ALBSTADT-SIGMARINGEN UNIVERSITY

Can off-policy correction be ignored for multi-step DQN algorithms?

Tobias Ziegler

supervised by Prof. Dr. Knoblauch

2021-02-22

Abstract

DQN algorithms had great success in the Atari 2006 benchmark. Many improvements to the original DQN algorithm were proposed over the years, one being multi-step learning. Off-policy algorithms usually require some form of off-policy correction like importance sampling. Q-Learning and DQN methods are an exception to this rule. If multi-step learning is involved, off-policy correction again becomes necessary though. Algorithms, that use DQN including multi-step learning improvements often ignore this correction. It is unclear, why the proposed equations lack importance sampling. In this paper we will examine why off-policy correction is needed in the first place and why some algorithms like the original DQN proposal can ignore it. We will further empirically test, if multi-step learning really has positive effects on convergence, even if off-policy correction is being ignored.

Hochschule Albstadt-Sigmaringen

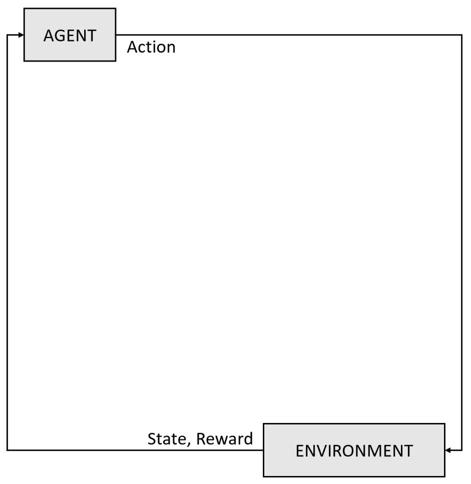
Albstadt-Sigmaringen University

Contents

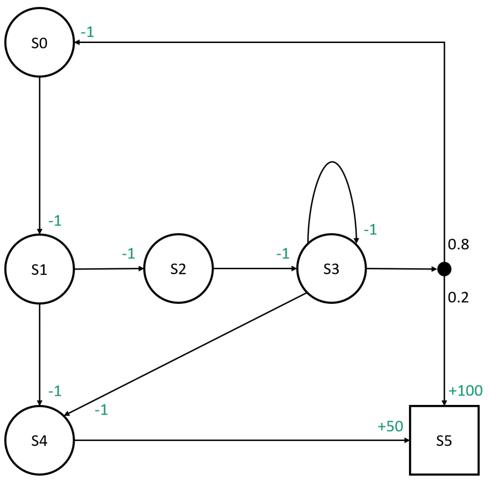
1 Introduction

Reinforcement learning has generated great success stories over the years. It was 1992, when Gerald Tesauro introduced a program, that would revolutionize reinforcement learning. The program proofed, that TD(λ), an algorithm by Sutton 1988 [1], could perform complex tasks. TD-Gammon was the first computer program, that was able to play backgammon on the same level as professional human players do. [2] Eleven years later another break-through impressed the reinforcement learning community. An agent was able to play different Atari 2006 games at or above human level. The algorithm has the same set of hyperparameters for each game. [3] The Q-Learning approach used deep neural networks to approximate state-action-values, hence it was called Deep Q-Learning (DQN). The original DQN algorithm [3] was tested on the Atari Benchmark, which is now standard in reinforcement learning: Agents compete in a range of Atari 2006 games, with nothing more than just raw pixels as input. Many improvements were made to this approach - Double DQN, Dueling Networks or multi-step learning are just three examples. These improvements were summarized in an amazing paper [4] in 2018, where new state-of-the-art results in the Atari Benchmark were proposed with an algorithm called rainbow. One specific improvement used in this paper is multi-step learning. It is an idea of temporal difference (TD) learning. Usually off-policy methods require some form of off-policy correction. Q-Learning methods and DQN specifically are an exception to this rule, but with the introduction of multi-step learning, off-policy correction again becomes necessary. The equations for multi-step learning in the rainbow paper [4] do not include this correction. It is not clear, why it is ignored and how this affects training. In this paper, we are going to examine why off-policy correction is needed, how multi-step learning improves convergence and how multi-step learning helps to escape the deadly triad. We provide empirical evidence, how multi-step learning behaves with respect to convergence - even when off-policy correction is being ignored.

First, the reader will be introduced to necessary theoretical concepts in the world of reinforcement learning, starting with the basic notations in a Markov decision process, followed by an overview about the taxonomy of reinforcement learning methods. Then, a technique called Monte Carlo learning will be introduced, where an agent can learn a task by sampling whole episodes. At this point it will become clear, what off-policy correction is and why it is needed. Next, temporal difference learning methods will be explained including Q-Learning. It will be clarified, why Q-Learning lacks importance sampling, although being an off-policy algorithm. Finally both methods, Monte Carlo and temporal difference learning, will be combined with n-step bootstrapping. Before we eventually propose empirical tests how multi-step learning behaves without off-policy correction, a closer look at the DQN algorithm and its multi-step learning variant is required.



(a)



(b)

Figure 1: Basic representations of (a) a reinforcement learning setup including an agent and the environment and (b) a simple example of a Markov decision process.

2 Background

A reinforcement learning system contains a learner, called the *agent*, and an *environment*. The agent can interact with the environment by taking different *actions* influencing none, partially, or fully observable *states* the agent perceives subsequently. The environment receives the desired action and consequently returns a new state and an associated feedback signal, the *reward*. This feedback signal is characterized by a scalar value and can either be positive, negative or zero (hence reinforcement). The agent's objective consists of finding a behaviour, the *policy*, maximizing its cumulative reward, the *return*. It differs from the supervised learning paradigm in the fact, that the agent never really gets supervision telling him directly what the best action would be. Instead, the agent has to figure it out on his own in a trial-and-error fashion. Figure 1a portrays the basic reinforcement learning setup visually.

This simple setting allows solving quite difficult tasks. Recent achievements in the field are getting great attention (see [5–7]). Although these accomplishments are pretty promising, reinforcement learning has some serious problems (see [8]), e.g. training efficiency. A general problem-solving algorithm has great potential though, despite those problems.

To formulate the reinforcement learning setup in a more formal way, let's consider an environment providing states at each discrete time step $t = 0, 1, 2, ...$ The produced states will be denoted with S_t and the actions chosen by the agent in S_t will be represented by A_t. The resulting reward R_{t+1} and state S_{t+1} follow consequently. [4] The state following an action a will often be denoted s' and the next chosen action a' similarly. Random variables are represented by capital letters and their values in lowercase, equivalent to the notation in Sutton and Barto's introduction to reinforcement learning [9]. Future rewards will be discounted by a factor γ weighting immediate rewards higher than the latter ones.

Reinforcement learning problems are formalized as finite (but mostly quite large) *Markov decision processes* (MDP) and will be introduced as tuples of $< \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma >$ shortly.

2.1 Markov decision processes

Modern reinforcement learning algorithms are based on control theory and Markov decision processes, which were approached by Richard Bellman in the 1950s [10] with a method called dynamic programming [11]. Some crucial equations, which will be introduced shortly, were discovered by Bellman at this time.

A Markov process models a stochastic process, where an environment with different states s can perform transitions to future states s' with certain probabilities, which do not change over time; thus the Markov process is considered stationary. A state space $\mathcal{S}$ contains all possible states s an environment can produce. Although $\mathcal{S}$ needs to be finite, it can be quite large. [12] This allows modelling complex problems, e.g. an agent learned to play StarCraft 2, a complex video game, at expert level. [6] The Markov property implies that a state s' only depends on previous states s and can be formally expressed in the following way:

$$Pr\{S_{t+1}|S_1, ..., S_t\} = Pr\{S_{t+1}|S_t\}.$$

Note, that it is always possible to add relevant information from the history to a current state s. [13, p. 6] A reward $r \in \mathcal{R}$ extends a simple Markov process to a Markov reward process by adding a scalar value to its state transitions. A factor γ can be used to balance weights of immediate and future rewards. A MDP additionally considers actions from an action space $a \in \mathcal{A}$. [12] The characteristics of a Markov decision process are summarized in table 1.

Table 1: Characteristics of a Markov decision process.

Symbol	Names	Description
$\mathcal{S}$	State space	Every state possible in the environment
$\mathcal{A}$	Action space	Every action the agent can take, often denoted like $\mathcal{A}(s)$ indicating that only the actions possible in state s are considered
$\mathcal{P}$	Transition probabilities	$P(s', r|s, a)$ denotes the probability of s' and the associated reward r given s and a
$\mathcal{R}$	Reward	$r(s, a, s')$ denotes the immediate reward the agent gets when taking action a in state s and ending up in state s'
γ	Discount factor	$\gamma \in [0, 1]$ to penalize later rewards when calculating cumulative rewards of a state s

Figure 1b shows an example illustrating the characteristics summarized by table 1. The Markov decision process defines an environment, where the agent can consider different actions represented by outgoing edges. Non-terminal states are represented by circles, squares serve as terminal states. Transitions are characterized by edges and their associated rewards are portrayed as green scalar values. Most of the transitions are deterministic, except the one described by an black dot, where the outcome probabilities are explicitly stated. Each transition is rewarded a -1 signal, excluding the terminal state S_5, where every incoming edge has a positive reward.

