Springer 2024

Algorithmic Foundations of Reinforcement Learning

Stephan Pareigis

Department of Informatics, Hamburg University Of Applied Sciences, Berliner Tor 7, 20099 Hamburg, Germany.

Abstract

A comprehensive algorithmic introduction to reinforcement learning is given, laying the foundational concepts and methodologies. Fundamentals of Markov Decision Processes (MDPs) and dynamic programming are covered, describing the principles and techniques for addressing model-based problems within MDP frameworks. The most significant model-free reinforcement learning algorithms, including Q-learning and actor-critic methods are explained in detail. A comprehensive overview of each algorithm's mechanisms is provided, forming a robust algorithmic and mathematical understanding of current practices in reinforcement learning.

Keywords: reinforcement learning, MDP, markov-decision process, dynamic programming, deep reinforcement learning, SARSA, Q-learning, DQN, REINFORCE, A2C, PPO, DDPG, SAC, policy gradient methods, exploration vs exploitation, sparse rewards, robotics, offline reinforcement learning

1 Introduction

The article provides a comprehensive introduction and overview over the algorithmic foundations of Reinforcement Learning (RL).

Reinforcement Learning forms an important part of artificial intelligence, characterized by learning optimal decision-making through interactions with dynamic environments. The field is characterized by numerous technological applications, including autonomous systems, strategic game playing, and complex decision-making processes.

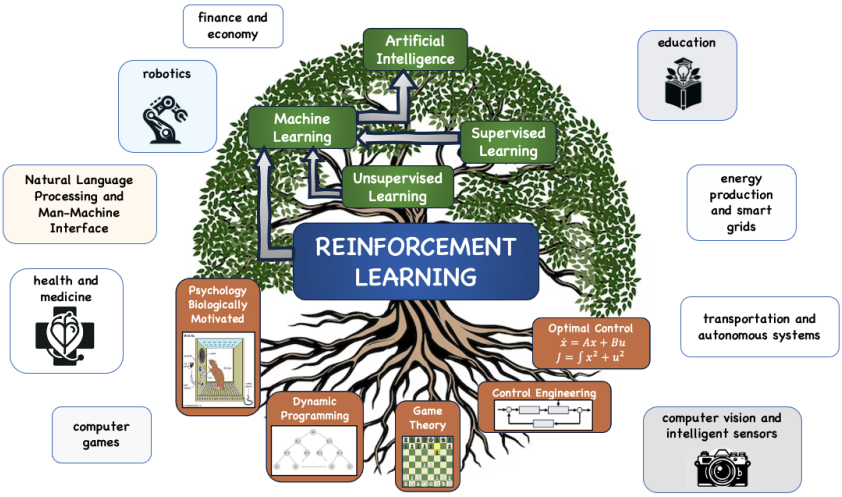


Fig. 1: The Roots of Reinforcement Learning: A visual map illustrating the interdisciplinary nature of Reinforcement Learning (RL). RL is considered a subfield of Artificial Intelligence. its relationship with Machine Learning, Supervised and Unsupervised Learning, and connections to Dynamic Programming, Game Theory, and Control Engineering. The diagram further branches out to show RL's diverse applications in sectors such as healthcare, finance, robotics, energy, education, computer vision, and more, underlining its profound impact across industries.

Figure 1 illustrates the algorithmic and theoretical roots of reinforcement learning, its categorization within the area of artificial intelligence, and important fields of application. RL can be categorized as a distinct type of machine learning, next to supervised learning and unsupervised learning. Its algorithmic roots lie within optimal control theory and dynamic programming. Applications include problems in which sequential decisions have to be made in order to optimize a given objective function.

A comprehensive book on reinforcement learning is R. Sutton's and A. Barto's book Reinforcement Learning: An Introduction [1]. Other books which cover the basics of reinforcement learning including practical examples and modern research areas and applications are M. Lapan's book on Deep Reinforcement Learning [2] and the workshop book from Palmas et.al. [3]. [4] gives theoretical background on modern Deep RL methods like PPO and A2C.

