\theta}^{Q'} \leftarrow \boldsymbol{\theta}^Q$
Initialize replay buffer $R \leftarrow \emptyset$
for iteration$= 1, I$ do
    Sample random minibatch of $m$ transitions from $R$
    set $y_i = \begin{cases} r_i & \text{if } t_i < T \\ r_j + \gamma V'(\boldsymbol{s}'_i|\boldsymbol{\theta}^{Q'}) & \text{if } t_i = T \end{cases}$
    Update $\boldsymbol{\theta}^Q$ by minimizing the loss $L = \frac{1}{m}\sum_i \left(Q(\boldsymbol{s}_i, \boldsymbol{a}_i|\boldsymbol{\theta}^Q) - y_i\right)^2$
    Update the target network: $\boldsymbol{\theta}^{Q'} \leftarrow \tau\boldsymbol{\theta}^Q + (1-\tau)\boldsymbol{\theta}^{Q'}$
end for// collector thread $n$, $n) = 1, \ldots, N$
Randomly initialize policy network $\boldsymbol{\mu}(\boldsymbol{s}|\boldsymbol{\theta}_n^\mu)$
for episode$= 1, M$ do
    Sync policy network weight $\boldsymbol{\theta}_n^\mu \leftarrow \boldsymbol{\theta}^\mu$
     Initialize random process $\mathcal{N}$ for action exploration
    Receive initial observation state $\boldsymbol{s}_1 \sim p(\boldsymbol{s}_1)$
    for t=1,T do
        Select action $\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t|\boldsymbol{\theta}^\mu) + \mathcal{N}_t$
        Execute action $\boldsymbol{a}_t$ and observe reward $r_t$ and new state $\boldsymbol{s}_{t+1}$
        Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$ in $R$
    end for
end for
```

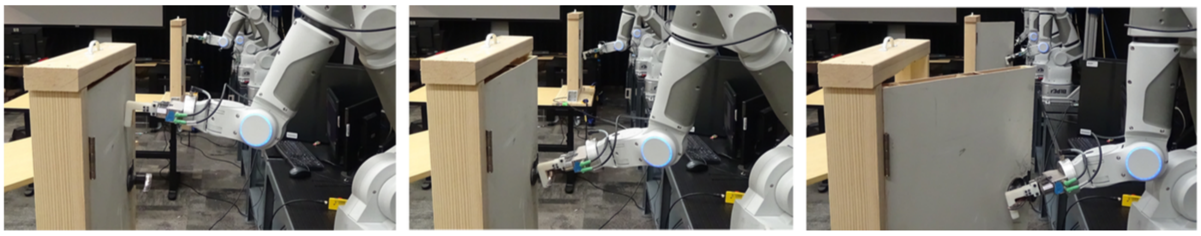
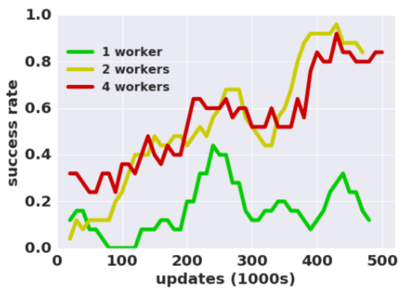


Fig. 5: The 7 DOF robots used for real world experiments on NAF with Asynchronous Off_Policy updates



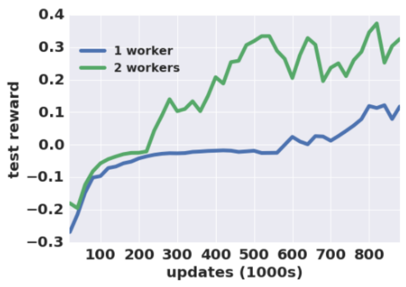


Fig. 6: Learning curves for real world experiments on NAF with Asynchronous Off_Policy updates applied to a reaching task and a door opening task.[7]

5 Conclusion

We presented the main algorithms that make it possible to apply Deep Q-learning to robotics Reinforcement Learning. First, we discussed the mathematical formulation and the challenges the problems that have to be taken into account to formulate practical algorithms. We, then, treated the case of the Atari games, which has been useful to formulate a basic Q-learning algorithm. After, we presented the main approaches that allow to apply Deep Q-learning to robotics tasks.

As discussed, continuous Deep Q-learning gives good results when applied to to basic tasks on the MuJoco simulator or in laboratory set-up Nevertheless, it will not work well on the more general and complex tasks involved by the real-world applications. A successful learning process in these scenarios can still be achieved only including some domain knowledge into the models (eg. reward shaping or adequate initialization of the Q-network).

Given that for complex tasks it is not possible to apply Deep Reinforcement Learning without including some pre-knowledge of the task, it seems necessary to find the most efficient way of including external information into the learning process. Imitation Learning could probably offer some techniques to guide the agent in the first episodes. For example by demonstrating some trajectories to the robot it would be possible to force it to evaluate non-optimal actions avoiding catastrophic behaviours.

References

[1] Jens Kober, Andrew J. Bagnell, and Jan Peters. Reinforcement Learning in Robotics: A Survey. *IJRR*, 2013.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *NIPS Deep Learning Workshop*, 2013.

[3] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-learning. *AAAI*, 2016.

[4] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *ICLR*, 2016.

[5] Timothy Lillicrap, Jonathan 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.

[6] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous Deep Q-Learning with Model-based Acceleration. *ICML*, 2016.

[7] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates. *ICRA*, 2017.