A sequence of states, actions and rewards is called an episode and ends with a terminal state S_T. A trajectory τ can be any subset of an episode. [13, p. 3]

$$\tau = s_0, a_0, r_1, s_1, a_1, ..., s_T.$$

A probability distribution p defines the likelihood of a transition to state s' with an associated reward r given state s and action a:

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\}, \tag{1}$$

for all $s', s \in \mathcal{S}, r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$ satisfying the Markov property. As p is a probability distribution for each s and a, the following equation has to be true:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}(s). \tag{2}$$

Hence p characterizes the dynamics of the MDP completely and thus can be used to compute anything relevant to the environment, like the state-transition-probabilities (equation 3), the expected reward for state-action-pairs (equation 4) or the expected reward for state-action-nextstate-triples (equation 5). [9, pp. 48–49]

$$p(s'|s, a) = Pr\{S_{t+1} = s'|S_t = s, A_t = a\} = \sum_{r \in R} p(s', r|s, a), \tag{3}$$

$$r(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a), \tag{4}$$

$$r(s, a, s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] = \sum_{r \in R} r \frac{p(s', r|s, a)}{p(s'|s, a)}. \tag{5}$$

The agent's objective is to find a behaviour π to maximize its obtained reward. Long-term rewards get penalized by adding a discount factor $\gamma \in [0, 1]$. This has mathematical reasons for convergence, but also makes sense intuition-wise: Getting an immediate reward is better than future ones. The return G_t as the cumulative future reward of an episode starting at time step t is defined in equation 6. [12]

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{6}$$

In most cases algorithms estimate some value-functions evaluating *goodness* of a state s. This obviously strongly depends on the actions the agent takes. This behaviour is defined by a policy π mapping probabilities to actions a conditioned by different states $\pi(a|s)$. There are two important types of value functions: the *state-value function* and the *action-value function*. The first one represents the value of a state s. It is defined as the expected return of a given state, following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{7}$$

The action-value function on the other hand conditions the value not only by the state, but additionally by the action a. This is also called the q-value and describes the expected return, when taking an action a in state s and then continue following the policy π. [9, p. 58]

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{8}$$

2.2 The Bellman Equation

Equations 7 and 8 can be rewritten in an iterative fashion to obtain the bellman equations for the on-policy value functions 9 and 10. [14]

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{9}$$

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a)] \tag{10}$$

A behaviour π can be considered better than another policy π' if

$$\pi \geq \pi' \text{ iff } v_\pi(s) \geq v_{\pi'}(s), \quad \forall s \in \mathcal{S}.$$

That means that there always exists at least one policy that is equal or better than all the other policies. [9, p. 62] If the agent follows one of these optimal behaviours π_*, the optimal state-value function v_* and the optimal action-value function q_* can be defined as follows:

$$v_*(s) = \max_\pi v_\pi(s), \quad \forall s \in \mathcal{S}, \tag{11}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a), \quad \forall s \in \mathcal{S}. \tag{12}$$

These formulas can then be transformed to

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a], \tag{13}$$

and

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')|S_t = s, A_t = a], \tag{14}$$

representing the bellman equations for the optimal value functions. These equations can also be stated with the p function we declared earlier, but we are interested in model-free approaches as this p-function is *normally* unknown. Dynamic Programming e.g. can solve those equations when there is a known model. The bellman equations are used quite heavily in all kinds of reinforcement learning methods. [14]

Dynamic programming could use the p-function to iteratively *evaluate* the value functions of a policy π and subsequently update (*improve*) the policy into the right direction by acting greedily. This algorithm is called *Generalized Policy Iteration* (GPI). Equation 15 denotes the policy evaluation step, 16 the policy improvement step and 17 illustrates the policy iteration with $\xrightarrow{E}$ as evaluation step and $\xrightarrow{I}$ as policy improvement. This iteration converges to optimality. [12]

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{15}$$

$$q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a] = \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{16}$$

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_* \tag{17}$$

But most of the time these dynamics in form of the p distribution are unknown. That's when algorithms from the Monte Carlo or temporal difference learning family can be used. They will be introduced shortly, but before we will take a closer look at these specific approaches, we will investigate in the general taxonomy of reinforcement learning algorithms.

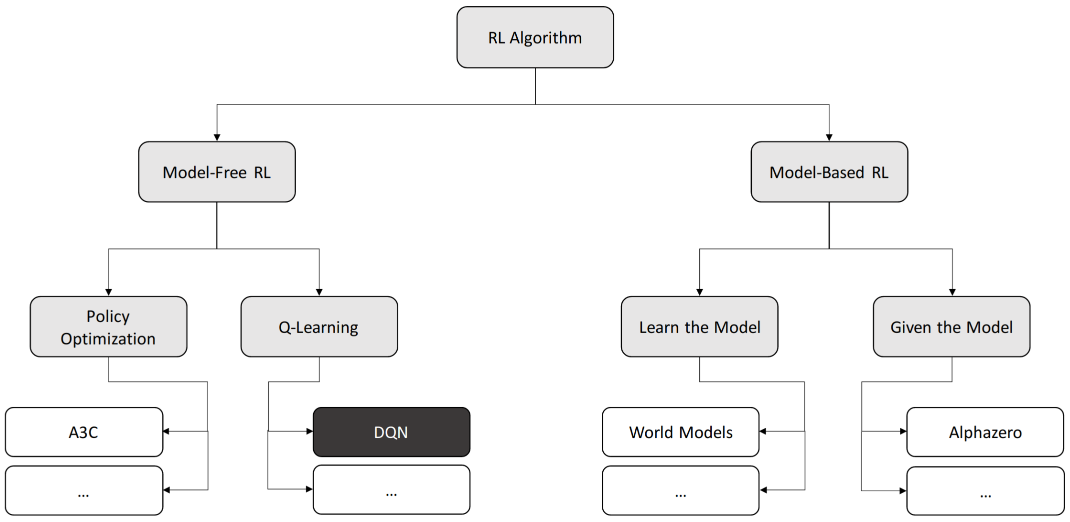


Figure 2: A non-exhaustive taxonomy of reinforcement learning.

2.3 Taxonomy of Reinforcement Learning Methods

Figure 2 shows an attempt from OpenAI [14] to draw a non-exhaustive taxonomy of the reinforcement learning world. Without being a full overview, it helps to understand the intuition behind these approaches and how they can be separated from one another.

First of all, algorithms can be distinguished by their knowledge of the dynamics of an environment. We already saw how a model of the environment can be used to approximate an optimal policy π_* by using a technique called dynamic programming. Methods, where the transition probabilities are known or an agent learns a model of the environment, are called *model-based* methods. *Model-free* approaches on the other hand, map observations directly to actions or related value functions. This means, that these techniques do not use transition probabilities, but rather purely rely on examples to learn. It can be quite hard to approximate the dynamics of an complex environment. That's why research often focuses on model-free approaches, yet more and more important and groundbreaking achievements were accomplished with model-based methods; MuZero [15] is one of these great success stories that easily comes to mind. The "fruitful" combination of different improvements to the original DQN algorithm [4] is a famous example of a great accomplishment for model-free algorithms. [12, 14]

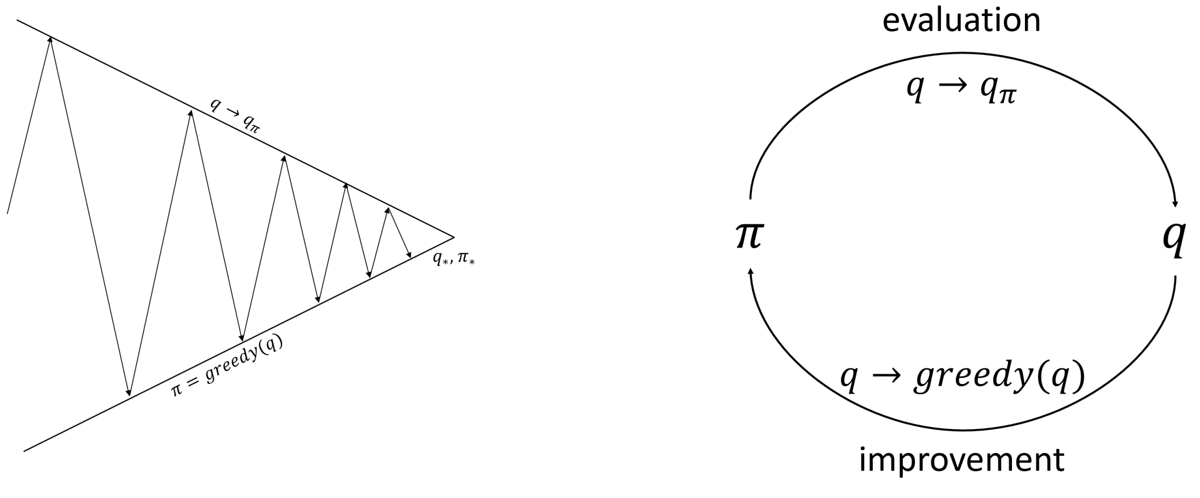
Model-free algorithms can further be subdivided into Policy Optimization and Q-Learning methods. Approaches like REINFORCE [16], that fall into the Policy Optimization category explicitly use a policy $p_\theta(a|s)$ to optimize the parameters θ with respect to an objective function $J(\pi_\theta)$, e.g. the expected reward $\mathbb{E}[R_t]$ and consequently approximate an optimal policy. [14, 17] A3C (Asynchronous Advantage Actor-Critic) uses an asynchronous actor-critic architecture and achieved state-of-the-art results in 2016 in the Atari benchmark. Other methods like Proximal Policy Optimization (PPO) [18] use surrogate objective functions to estimate $J(\pi_\theta)$. Q-Learning algorithms on the other hand try to estimate action-value functions to find an optimal behaviour. There are several ways to evaluate state-action pairs. They differ in the way how they use samples: While Monte Carlo methods use whole episodes to learn, temporal difference learning starts learning before an whole episode is fully sampled. They resemble one another along with dynamic programming by both using some form of GPI algorithm to approximate an optimal policy. Algorithms like [3, 4, 19, 20] are great examples for Q-Learning methods. Because Policy Optimization and Q-Learning both have advantages and disadvantages, some methods try to combine the best of both worlds (e.g. DDPG or SAC [21, 22]). In the succeeding part of this work, Q-Learning will be the main focused method. DQN is a form of Q-Learning, where the value functions are getting approximated by a neural network. [13, pp. 11–15] [14]

Table 2: Methods, that will be discussed in the following.

Method	**Description**
Dynamic Programming	Uses an p function to determine value functions.
Monte Carlo	Uses samples of whole episodes to determine value-functions.
TD(0)	A special form of TD-Learning, where learning starts immediately after taking an action. SARSA, Q-Learning and DQN are three TD(0) algorithms, we will discuss.
n-step bootstrapping	TD-Learning, where learning starts immediately after n steps. We will take a closer look at n-step DQN algorithms.

Algorithms also differ in their data they are using to learn. Some methods use historical data from previous versions of the agent or even from human demonstrations - these are called off-policy. An agent can also obtain his own data from the environment and learn directly from this behaviour. Algorithms of this kind are called on-policy learning methods. Examples of both approaches will be covered in this paper. [12, 14] [13, p. 16] Q-Learning methods are considered off-policy.

Now let's have a look how Monte Carlo methods use samples to estimate value-functions and consequently approximate an optimal policy.



(a) **(b)**

Figure 3: The GPI algorithm repeatedly evaluates action-values from a policy and improves this policy greedily, subsequently. Both (a) and (b) visualize this algorithm equally.