Chapter 2 gives an introduction to the theoretical concepts of reinforcement learning. Chapter 2.1 covers the key concept of a Markov Decision Process (MDP). The problem setting is to find an optimal strategy for the MDP in form of a policy which optimizes the total reward. If full knowledge of the MDP is given, dynamic programming principles can be applied to obtain an

optimal strategy. Chapter 2.2 explains how the Bellman Equation is used to iteratively approximate a solution.

If no model of the MDP is given, then exploration methods must be used to learn from the interactions with the MDP as covered in chapter 3. The methods are based upon dynamic programming principles as described previously, and they can be devided into on-policy and off-policy methods.

When observation spaces are based on image inputs from cameras or are otherwise too large to handle, then artificial neural networks are used to approximate an optimal strategy. This area is referred to as deep reinforcement learning. Obtaining a solution for the optimization problem can either be done based on the approximation of an action-value function as described in chapter 3.2 or the approximation of a policy as explained in chapter 3.3. A combination of value and policy based methods are actor-critic methods as described in chapter 3.4.

If the action space is continuous, then specially designed methods are used as carried out in chapter 3.5.

2 Reinforcement Learning Fundamentals

The theoretical foundation of reinforcement learning is based upon Markov Decision Processes (MDPs), which provide a mathematical framework for modeling decision-making in stochastic environments. The central objective is to identify an optimal decision-making strategy that maximizes a specified objective function. The chapter covers the definition of an MDP, derives a policy-induced trajectory distribution, introduces value functions, and the Bellman Equation. The chapter closes with dynamic programming principles which are used to obtain a value function given full knowledge of the underlying MDP.

2.1 Markov Decision Processes (MDPs)

The following basic definitions from stochastic processes are frequently used in the formulation of MDPs and Reinforcement Learning.

Definition The *conditional probability* of an event A given an event B is defined as $P(A|B) := \frac{P(A \cap B)}{P(B)}$.

If events A and B are independent, then $P(A|B) = P(A)$. In this case, the probability that both events A and B occur can be calculated as

$$P(A \cap B) = P(A) \cdot P(B). \tag{1}$$

The product rule for independent events is essential for calculations in MDPs.

Definition If X is a random variable and p is its probability distribution, then the *expected value* of X is defined as

$$\mathbb{E}[X] := \int x \cdot p(X = x)\, dx \quad \text{or} \quad \mathbb{E}[X] := \sum_x x \cdot p(x) \tag{2}$$

depending on whether X is continuous or discrete.

2.1.1 Elements of an MDP

An MDP is composed of the following components:

- **States ($\mathcal{S}$):** A state represents a specific situation in the environment, the physical world, or system to be controlled. A state space may be high dimensional like the raw input from a camera. It may also be low dimensional like a 2D grid or a feature space. An MDP models a system that transitions between different states.
- **Observations (O):** Ideally all states can be fully observed. In this case $\mathcal{S} = O$. If not all states of an MDP can be fully observed, then it is called a *partially observable MDP* or POMDP. The *observation space* is used in RL to describe the set of all possible observable input (sensor input, feature information, odometric information, etc).
- **Actions (A):** Actions are the decisions or moves that an agent can take in a given state. The set of all possible actions in a state is denoted as $A(s)$.
- **Transitions $\mathcal{P}$:** The transition probability function

$$P : \mathcal{S} \times A \times \mathcal{S} \to [0, 1], \qquad (s, a, s') \mapsto P(s'|s, a) \in [0, 1] \tag{3}$$

  describes the probability of transitioning from one state to another given a particular action. It is often represented as $P(s'|s, a)$, denoting the probability of transitioning to state s' from state s by taking action a.
  The Markov property of an MDP states that the transition probability to a state s_{t+1} is only dependent on the predecessor state s_t and action a_t, not on past states, i.e. a MDP is memory-less

$$p(s_{t+1}|s_t, a_t) = p(s_{t+1}|a_t, s_t, a_{t-1}, s_{t-1}, \cdots).$$

- **Rewards R:** Each state-action-state triple receives a numerical reward

$$R : \mathcal{S} \times A \times \mathcal{S} \to \mathbb{R}.$$

  The immediate reward is denoted as $R(s'|s, a)$. The expected immediate reward in a state s with action a can be expressed as

$$\mathbb{E}[R(s, a)] = \sum_{s' \in \mathcal{S}} P(s'|s, a) \cdot R(s'|s, a) \tag{4}$$