2.4 Monte Carlo methods

Monte Carlo methods learn without prior knowledge about the environment's dynamics. The model-free, value-based approaches try to estimate value functions by averaging observed returns of each state. These approximations can be used to estimate an optimal policy π_*. They only work on episodic tasks, meaning that episodes need to terminate. *Experiences* are used to average returns of samples and are generated by interaction with or simulation of an environment. These experiences are trajectories of states, actions and rewards and MC methods use trajectories of a whole episode as experience. Although the algorithm learns in an incremental way, it cannot be called true online learning, because MC must sample a whole episode to learn, rather than learning step-by-step like temporal difference learning does, which will be introduced shortly. Figure 4 shows the differences between a dynamic programming approach, where all the dynamics of the environment are known, a MC method, where the whole episode is being sampled and TD(0)-Learning, where only one step has to be sampled to start learning. [9, pp. 86–87] [23]

Monte Carlo approaches try to improve a policy iteratively to approximate an optimal policy π_*. They use an idea of dynamic programming called generalized policy iteration (GPI), where a value function q_π is repeatedly approximated to improve the policy w.r.t. the current q_π value. Figure 3 clarifies the interaction of *prediction* and *control*, where the algorithm iteratively estimates the state-action-values of the current policy and then uses these to improve the current policy $\pi \to \pi'$. [9, pp. 91–92] The new, optimized policy π' can then be used to again predict the state-action-values until approaching π_* leading to an optimal policy, which can be proven as follows: [12]

Consider a policy π and an improved policy π' by greedily taking actions w.r.t. the current state-action-values:

$$\pi'(s) = arg \max_{a \in A} Q_\pi(s, a).$$

π' is guaranteed to be better, because:

$$\begin{aligned} Q_\pi(s, \pi'(s)) &= Q_\pi(s, arg \max_{a \in A} Q_\pi(s, a)) \\ &= \max_{a \in A} Q_\pi(s, a) \geq Q_\pi(s, \pi(s)) = V_\pi(s). \end{aligned}$$

Monte Carlo Prediction There are two different ways to compute the returns for estimating v_π or q_π, each one dealing differently with the fact that a state (or state-action pair) can be visited more than once in an episode. The *first-visit MC method* estimates the value functions by calculating the average return of a state s since the first time the state has been visited. The *every-visit MC method* on the other hand averages over the returns of each visit of the state s. [9, pp. 92–93]

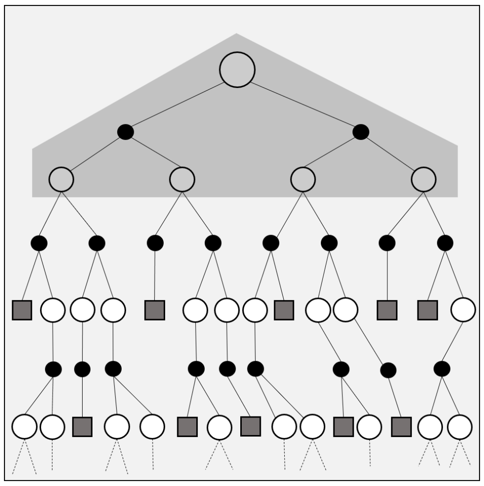
Action-values are more useful than state-values if the dynamics of an environment are unknown. Unfortunately, this leads to an issue called the maintaining exploration problem. Many state-action-pairs may never been visited, especially in deterministic policies, hence the returns of alternative actions cannot be computed. To solve this problem one can use exploration starts: Episodes start in certain state-action-pairs with a probability greater than zero. This is only possible when the environment allows a simulation-like interaction. Another easy solution would be to just allow stochastic nonzero-probability policies. More practical approaches to ensure that the agent is taking every path an infinite amount of times are ϵ-soft policies, for example the ϵ-greedy policy. They slowly approach a deterministic policy starting from $\pi(a|s) > 0$. [9, pp. 96–97, 100–103]

Monte Carlo Control After estimating the values of state-action pairs, the policy can be improved greedily with respect to the current action-value. This means that the action with the highest action-value can be chosen:

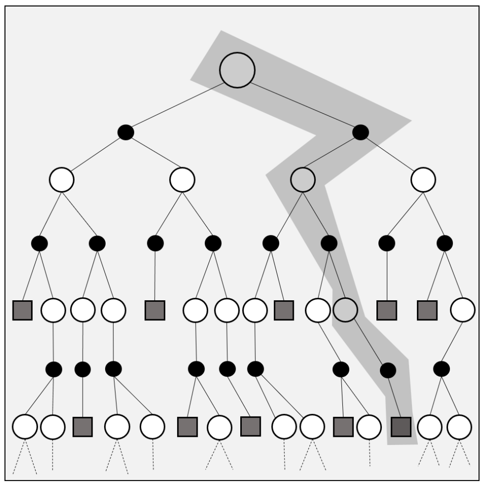
$$\pi(s) = arg \max_{a} q(s, a).$$

Algorithms can either be implemented on- or off-policy. While on-policy methods evaluate and improve the policy which is making the decisions, off-policy implementations in fact encapsulate the target policy (the policy being learned about) from the behaviour policy (which generates data by its decisions). Algorithm 1 from Sutton and Barto [9, p. 101] shows an on-policy first-visit MC implementation with ϵ-soft policies, estimating $\pi \approx \pi_*$. [9, pp. 97, 100–101]

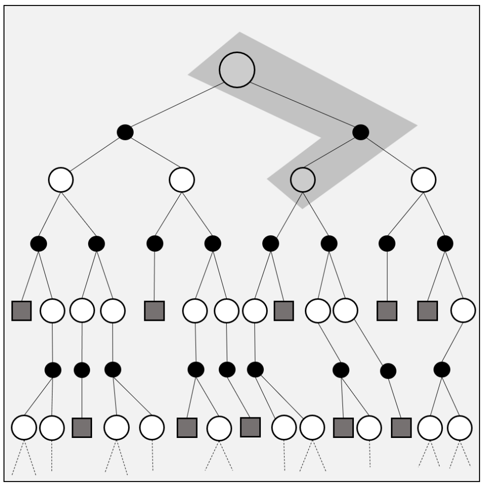
This on-policy implementation only approaches a policy, that is close to optimal, because it still contains exploration. Off-policy implementations on the other hand really approach optimal policies. Although being more powerful in general, off-policy implementations suffer from greater variance and slower convergence. Off-policy methods separate the behaviour policy from the target policy: The behaviour policy is used to generate data by taking decisions and can be of an exploratory nature; the target policy is the policy being learned about. In fact on-policy methods are a special case of off-policy learning, where the target policy is the behaviour policy. [9, p. 103]



(a) Backup diagram of a DP approach.



(b) Backup diagram of a MC approach.



(c) Backup diagram of a TD(0) approach.

Figure 4: Comparison between different backup diagrams from [23].

Algorithm 1 On-policy first-visit MC control (for ϵ-soft policies), estimates $\pi \approx \pi_*$

```
Algorithm parameter: small $\epsilon > 0$
Initialize:
    $\pi \leftarrow$ an arbitrary $\epsilon$-soft policy
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Repeat forever (for each episode):
    Generate an episode following $\pi : S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, ..., 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1, ..., S_{t-1}, A_{t-1}$ :
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow$ average($Returns(S_t, A_t)$)
            $A^* \leftarrow arg\max_a Q(S_t, a)$    (with ties broken arbitrarily)
            For all $a \in \mathcal{A}(S_t)$ :
                $\pi(a|S_t) \leftarrow \begin{cases} 1-\epsilon+\epsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
```

Importance sampling [9, pp. 103–110] The off-policy case raises a problem, that needs to be solved: Data generated by the behaviour policy is not sufficient to estimate values for the target policy, because they have different probability distributions. *Importance sampling* can be used to estimate v_π or q_π from experience of another policy b. Importance sampling presuppose the *assumption of coverage*: If a probability of taking an action a under policy π is greater than zero $\pi(a|s) > 0$, the probability of taking the same action a under policy b has also to be greater than zero $b(a|s) > 0$, meaning that every action taken under π is also to be taken (at least sometimes) under b.

$$\pi(a|s) > 0 \Rightarrow b(a|s) > 0.$$

Note that this doesn't mean, that π cannot be deterministic. Actually π is deterministic in many cases, e.g. when they act greedy w.r.t. the current estimate of the action value, while b remains stochastic for continuous exploration. The probability of a trajectory $A_t, S_{t+1}, A_{t+1}, ..., S_T$ occurring under a policy π, can be stated as follows:

$$\begin{aligned} &Pr\{A_t, S_{t+1}, A_{t+1}, ..., S_T | S_t, A_{t:T-1} \sim \pi\} \\ &= \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1})...p(S_T|S_{T-1}, A_{T-1}) \\ &= \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k), \end{aligned} \tag{18}$$

with p being the state-transition probability. The *importance-sampling-ratio* as the relative probability of the trajectory occurring under the target and behaviour policies, can be stated like this:

$$p_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}, \tag{19}$$

with the normally unknown probability functions of the MDP cancelling out. With this ratio and the returns G_t from a behaviour policy, we can estimate returns of the target policy. The importance-sampling-ratio can be used to compute $v_\pi(s)$ with the *wrong* expectation $\mathbb{E}[G_t|S_t = s] = v_b(s)$:

$$\mathbb{E}[p_{t:T-1}G_t|S_t = s] = v_\pi(s).$$

Ordinary importance sampling can be formulated in a way represented by equation 20, following this notion: Time steps t increment across episode boundaries, i.e. an episode 1 starting at step $t = 0$ and ending at $t = 100$ will be followed by an episode 2 starting at $t = 101$. $\mathcal{T}(s)$ denotes a set of all time steps where s is visited. $T(t)$ denotes the first time an episode is terminated following step t, hence G_t represents the return from t up to $T(t)$. The returns of a state s are contained by $\{G_t\}_{t\in\mathcal{T}(s)}$ and similarly $\{p_{t:T(t)-1}\}$ contains the corresponding importance-sampling ratios. [9]

$$V(s) = \frac{\sum_{t\in\mathcal{T}(s)} p_{t:T(t)-1}G_t}{|\mathcal{T}(s)|}. \tag{20}$$

Weighted importance sampling on the other hand uses an weighted (instead of the earlier used simple) average:

$$V(s) = \frac{\sum_{t\in\mathcal{T}(s)} p_{t:T(t)-1}G_t}{\sum_{t\in\mathcal{T}(s)} p_{t:T(t)-1}}. \tag{21}$$

Algorithm 2 from [9, p. 111s] shows an implementation of an off-policy every-visit MC algorithm estimating π_* using weighted importance sampling. It updates the state-action-values incrementally, which can be formulated as follows:

Consider a sequence of returns $G_1, ..., G_{n-1}$ belonging to a state s and associated random weights W_i. Equation 21 can then be used to calculate V_n of s:

$$V_n = \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \quad n \geq 2.$$

The value of the state can then be updated after receiving a new return G_n:

$$V_{n+1} = V_n + \frac{W_n}{C_n}[G_n - V_n], \quad n \geq 1.$$

A cumulative sum C_n of the given weights of the first n returns must be maintained. C_n must be updated subsequently:

$$C_{n+1} = C_n + W_{n+1}.$$

Algorithm 2 Off-policy MC control, for estimating $\pi \approx \pi_*$

```
Initialize, for all $s \in S, a \in A(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$
    $\pi(s) \leftarrow arg\max_a Q(s,a)$    (with ties broken consistently)

Loop forever (for each episode):
    $b \leftarrow$ any soft policy
    Generate an episode using $b : S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, ..., 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
        $\pi(S_t) \leftarrow arg\max_a Q(S_t, a)$    (with ties broken consistently)
        If $A_t \neq \pi(S_t)$ then exit For loop
        $W \leftarrow W\frac{1}{b(A_t|S_t)}$
```

2.5 Temporal difference learning

"If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal difference (TD) learning." [9, p. 119]

Temporal difference learning goes back to 1959, where Samuel [24] introduced a program, that learned to play checkers. More applications of this technique followed [25–27], but it was Sutton [1], who would lay the theoretical foundations for later work and explained why these methods worked in the first place. He proved that these algorithms converge to optimality.

TD Learning is another model-free approach that learns from experience. It combines ideas from Monte Carlo and dynamic programming. Unlike MC methods, it calculates returns from incomplete episodes and consequently allows true online learning. TD(0) is a special case of TD learning, where the algorithm adjusts it's state values immediately after taking one step instead of waiting for the episode to end. Consider a simple MC every-visit method like

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)], \tag{22}$$

with α representing the learning rate. Before the value of S_t can be updated, the return G_t of the whole episode needs to be estimated. TD(0) on the contrary adjusts $V(S_t)$ immediately after one step:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)], \tag{23}$$

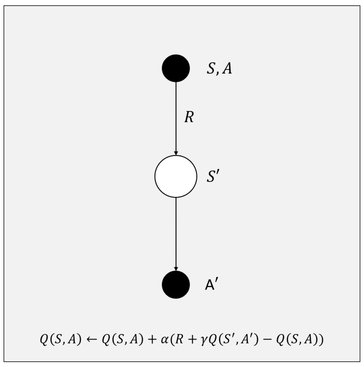
with the TD-Target $G_{t:t+1}$ and TD-Error δ, defined as

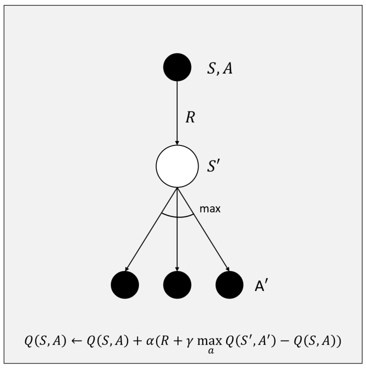
$$G_{t:t+1} = R_{t+1} + \gamma V(S_{t+1}),$$

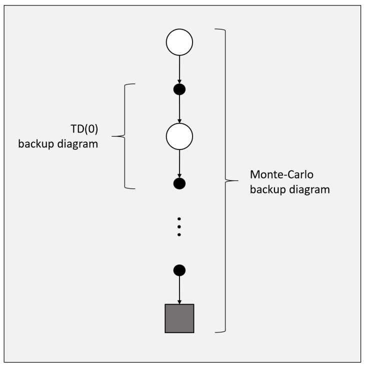
and

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t).$$

Because the update is based on an estimate, TD-learning is considered a *bootstrapping* method. Both TD and MC converge to optimal solutions, yet it is not mathematically proven, which one converges faster, although TD is claimed to converge more rapidly. Bootstrapping furthermore does not require a model like dynamic programming would do. One advantage TD-Learning has over Monte Carlo methods is, that it naturally is implemented in an incremental, online way and thus does not require the episode to end. This can be specially convenient for environments with very long episodes or episodes, that do not end at all. [9, pp. 119-138]







(a) **(b)** **(c)**

Figure 5: Comparison between the backup diagram of (a) the SARSA algorithm and (b) the Q-Learning approach; (c) once again demonstrates the difference of TD(0) and Monte Carlo methods.

2.5.1 SARSA

Let's first introduce a TD-Learning algorithm for on-policy control called SARSA. It is an TD(0) approach, hence learning starts immediately after taking one step. Action-values are again favoured over simple state-values, like already discussed in the previous section about Monte Carlo. Hence equation 23 can be extended by actions, resulting in

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \tag{24}$$

An optimal policy can be attained by an iterative step-by-step update of $Q(S_t, A_t)$. If the next state should be terminal, the q-value is defined as zero. The name SARSA comes from the involved experience represented as $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. Figure 5 shows the backup diagram for the SARSA algorithm compared to Q-Learning, which is the off-policy variant of SARSA. [9, pp. 129–131]

2.5.2 Q-Learning

Q-Learning is an off-policy control algorithm introduced by Watkins [28]. It takes actions according to a behaviour policy that often is an ϵ-greedy one. The target policy on the other hand is a greedy policy w.r.t. the bellman optimality equation, thus chooses the action with the highest action-value independently. The behaviour policy ensures that all state-action pairs are getting visited and thus the algorithm directly estimates q_*.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]. \tag{25}$$

Algorithm 3 from Sutton and Barto [9, p. 130] exemplifies an implementation of the SARSA method; algorithm 4 from Sutton and Barto [9, p. 131] the implementation of the Q-Learning approach, respectively. Both seem to be quite similar, but note that SARSA takes the action w.r.t. the ϵ-greedy policy (meaning, that the agent takes the action with the maximum q-value with $p = 1 - \epsilon$ and a random action otherwise. Q-Learning on the other hand always takes its action greedily w.r.t. the q-value. [9, pp. 131–134]

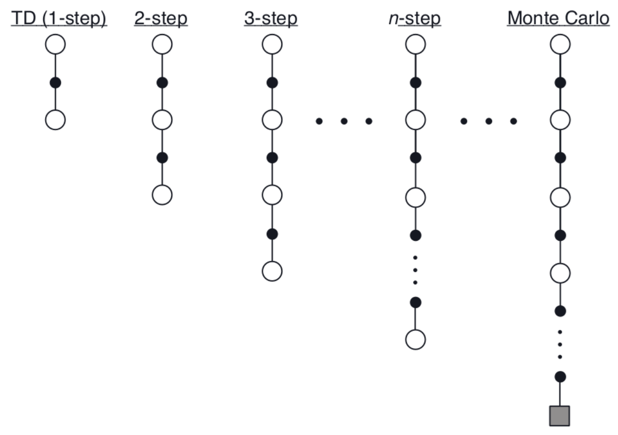


Figure 6: The different possibilities temporal difference learning offers between TD(0) on one, and MC on the other side. [9, p. 142]

Algorithm 3 SARSA algorithm estimating q_*:

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R, S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'; A \leftarrow A'$;
until $S$ is terminal
```

Algorithm 4 Q-Learning algorithm estimating q_*:

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A$, observe $R, S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
until $S$ is terminal
```

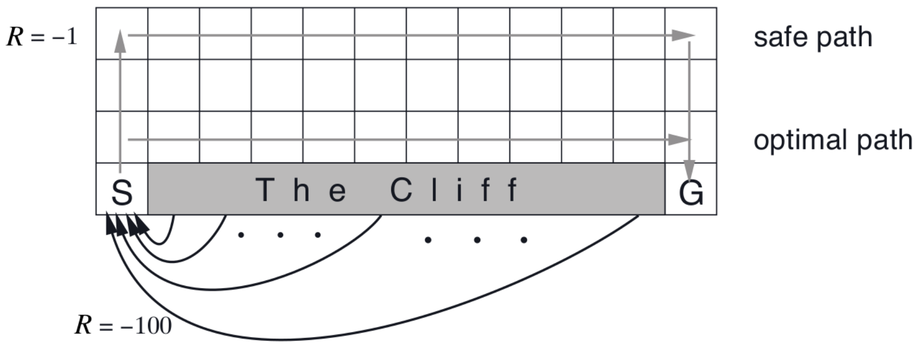


Figure 7: Comparison between the learned policies of the SARSA algorithm and Q-Learning. [9, p. 132]

The only difference between Q-Learning and SARSA is, that SARSA chooses A' by following its behaviour policy (ϵ-greedy) while Q-Learning chooses A' by taking the action with the maximal q-value, following it's target policy (greedy w.r.t. the q-value), which seems like a small difference, but indeed is an important distinction. Figure 7 compares the learned policies of both approaches:

Imagine a grid world, where each step rewards a -1 signal, while falling of the grid is punished with a -100 *reward* and lets the agent start over at the initial state S. The episode ends, when the agent reaches the state G. The behaviour policy is an ϵ- greedy one with $\epsilon = 0.1$. The agent's objective is to maximize its obtained reward. While Q-Learning learns the optimal path and ignores the possibility of falling of the edge (because its only part of the behaviour policy), SARSA learns the safer path. This would lead to an higher *online* performance of SARSA compared to Q-Learning. If ϵ would be reduced gradually, both approaches would converge to the optimal path. [9, p. 132]

2.5.3 Why Q-Learning lacks importance sampling

Equation 25 defines the Q-Learning update of state-action-values, which is the off-policy variant of the SARSA algorithm. Section 2.4 claimed, that off-policy methods need importance-sampling due to the differences between target- and behaviour policy. This raises the question, why equation 25 lacks an importance-sampling-ratio? Well, although being off-policy, Q-Learning represents an exception. The update is conditioned by a state S and an action A taken under the behaviour policy. The corresponding reward R_{t+1} is independent of the policy (because the action has already been taken) and the subsequent greedy-action A' is exactly what the target-policy would have done too, thus the probability distributions of both policies are the same, after the action is taken. [9, p. 149] Due to the assumption of coverage all the actions the target policy would take, will also be taken by the behaviour policy. Therefore importance sampling is not needed.

Note, that this only holds true, if the TD-Target is an *one-step return*, i.e. it is a TD(0) implementation. If the TD-Target includes an whole trajectory, the probability distributions of the behaviour- and target policy would again diverge, hence importance sampling would be necessary. N-step bootstrapping does exactly this and will be introduced now.