- **Policy (π):** A policy is a strategy that the agent follows to choose actions in each state. It can be deterministic

$$\pi : S \longrightarrow A, \quad \pi(s) = a \in A \tag{5}$$

or stochastic

$$\pi : A \times S \longrightarrow [0,1], \quad \pi(a|s) = p \in [0,1]. \tag{6}$$

When a policy π is given, the expected immediate reward in a state s can be written as

$$\mathbb{E}[R(s)] = \sum_{s' \in S, a \in A} \pi(a|s) \cdot P(s'|s,a) \cdot R(s'|s,a) \tag{7}$$

- **Realizing an MDP:** A sequence of states $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \cdots, s_T)$ is called a realization of an MDP for some starting state s_0, when a_t is choosen according to π, i.e.

$$a_t = \pi(s_t) \text{ or choosen from the distribution } a_t \sim \pi(.|s_t)) \tag{8}$$

  and $r_t = R(s_{t+1}|s_t, a_t)$. τ is also called trajectory or path. T can be a finite or infinite time horizon.
- **Total Return:** The cumulated reward or total return G for some realization τ is defined as

$$G := \sum_{t=0}^{T-1} \gamma^t R_t, \quad R_t = R(s_{t+1}|s_t, a_t) \text{ along } \tau \tag{9}$$

  for some discount factor $\gamma \in]0,1]$. The goal in RL is to find a policy $\pi^\star$ which maximizes the expected total reward $\mathbb{E}_{\tau \sim \pi}[G]$. $\tau \sim \pi$ denotes a realization of a trajectory given a policy π.
  The discount factor γ may be set to 1 for MDPs which terminate after a finite time. For infinite horizon MDPs a discount factor $\gamma < 1$ is necessary to ensure existence of G.

2.1.2 Policy Induced Trajectory Distribution $\tau \sim \pi$

For a given policy π every concrete trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \cdots, s_T)$ has a certain probability. The probability distribution for trajectories for a fixed policy π will be derived in this section.

According to (3) the probability for ending up in state s_1 when starting in state s_0 with a given action a_0 is $p(s_1|s_0, a_0)$. If a_0 is chosen according to the given policy $a_0 \sim \pi(.|s_0)$, then the probability for the sequence (s_0, a_0, s_1) is

$$P((s_0, a_0, s_1)) = \pi(a_0|s_0) \cdot P(s_1|s_0, a_0) \tag{10}$$

Policy Induced Trajectory Distribution Using the memory-lessness of an MDP together with the product rule for independent probabilities (1), the probability for a specific trajectory τ_{s_0} starting in state s_0 can be calculated

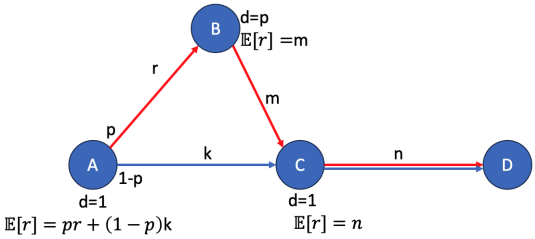


Fig. 2: The example illustrates equation (12) and shows a Markov Prozess which allows a red path $\tau_{\text{red}} = (A, B, C, D)$ and a blue path $\tau_{\text{blue}} = (A, C, D)$. The rewards are k, r, m, n as shown on the edges of the graph. The probability for τ_{red} is $P(\tau_{\text{red}}) = p \cdot 1 \cdot 1$. The probability for τ_{blue} is $P(\tau_{\text{blue}}) = (1-p) \cdot 1$. The expected total return for A is therefore $\mathbb{E}[G] = p{\cdot}(r{+}m{+}n){+}(1{-}p){\cdot}(k{+}n)$. Using the occupancy measure and expected one-step rewards in each state as shown in the image, alternatively the expected return can be calculated as $\mathbb{E}[G] = \sum_{s \in S} d(s) \cdot \mathbb{E}[r(s)] = 1 \cdot (pr + (1-p)k) + p \cdot m + 1 \cdot n$.

as the product of the probabilities in (10)

$$P_\pi(\tau) = P_\pi(s_0, a_0, \cdots, s_T) = \prod_{t=0}^{T-1} \pi(a_t|s_t) \cdot P(s_{t+1}|s_t, a_t) \tag{11}$$

Equation (11) is called the *policy induced trajectory distribution* for $\tau \sim \pi$.