2.6 Combining MC and TD: n-step bootstrapping

TD(0) was a special case of TD-Learning, where the agent starts learning after taking one step. MC on the other hand was special too, where the agent can only learn after sampling an whole episode. N-step bootstrapping is filling the gap between these two approaches and provides an whole spectrum of possible solutions, where the best approach usually lies in between these two extremes. [9, p. 145] Figure 6 visualizes the connection between TD(0) with $n = 1$ and MC on the other end with $n \to \infty$. Recall, that the return of an episode, starting at timestep t and ending at the terminal timestep T is used as *target* by MC and can be formalized like this:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-t-1} R_T.$$

TD(0) on the other hand uses bootstrapped one-step updates and thus the TD-target is defined as

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1}),$$

and can be called *one-step return*. V_t denotes the estimate of v_π at time t. This can be generalized into the following form:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), \tag{26}$$

describing the *n-step return* using rewards from timestep t up to $t + n$ and the current estimate of $v(S_{t+n})$. This return can be used as TD-Target to update the value-function, similar to how TD(0) uses the one-step return:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)], \quad 0 \leq t < T, \tag{27}$$

where the update obviously can only be executed after $t + n$ timesteps. [9, p. 143]

N-step SARSA is an algorithm, which implements n-step TD for control and generalizes the already introduced *one step SARSA* method. Equation 26 can be expanded with actions, obtaining the n-step returns conditioned by states and actions:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), n \geq 1, 0 \leq t < T - n, \tag{28}$$

which can be used for the update:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \quad 0 \leq t < T. \tag{29}$$

This on-policy update can be transformed to off-policy once again by introducing importance sampling into this equation:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha p_{t+1:t+n}[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \quad 0 \leq t < T, \tag{30}$$

where the importance-sampling-ratio starts and ends one step later. Section 2.5.3 already clarified, why this is necessary. [9, pp. 146–149]

We have seen how MC and TD algorithms work and how they relate to each other. Now we have the theoretical knowledge to understand how DQN algorithms work and how multi-step returns can be used to extend the original DQN algorithm to obtain more stable algorithms.

3 Deep Q-Learning

3.1 Original DQN

Mnih, et al. [3] proposed a deep learning approach to Q-Learning called DQN in 2013. They used a neural network as value-function approximation for estimating action-values given raw pixels of seven Atari 2006 games as input. They outperformed previous approaches in six games and humans at expert level in one. Two years later Mnih, et al. [19] achieved another breakthrough. They were able to achieve human-level control in 49 Atari games. Each game was solved with the same algorithm, same hyperparameters and same architecture.

The images provided by the emulator were preprocessed to reduce computational cost: They were rescaled, greyscaled and cropped. One state is always represented by four of these processed images to allow the algorithm to understand speed and directions. A function called ϕ applied these steps to the traditional images.

Given these states s a neural network approximates the action-values for each possible action a. The network is trained by batches of transitions (s_t, a_t, r_t, s_{t+1}), obtained by the experiences from the agent. By truncating the trajectories into these smaller transitions, correlations between consequent experiences can be broken, enabling a more efficient training process. The network learns by minimising an error given by a loss function $L_i(\theta_i)$, which was formulated by Mnih, et al. [19] as follows:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \overline{\theta_i}) - Q(s, a; \theta_i))^2],$$

where θ_i are the parameters of the policy net and $\overline{\theta_i}$ are parameters of the target net used to train the network at iteration i. The target net is an exact copy of the policy net, which gets copied every C steps. The experiences (s, a, r, s') are drawn uniformly; a method called prioritized experience replay actually tries to improve this, by drawing more important samples more likely and was introduced by Schaul, et al. [29]. Note that they too speak of importance sampling, but in another context. The term importance sampling in this paper means off-policy correction unless explicitly stated otherwise. Several other improvements were made to this approach like Double DQN [30], Dueling Networks [31], Multi-step-bootstrap targets [1] or distributional Q-Learning [20]. These methods were summarized by the rainbow algorithm [4], which achieved new state-of-the-art results by combining them in a "fruitful"-manner. We will take a closer look at using Multi-step bootstrap targets as improvement to the original DQN algorithm.

Algorithm 5 from Mnih, et al. [19] describes the training process of this approach in more detail. Note that DQN is an off-policy method, which does not require importance sampling (see Q-Learning) as it is basically a Q-Learning algorithm expanded with a neural network as function approximation. The behaviour policy is an ϵ-greedy one with an decaying ϵ.

Algorithm 5 Deep Q-Learning with Experience Replay

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\overline{\theta} = \theta$
For episode = 1, $M$ do:
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For t = 1, T do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = arg\max_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \overline{\theta}) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network
        parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    end for
end for
```

3.2 N-step DQN

DQN is an off-policy method, uses function approximation to calculate q-values and is considered a bootstrapping method. Sutton and Barto [9, pp. 264–265] describe this combination of function approximation, bootstrapping and off-policy training as *the deadly triad*. The deadly triad should lead to instability and divergence, which can be avoided if only two of these three criteria apply. Mnih, et al. [19] already showed that algorithms like DQN can be successful, although falling into this category. That's why Hasselt, et al. [32] practically examined the deadly triad in context of DQN. They state, that these criteria are in no means of binary nature. Each one of these three elements can be tweaked. In this paper we are focusing on the bootstrapping criterion.

Multi-step returns can be used to reduce the impact of bootstrapping. This has been already shown beneficial by implementations like rainbow [4] or Distributed Prioritized Experience Replay [33]. Hasselt, et al. [32] hypothesized, that *longer multi-step returns will diverge less easily* and claim, that their experiments provided strong support for this hypothesis. They found that a higher n clearly reduces instability.

Hessel, et al. [4] formulated their approach to n-step DQN as follows:

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1},$$

$$(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\overline{\theta}}(S_{t+n}, a') - q_\theta(S_t, A_t))^2,$$

which basically reflects equation 28 with A_{t+n} chosen greedily and equation 29 as the update for the on-policy case, thus lacking importance sampling to synchronize the behaviour and target policy. Because the behaviour policy of the DQN algorithm is an ϵ-greedy one, but the target policy is greedy w.r.t. the current q-values and because the Q-Learning exception in section 2.5.3 does not hold anymore, the equation from rainbow [4] should include off-policy correction (e.g. importance sampling), resulting in the following equation to update the weights of the neural network:

$$\theta_{t+n} = \theta_{t+n-1} + \alpha p_{t+1:t+n}(G_{t:t+n} - q(S_t, A_t; \theta_{t+n-1}))\nabla q(S_t, A_t; \theta_{t+n-1}),$$

where $G_{t:t+n}$ chooses a' greedy w.r.t. the current q values of the target net.

Consequently one can argue that the equation used by Hessel, et al. [4] is not completely right. Although being theoretically inaccurate, it should improve training nevertheless like their experiments also proved: They tested $n \in \{1, 3, 5\}$ and chose $n = 3$. That could be, because they are escaping the deadly triad by choosing a higher n. Using an ϵ greedy approach and by decaying ϵ to a small number, allows the two policies to approach each other while training progresses, reducing the impact importance sampling would have.

This leads to the hypothesis, that it helps to bootstrap with an higher n, even if off-policy correction is ignored. With a higher n on the other hand, the probability distributions of the target and behaviour policy should diverge, leading to more instabilities in training. This effect could be countered with the strongly decaying ϵ later in training though. This hypothesis will be tested empirically in the following section.

4 Experiments

4.1 OpenAI Gym

Gym is an open source library providing several reinforcement learning tasks, algorithms can be evaluated on. These tasks can vary from several walking simulators, robotic tasks, simple control theory problems from classic RL literature to even Atari games, which are simulated through the Arcade Learning Environment (ALE) introduced by Bellemare, et al. [34]. Researchers often compare their algorithms by matching them against classic Atari games like Pong, Breakout or Pinball. (see [3, 4, 19, 29, 31]) An algorithm is normally benchmarked against more than just one game.

Gym offers an easy to use interface to create and manage environments. An instance of the pong environment for example, can be easily initialized by the following command:

```
import gym
env = gym.make("PongNoFrameskip-v4")
```

Afterwards an action can be executed and the subsequent state, the corresponding reward, a flag stating that the subsequent state is terminal or not and additional information is returned:

```
next_state, reward, done, _ = env.step(action)
```

The environment only accepts actions from the action space and returns states from the state space, respectively. Information about these spaces can be accessed easily:

```
env.action_space
env.observation_space
```

More information about the gym library can be obtained from the official documentation [35].

4.2 Algorithm & Hyperparameters

The DQN algorithm learns to predict q-values using a neural network. The network is given raw pixels as input signal representing the state of the environment. This enables the agent to greedily choose actions considering the predicted q-values. The agent chooses random actions with probability $\epsilon = 1$ to balance exploration and exploitation. ϵ will be linearly decreased over time, until finally hitting a low barrier of $\epsilon = 0.01$ after $200,000$ frames and continues with this final value.

Table 3: List of hyperparameters.

Hyperparameter	**Value**
minibatch size	32
replay memory size	$100,000$
agent history length	4
target network update frequency	$10,000$
discount factor	0.99
learning rate	0.00001
initial exploration	1
final exploration	0.1
final exploration frame	$200,000$
replay start size	$10,000$

The produced states of the Atari emulator are 210x160 RGB images. To reduce complexity, these observations are rescaled to 84x84 grey-scaled images. In fact, the agent obtains the last four frames as input, because this adds information about directions and speed. Mnih, et al. [19] and Hessel, et al. [4] also used these preprocessing steps.

Table 4: Preprocessing.

Hyperparameter	**Value**
greyscaling	True
observation down-sampling	(84, 84)
frames stacked	4
action repetitions	4
terminal on loss of life	True

Experiences are stored in a replay memory (introduced by Lin [36]) with a capacity of $N = 100,000$ to store the current state, the action the agent took, the obtained reward and the resulting state. Mnih, et al. [19] and Hessel, et al. [4] used a bigger capacity of $N = 1,000,000$. A smaller capacity was chosen due to memory limitations and because it is not necessary for the shortly introduced experiments, where only one specific Atari game will be solved.

The return and multi-step losses will be computed equivalent to how Hessel, et al. [4] did it, as shown in section 3.2. Note that this algorithm consequently lacks importance sampling, which theoretically should be included due to off-policy correction.

The used neural network architecture is derived from the initial paper of Mnih, et al. [3] and is also used by their followed paper [19]. The network is feeded an 84x84x4 image as input data. The first hidden layer is a 2d convolution over the input data with 32 filters, a kernel size of 8 and a stride of 4. The second 2d convolution layer has 64 filters, a kernel size of 4 and a stride of 2. The third convolution layer has again 64 filters with a kernel size of 3 and a stride of 1. After each convolution layer a ReLU-function is applied. The last hidden layer is a fully connected layer with 512 neurons, again a ReLU activation function was used. The output layer represents the q-values for each specific action the agent could take.

Table 5: Network architecture.

Hyperparameter	**Value**
channels	$32, 64, 64$
filter size	8 x 8, 4 x 4, 3 x 3
stride	$4, 2, 1$
hidden units	512
output units	Number of actions

The neural network was trained on batches of size 32. Every $10,000$ steps the weights of the neural network get duplicated to the target network. Adam was used as optimizer with a learningrate of 0.00001. The discount factor γ is set to 0.99.

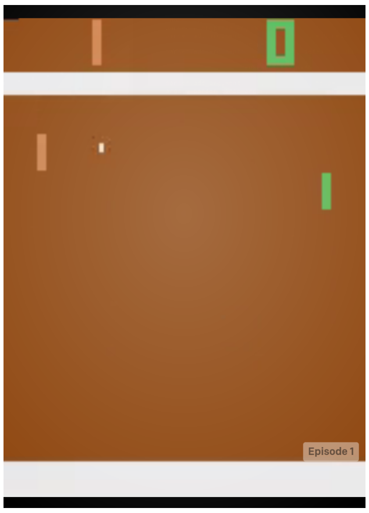


Figure 8: Pong, one of the oldest video games of the world, serves as an environment where the agent plays against a rudimentary AI in a tabletop like game. If one player manages to get the ball behind the opponent, he is rewarded a positive reward of $+1$, while the other one gets a negative reward of -1. The game is played until one player reaches 21 points. The return of an agent could for example be 8, i.e. that he would win 21 games and lose 13 games.

4.3 Methodology

Researchers often evaluate their algorithms on more than just one Atari game, but due to computational limitations, the following experiments will only be benchmarked against one environment called *pong*. To clarify effects of n-step bootstrapping without importance sampling on basic DQN training, the agent plays several games of pong with different values for n. Atari games are widely used as benchmark for reinforcement learning algorithms and pong is one of the easier tasks to learn. This helps lowering computational costs: With the used hyperparameters, pong was solved within about one million frames, using the standard configuration for the step size $n = 1$. Figure 8 further explains the pong environment.

Let's first have a closer look at the basic training process. Training starts at frame 0. The agent executes an action (*no action* is also considered an action) for every preprocessed observation it gets from the Atari emulator. The environment then returns a new state and an associated reward. The frame will subsequently be incremented and the agent will play the next step, until the game eventually ends with a terminal state. A total reward (return) of the game will be stored in a list. The return is calculated like figure 8 explains. A m_{100}-score represents the averaged return of the last 100 games. The agent plays more and more games and hopefully learns to play the game until training ends at some point.

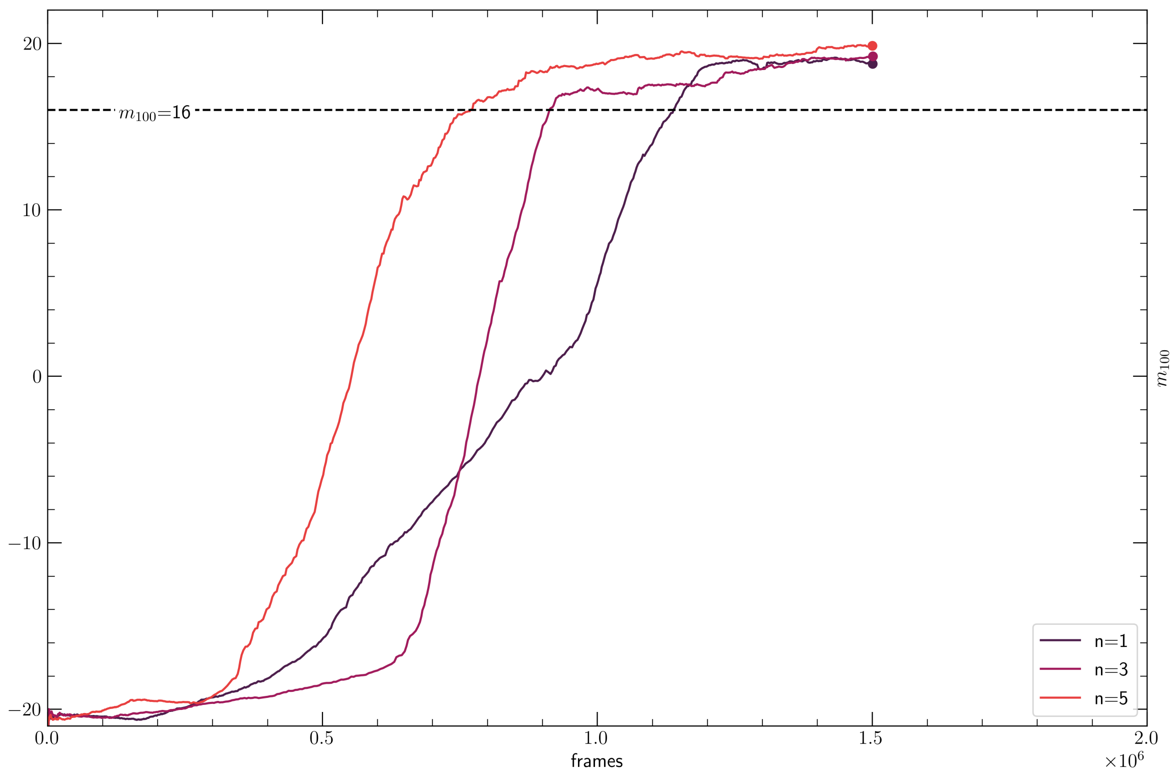


Figure 9: m_{100}-value plotted against time for step size $n \in \{1, 3, 5\}$. The dotted line represents a threshold, where the agent wins 21 and loses 5 games, resulting in a return of 16. (on average for the last 100 games)

Figure 9 plots the averaged returns against time (i.e. frames) and offers some more intuition about training dynamics. The agent loses every game at the beginning, but then quickly starts to learn and beat the pong-AI. Note that the m_{100}-score seems to flatten out at about 16 and then slowly approaches a score of 20-ish. The figure compares the values for the step sizes $n \in \{1, 3, 5\}$. Training automatically ends after roughly 1.5 million frames.

An experiment consists of an agent, who starts learning to play pong until he hits the threshold of $m_{100} = 16$, i.e. until the averaged returns of the last 100 games equal 16. A return of 16 means that the agent wins 21 games and loses 5, respectively (as already explained). This threshold helps lowering computational costs. Each return will be stored in a list along with the total frames played. The last entry of this list can be used to compare convergence between experiments. This experiment will then be repeated ten times for each step size $n \in \{1, 5, 15, 30\}$. The resulting lists of different convergence behaviours can be used to test the hypothesis, that a higher n should benefit training (due to the escaping of the deadly triad), but as n gets higher, this improvement should decrease, because of the lacking off-policy correction. This effect could be countered by a strongly decaying ϵ though, where the behaviour policy approaches the target policy more and more. Finally figure 10 summarizes these experiments visually.

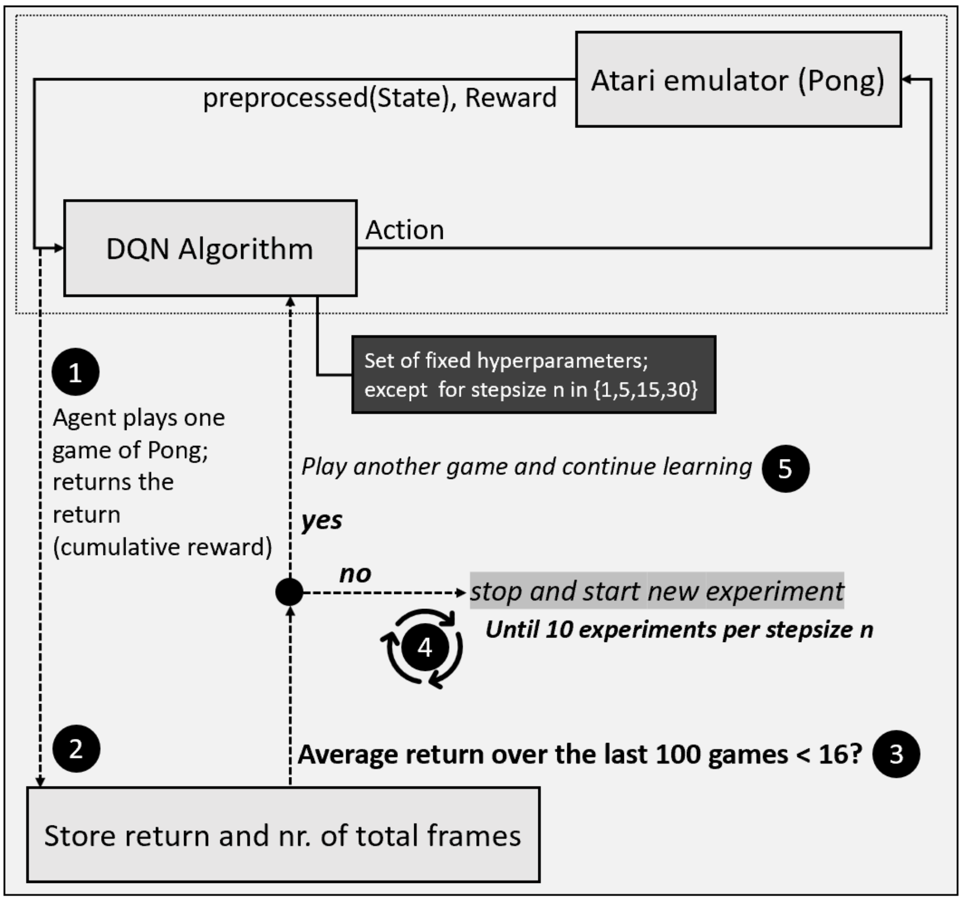

Atari emulator (Pong)
preprocessed(State), Reward
DQN Algorithm
Action
Set of fixed hyperparameters; except for stepsize n in {1,5,15,30}
1
Agent plays one game of Pong; returns the return (cumulative reward)
Play another game and continue learning
5
yes
no
stop and start new experiment
4
Until 10 experiments per stepsize n
2
Average return over the last 100 games < 16?
3
Store return and nr. of total frames