This probability can be used to calculate the expected value of the total return $\mathbb{E}_{\tau\sim\pi}[G]$ over all trajectories τ starting in state s_0.

Equation (12A) uses the definition of the expected value. (12B) shows that $\mathbb{E}_{\tau\sim\pi}[G]$ may be expressed as expected values of local rewards, using the *occupancy measure* d^π.

Using (9) and (11) the expected total return for G_π can be calculated as

$$\mathbb{E}_{\tau\sim\pi}[G_\pi] \overset{A}{=} \sum_\tau P_\pi(\tau) \cdot G_\pi(\tau) \overset{B}{=} \sum_s d^\pi(s) \cdot \mathbb{E}[R(s)] \tag{12}$$

Occupancy Measure The value d^π as introduced in equation (12) is called the *discounted occupancy measure*.

$$d^\pi(s) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T-1} \gamma^t \cdot \mathbb{I}(s_t = s)\right]$$

where $\mathbb{I}(s_t = s)$ is 1 if the $s_t = s$. The value d^π expresses the discounted frequency of visiting state s given a policy π. Figure 2 explains the property (12B) in a simple example. The expected total return in state A can hence be expressed in two different ways.

The following sections will deal with the expected total return, which will be called value function V, and methods for calculating it.

2.1.3 Value Functions

The objective is to find an optimal policy that maximizes the expected cumulative reward or total return. If in every state $s \in S$ a prediction of the expected total return G_s were known, then an optimal action a could be chosen which leads to a subsequent state in which the prediction of the expected total return is maximized. Such a prediction is called a state value function V.

State Value Function (V): The *state value function* $V(s)$ for a policy π in an MDP is defined as the expected total return starting from state s and following policy π thereafter. Mathematically, it can be expressed as:

$$V^{\pi}(s) := \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T-1}\gamma^k R_t \mid S_0 = s\right] \tag{13}$$

In order to be able to calculate a greedy action or policy based on the state value function V^{π}, the transition properties of the MDP must be known. A greedy policy $\pi^{*,\pi}$ with respect to some value function V^{π} (based on a fixed policy π) is then calculated as

$$\pi^{*,\pi}(s) = \arg\max_{a\in A}\left[\sum_{s'\in S} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')]\right] \tag{14}$$

In each state s the action $a^* = \pi^{*,\pi}(s)$ is chosen, which maximizes the expected total return when taking one step with a^* and continuing with policy π thereafter. Note that $\pi^{*,\pi}$ is generally not an optimal policy. It is a greedy policy based on the value function V^{π} of given policy π.

If the transition properties of the MDP are not known, then the action value function Q^{π} can serve as a prediction for the expected total return.

Action Value Function (Q): The *action-value function* $Q(s,a)$, for a policy π, in an MDP is defined as the expected return starting from state s, taking an action a, and thereafter following policy π. It can be formally written as:

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T-1}\gamma^k R_t \mid S_t = s, A_t = a\right] \tag{15}$$

The difference to the above is, that Q also depends on an action a. This way an optimal action can be chosen by taking the action a^* which maximizes the Q-value in the current state s:

$$a^* = \arg\max_{a\in A} Q(s,a) \tag{16}$$

Comparing (14) and (16) shows that in the latter case no transition information of the MDP is necessary in order to choose a greedy action in every state.

2.1.4 Bellman Equations

The Bellman Equations form an essential component of the algorithmic structure of dynamic programming and reinforcement learning. They express that the value of a state can be expressed as an immediate reward plus the value in the next state.

The **Bellman Expectation Equation** for the **state value function** V given a policy π is given by

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \cdot \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')] \tag{17}$$

The **Bellman Expectation Equation** for the **action value function** Q given a policy π is given by

$$Q^{\pi}(s,a) = \sum_{s' \in S} P(s'|s,a) \cdot \left[R(s,a,s') + \gamma \sum_{a' \in A} \pi(a'|s') Q^{\pi}(s',a') \right] \tag{18}$$

The **Bellman Optimality Equation** for the **state value function** is:

$$V^{*}(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a) \cdot [R(s,a,s') + \gamma V^{*}(s')] \tag{19}$$

The **Bellman Optimality Equation** for the **action value function** Q is given by:

$$Q^{*}(s,a) = \sum_{s' \in S} P(s'|s,a) \cdot [R(s,a,s') + \gamma \max_{a' \in A} Q^{*}(s',a')] \tag{20}$$

The Bellman Equations form a basis for the following algorithms.

2.2 Solving an MDP: Dynamic Programming

If the transition information of the MDP is known, then equation (17) can be applied iteratively to calculate an optimal policy π^{*}. The methods are explained in the following section.

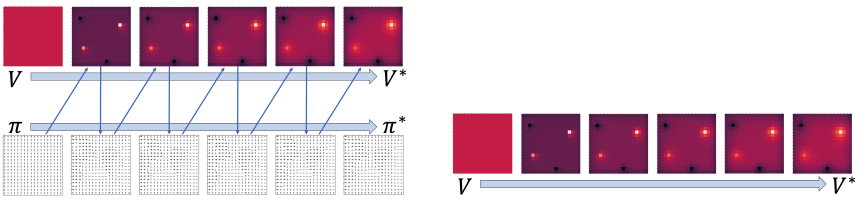
2.2.1 Policy Iteration and Value Iteration

In **Policy Iteration** an optimal policy is directly calculated using the following two steps:

1. **Policy Evaluation:** Compute V^{π} for the current policy, iteratively applying the Bellman Expectation Equation (17).
2. **Policy Improvement:** Update the policy π to a greedy policy $\pi^{*,\pi}$ by selecting in every state the action which maximizes the expression in the equation (14).

3. Repeat until convergence.

Figure 3a shows the progression of the algorithm for a simple grid example. The heat maps in the top row show the value function, the bottom row shows the greedy policy with respect to the current value function.
In **Value Iteration** the optimal value function V^* is calculated directly using iterative application of the Bellman Optimality Equation (19). It can be said that policy evaluation and policy improvement are integrated into a single update, iterating until convergence as shown in figure 3b.



(a) Policy Iteration (b) Value Iteration

Fig. 3: In *policy iteration* (left) a policy is evaluated and then improved acting greedily. Often it is not necessary to calculate the value function precisely as the policy converges fast to an optimal policy. In *value iteration* (right) the optimal value function is directly approximated using the Bellman Optimality Equation (19).

2.2.2 Dynamic Programming Iteration Methods

There exist various methods for the sequence in which the states are be updated when applying policy evaluation or value iteration. The methods are depicted in figure 4.
Synchronous Backups: In each iteration, the algorithm computes the new value for every state based on the old values (from the previous iteration) and updates them all at once at the end of the iteration. This method ensures that the updates in one iteration are independent of each other, providing stability in the convergence process.
In-Place Dynamic Programming: The algorithm updates the value of a state as soon as the new value is computed, using these updated values for subsequent state value computations within the same iteration. It can lead to faster convergence as it uses the most updated information available.
Prioritized sweeping The algorithm prioritizes updates for states where the value change is expected to be largest. This way, it uses computational resources more effectively.
Real-Time Dynamic Programming (RTDP): RTDP involves executing trials (realizations) updating the value of states that are encountered in these

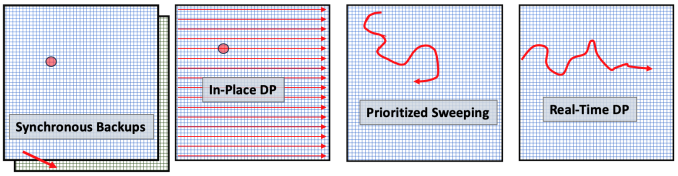


Fig. 4: Different methods for iterating through the state space: Synchronous Backups, In-Place Dynamic Programming, Prioritized Sweeping, RTDP. All the methods require that the transition information of the MDP is given.