Figure 10: The agent plays a game of pong until either he or his opponent acquire a score of 21 games won (1). Every time the agent wins one rally, he is rewarded a +1 signal, if he loses -1, respectively. The return, along with the number of frames the agent played in total, is then stored (2). If the agent has not managed to get an average return of the last 100 returns of 16 yet, he is going to play another game and continues learning (5). Otherwise the experiment gets stopped (4). Ten experiments for each step size will be executed. This allows for comparing the number of frames necessary to achieve said score for every step size.

4.4 Results

Figure 11 demonstrates the training dynamics for each experiment. The algorithm was executed ten times for each $n \in \{1, 5, 15, 30\}$ resulting in an total of 40 experiments. The y-axis represents the value of m_{100} and is plotted against time (i.e. number of frames). Finished experiments are highlighted with dots ($m_{100} = 16$). The dotted line highlights the averaged frames it took the algorithm with $n = 1$ to achieve $m_{100} = 16$.

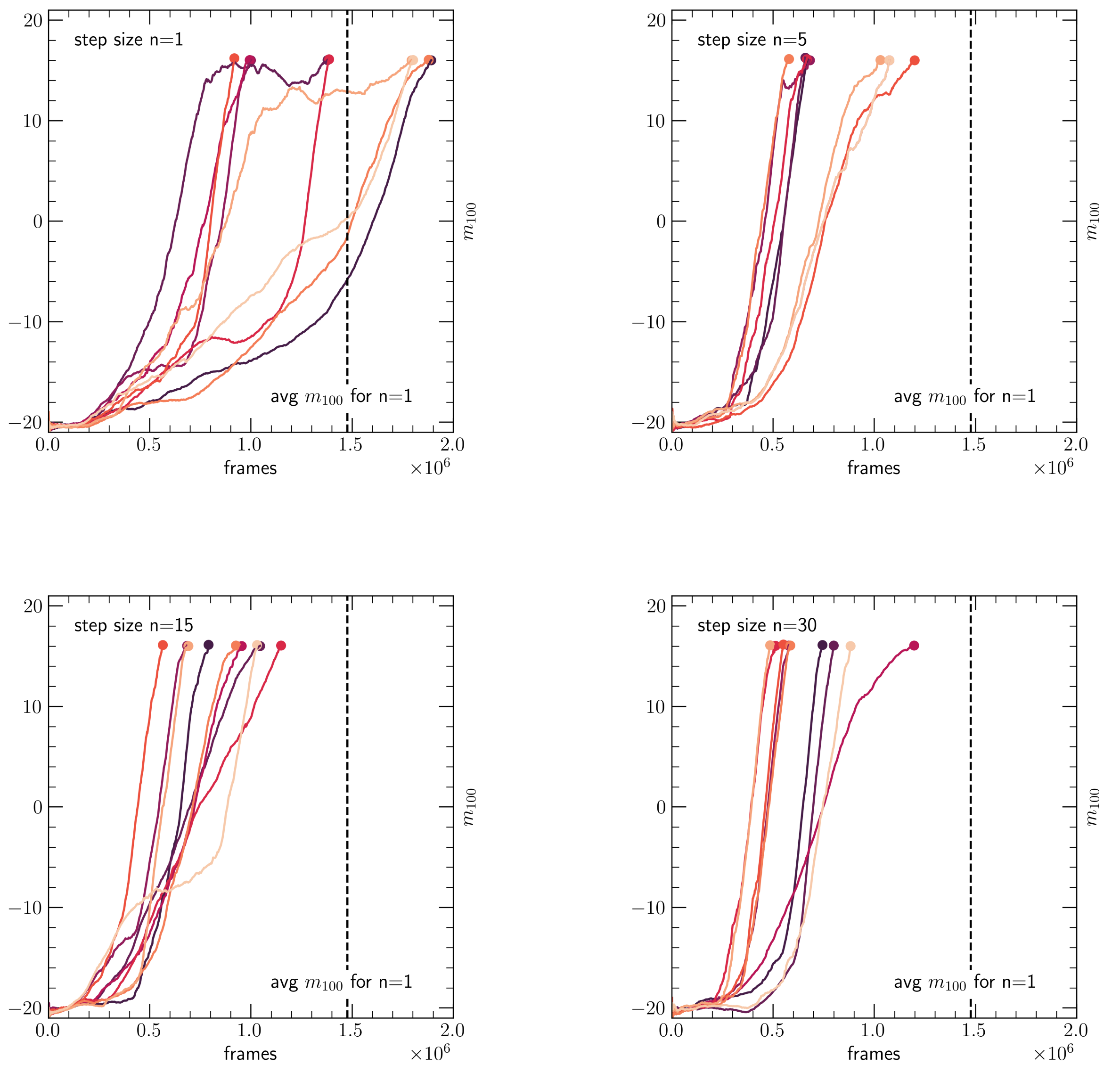


Figure 11: Results of the experiments for each step size.

It seems like a higher n leads to better convergence, but the improvement decreases as n gets higher. The assumption of a performance-worsen effect with increasing n seems to not hold true, but keep in mind, that it still could be, that a higher n would lead to an even better result, if off-policy correction would be considered. This certainly is questionable with $\epsilon = 0.01$ after $200,000$ frames, as the target- and behaviour policy both highly resemble one another at this point.

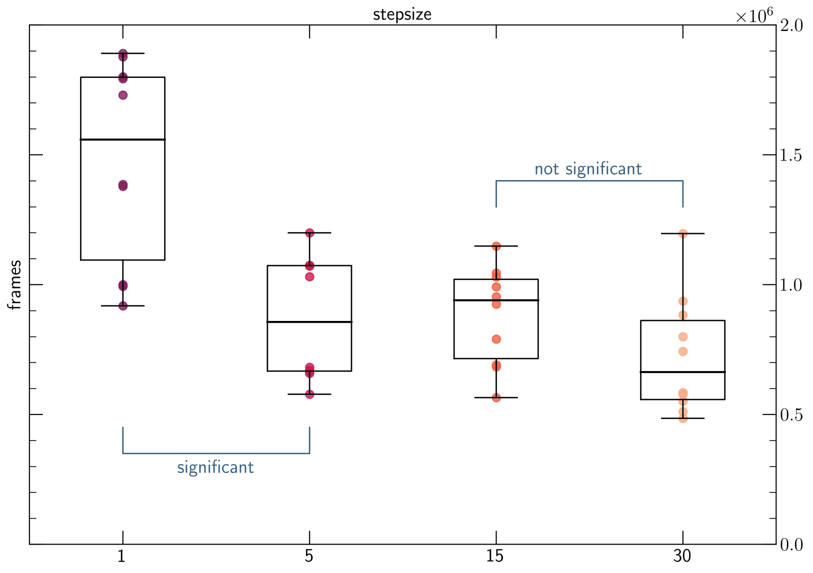


Figure 12: The boxplot visualizes distribution characteristics of the experiments.

Table 10 summarizes the results by tabulating the frames each experiment needs to score $m_{100} = 16$ (figure 11 emphasized these with the dots). This data can be used to further examine the results.

Figure 12 then visualizes these data points presenting deeper understanding of the data distributions. The box plot offers insights for each n and their relation to each other. The frames needed to score $m_{100} = 16$ are plotted against n. It again seems like a higher n improves convergence, but with a diminishing effect of returns. The differences between $n = 1$ and $n = 5$ seem to be significant, while the differences between the other step sizes do not.

Table 6 finally summarizes the data of table 10 with statistical quantities.

Table 6: Mean and variances for the results of the experiments.

n-step	mean	SD
1	1,476,908	373227.396
5	870,375.6	224564.206
15	882,360.3	179786.327
30	726,796.0	217452.113

Let's first check for statistical significant differences between the results of each n and thus define the following null hypothesis: All means for each n are the same. This leads to the alternative hypothesis, that at least two group means have statistical significant differences from one another. This can be checked with the one way ANOVA test. Let's choose a significance level α of 0.05. The critical F-value consequently is 2.866.

Table 7: Results of the ANOVA test.

p	**F**	$\mathbf{df_{between}}$	$\mathbf{df_{within}}$
0.00000193	14.803	3	36

$p < \alpha$ indicates, that we can reject the null hypothesis for the alternative hypothesis, that at least two group means are statistical significant different from one another.

The ANOVA test is a subject to the following three assumptions:

- Each group is normal distributed
- Homogeneity in variances across groups
- Independence of observations

ANOVA is quite robust to violations of the first two criteria though. [37]

The normal distribution for each group could be tested by looking at the histogram or Q-Q plot, but the data is too sparse to see something significant in these plots. Hence, a statistical test in form of the Shapiro-Wilk-Test can be executed, resulting in the following p-values for each group:

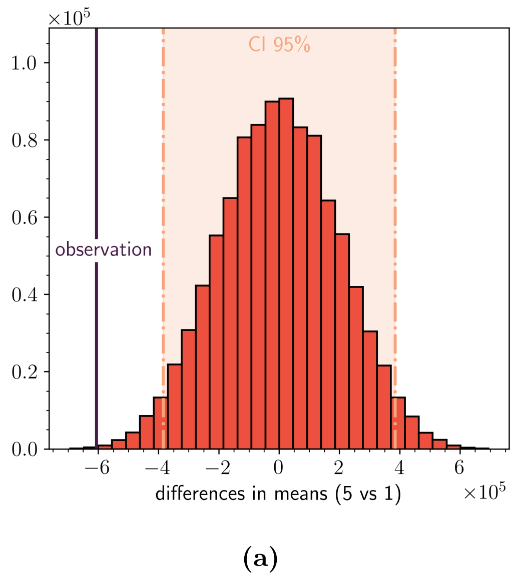
Table 8: Results of the Shapiro-Wilk test.

n	**p**
1	0.0518
5	0.0259
15	0.590
30	0.223

With α again set to 0.05, we would reject the null hypothesis, that all groups are normal distributed, for the $n = 5$ case.

Levene's test is suitable to test homoscedasticity. With $\alpha = 0.05$ and the null hypothesis, that all group variances are the same and the resulting p-value of 0.014, we can reject this hypothesis. This indicates a violation of this assumption.

The independence of observations should be satisfied by nature of this experiment.



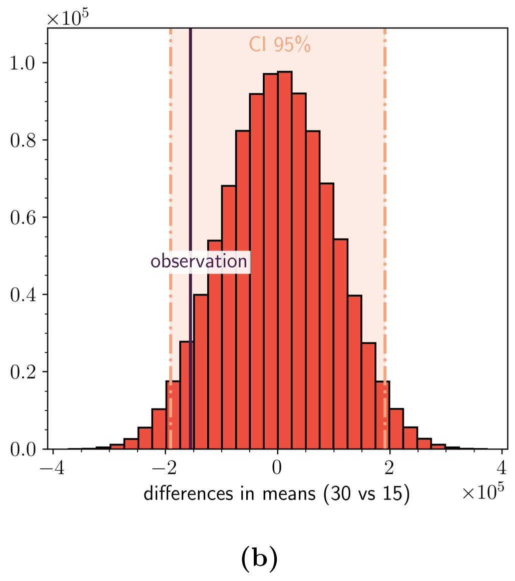


Figure 13: Bootstrap tests for (a) $n = 1$ and $n = 5$ and (b) $n = 15$ and $n = 30$.

We can consequently run the Kruskal-Wallis test as an alternative to the ANOVA test like suggested in [38]. But keep in mind, that the ANOVA test already is quite robust to these violations.

Table 9: Results of the Kruskal-Wallis test.

p	**F**	$\mathbf{df}_{\mathbf{between}}$	$\mathbf{df}_{\mathbf{within}}$
0.0009469	16.381	3	36

The ANOVA test with two violations and the consequently used Kruskal-Wallis test both suggest the hypothesis, that at least two group means are statistical significant different from one another.

Now, that we can assume statistical significant differences in convergence behaviour for at least two step size choices, let's try to further support this assumption by running a bootstrap test. We'll test differences in the number of frames, each group needs to score a m_{100}-value of 16 (see table 10). Let's test the different behaviours for the groups $n = 5$ vs $n = 1$ and $n = 30$ vs $n = 15$ separately. On average, $n = 1$ needs $1,476,908$ frames; $n = 5$ needs $870,375.6$, respectively. The difference of both values consequently is $-606,532$. This observation is also highlighted in figure 13 (similarly for the $n = 30$ vs $n = 15$ case). Figure 13 also shows data distributions for $1,000,000$ random samples along with an confidence interval, where 95% of all random samples lie. We can see, that the observation in the $n = 5$ vs $n = 1$ case is significant, while the observed difference of $n = 30$ vs $n = 15$ could still be obtained by chance. Hence, we can safely assume, that a choice of $n = 5$ is superior to choosing $n = 1$.

Table 10: Datapoints of the experiments.

n-step	**experiment**	**frames needed to $m_{100} = 16$**
1	1	1,890,790
1	2	1,378,547
1	3	1,000,030
1	4	993,340
1	5	1,385,983
1	6	917,916
1	7	1,877,428
1	8	1,793,572
1	9	1,801,014
1	10	1,730,456
5	1	670,595
5	2	658,755
5	3	681,441
5	4	1,073,808
5	5	666,274
5	6	1,199,282
5	7	578,229
5	8	1,030,166
5	9	1,073,808
5	10	1,071,398
15	1	790,326
15	2	1,044,930
15	3	684,073
15	4	953,802
15	5	1,148,642
15	6	564,790
15	7	924,893
15	8	690,775
15	9	1,029,454
15	10	991,918
30	1	743,094
30	2	799,545
30	3	577,574
30	4	1,196,705
30	5	511,407
30	6	550,878
30	7	584,029
30	8	485,525
30	9	882,721
30	10	936,482

5 Conclusion

After the reader was introduced to general ideas behind reinforcement learning, different algorithmic concepts were explored: We started with Monte Carlo methods, where a value of a state or state-action-pair is determined by sampling whole episodes and averaging over obtained rewards. This was followed by an introduction of TD(0) learning, where an agent starts learning immediately after taking one step. This consequently allows true online learning. In this context Q-Learning was presented, where the fundamental question, why Q-Learning lacks off-policy correction, was answered. Then, these two extremes were put into relationship to each other by n-step bootstrapping, which opens a whole spectrum between these two extremes. Afterwards, an algorithm called DQN was introduced, including an implementation with n-step bootstrapping. This improvement to the original implementation was used in an amazing paper from Hessel, et al. [4]. The presented algorithm, called rainbow, achieved state-of-the-art results in 2018. This improvement was chosen consciously for further examination, due to its mathematical ambiguities: Off-policy correction is theoretically necessary for n-step DQN algorithms.

Hessel, et al. [4] chose to ignore off-policy correction, although it is necessary for n-step DQN algorithms. It is unclear, why they chose to ignore it and it is not explicitly stated in their paper. Based on the author's obtained findings, one could guess, that with $\epsilon = 0.01$ being that low, both the target- and behaviour policy resemble one another so closely, that it doesn't really matter.

The conducted experiments clarified training dynamics with n-step bootstrapping, when off-policy correction is being ignored. Different step sizes n were studied. Hessel, et al. [4] claimed that training with $n = 3$ lead to the best results. Bootstrapping solutions between TD(0) and Monte Carlo approaches often are superior, according to Sutton and Barto [9, p. 141]. Implementations like rainbow underline this statement. The experiments also support this claim. An explanation, why this is the case, can be found in section 3.2: A higher n helps escaping the deadly triad.

The author's second assumption, that this positive effect decreases as n gets higher and higher, was not confirmed by these experiments. The assumption was based on the fact, that although the deadly triad was escaped, yet the behaviour and target policy diverged more and more at the same time. As already mentioned, the target and behaviour policy approach each other strongly while ϵ decreases to the final value of 0.01. This could make off-policy correction practically unnecessary. Keep in mind, that off-policy correction could still lead to better results. Nevertheless, it has been shown, that a higher n benefits DQN training, even if off-policy correction is being ignored - like Hessel, et al. [4] probably intended in the first place.

It would be interesting to see how the results with off-policy correction compare to the setup introduced in the context of this paper. The author's guess based on the acquired findings in this work is, that it will behave similarly and will have little impact, if any at all. It would furthermore be useful to know, how the experiment behaves for the whole Atari benchmark. Pong was best solved with a quite high value for n, compared to the choice Hessel, et al. [4] took for the rainbow algorithm.

References

(1) Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine learning* **1988**, *3*, 9–44.

(2) Tesauro, G. In *Applications of Neural Networks*, Murray, A. F., Ed.; Springer US: Boston, MA, 1995, pp 267–285.

(3) Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; Riedmiller, M. Playing Atari with Deep Reinforcement Learning, 2013.

(4) Hessel, M.; Modayil, J.; Van Hasselt, H.; Schaul, T.; Ostrovski, G.; Dabney, W.; Horgan, D.; Piot, B.; Azar, M.; Silver, D. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018; Vol. 32.

(5) Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Dębiak, P.; Dennison, C.; Farhi, D.; Fischer, Q.; Hashme, S.; Hesse, C., et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680* **2019**.

(6) Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P., et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* **2019**, *575*, 350–354.

(7) Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; Hassabis, D. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* **2018**, *362*, 1140–1144.

(8) Irpan, A. Deep Reinforcement Learning Doesn't Work Yet, `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018.

(9) Sutton, R. S.; Barto, A. G., *Reinforcement learning: An introduction*; MIT press: 2018.

(10) BELLMAN, R. A Markovian Decision Process. *Journal of Mathematics and Mechanics* **1957**, *6*, 679–684.

(11) Bellman, R.; Bellman, R.; Corporation, R., *Dynamic Programming*; Rand Corporation research study; Princeton University Press: 1957.

(12) Weng, L. A long Peek into Reinforcement Learning, https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html, 2018.

(13) Graesser, L.; Keng, W., *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*; Addison-Wesley Data & Analytics; Addison Wesley: 2019.

(14) Achiam, J. Spinning Up in Deep Reinforcement Learning. **2018**.

(15) Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T., et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* **2020**, *588*, 604–609.

(16) Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* **1992**, *8*, 229–256.

(17) Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T. P.; Harley, T.; Silver, D.; Kavukcuoglu, K. Asynchronous Methods for Deep Reinforcement Learning, 2016.

(18) Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal Policy Optimization Algorithms, 2017.

(19) Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature* **2015**, *518*, 529–533.

(20) Bellemare, M. G.; Dabney, W.; Munos, R. A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887* **2017**.

(21) Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; Wierstra, D. Continuous control with deep reinforcement learning, 2019.

(22) Haarnoja, T.; Zhou, A.; Abbeel, P.; Levine, S. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, 2018.

(23) Silver, D. Introduction to reinforcement Learning, Lecture 4: Model-Free Prediction, https://www.davidsilver.uk/wp-content/uploads/2020/03/MC-TD.pdf, 2015.

(24) Samuel, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of research and development* **1959**, *3*, 210–229.

(25) Holland, J. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. *Machine learning, an artificial intelligence approach* **1986**.

(26) Witten, I. H. An adaptive optimal controller for discrete-time Markov environments. *Inf. Control.* **1977**, *34*, 286–295.

(27) Sutton, R. S. Temporal Credit Assignment in Reinforcement Learning. **1985**.

(28) Watkins, C. J. C. H. Learning from delayed rewards. **1989**.

(29) Schaul, T.; Quan, J.; Antonoglou, I.; Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* **2015**.

(30) Van Hasselt, H.; Guez, A.; Silver, D. In *Proceedings of the AAAI conference on artificial intelligence*, 2016; Vol. 30.

(31) Wang, Z.; Schaul, T.; Hessel, M.; Hasselt, H.; Lanctot, M.; Freitas, N. In *International conference on machine learning*, 2016, pp 1995–2003.

(32) Van Hasselt, H.; Doron, Y.; Strub, F.; Hessel, M.; Sonnerat, N.; Modayil, J. Deep reinforcement learning and the deadly triad. *arXiv preprint arXiv:1812.02648* **2018**.

(33) Horgan, D.; Quan, J.; Budden, D.; Barth-Maron, G.; Hessel, M.; Van Hasselt, H.; Silver, D. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933* **2018**.

(34) Bellemare, M. G.; Naddaf, Y.; Veness, J.; Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* **2013**, *47*, 253–279.

(35) Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; Zaremba, W. OpenAI Gym, 2016.

(36) Lin, L.-J. *Reinforcement learning for robots using neural networks*; tech. rep.; Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

(37) One-Way ANOVA, `https://statistics.laerd.com/statistical-guides/one-way-anova-statistical-guide.php`.

(38) Scipy Documentation, `https://docs.scipy.org/doc/scipy-0.13.0/reference/index.html`.

List of Figures

List of Tables