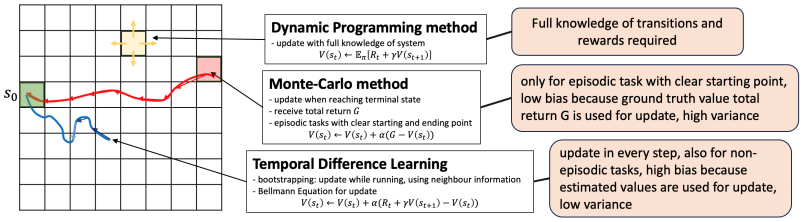


Fig. 5: Overview of update methods: The diagram illustrates three strategies. Dynamic Programming (DP) depicted by the yellow arrows requires full knowledge of state transitions and rewards. Monte-Carlo methods learn from complete episodes and update values based on an empirical mean of total returns. Temporal Difference Learning, also known as bootstrapping, updates values incrementally using neighbor information, applicable to both episodic and non-episodic tasks.

trials. It is particularly useful in large or continuous state spaces and where the full model of the environment is not available.

2.3 Model-free prediction

The previous methods were based on a given model of the MDP. The iterative application of the Bellman Equation required knowledge of the transition properties of the MDP. In the model-free case, value functions have to be learned through observation of the system. These methods can be categorized into *learning from complete episodes* and *learning in temporal differences*. Figure 5 figuratively shows the difference between the methods dynamic programming (DP), Monte-Carlo Methods (MC), and temporal difference methods (TD).

2.3.1 Learning from complete episodes: Monte-Carlo Method

Monte-Carlo methods (MC) estimate the value function based on complete episodes of experience. The value of a state is computed as the average of the returns following that state over many episodes. An episode must terminate for MC methods to update the value function, making them suitable for episodic tasks.

Update Rule The MC updates the value function in every state using an empirical mean calculated from the total returns

$$V_{n+1} \leftarrow \frac{1}{n} \cdot \sum_{i=1}^{n} G_i = V_n + \frac{1}{n}(G_n - V_n) \tag{21}$$

Equation (21) shows that the update method can be implemented using only the previous value of the value funcation approximation V_n, the number of visits n and the current total return G_n.

Aspect	**Monte-Carlo (MC)**	**Temporal Difference (TD)**
Definition	Empirical average returns from complete episodes.	Empirical estimation of the value function based on the rewards and estimated values of subsequent states.
Evaluation	$V_{n+1} \leftarrow \alpha \cdot (G_n - V_n)$	Bellman Expectation Equation $V(s_t) \leftarrow V(s_t) + \alpha\,[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$
Bias	Unbiased estimates of the value function, as they rely on actual returns observed in complete episodes.	Biased estimates due to bootstrapping (using estimates to make further estimates), but can lead to faster convergence.
Variance	High variance, as estimates are directly influenced by the complete returns of individual episodes.	Lower variance compared to MC, as updates are more incremental and smoothed over multiple steps.

Table 1: Comparison of Monte-Carlo and Temporal Difference Learning

2.3.2 Bootstrapping: Temporal difference learning

Temporal Difference (TD) learning combines the Monte-Carlo idea of updating empirically from observed values and Dynamic Programming (DP) ideas using adjacent information for an update. The fact that TD learning uses estimates based on other learned estimates is known as *bootstrapping*.
Update Rule TD(0) The update rule for the value function $V(s)$ is given by:

$$V(s_t) \leftarrow V(s_t) + \alpha\,[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

TD(λ) Learning TD(λ) is an extension of basic TD(0) in which more information from further away along the trajectory is involved in the update process. To implement this idea the concept of *eligibility traces* is introduced. These are temporary records that assign credit to states based on how recently and frequently they have been visited. Eligibility traces decay over time, controlled by the parameter λ, which ranges from 0 to 1. Figure 13c shows the decay rate for eligibility traces of different lengths.

3 Reinforcement Learning Algorithms

This section covers RL algorithms. In contrast to the assumptions in the previous chapter it is now assumed that the agent has no model or knowledge of the underlying MDP.

3.1 Model-free reinforcement learning

Unlike model-based approaches, model-free methods such as SARSA and Q-Learning enable agents to learn optimal policies directly from interactions with the environment. Table 2 shows an overview over the methods.

Category	Subcategory	Definition
Value-Based	**On-Policy** SARSA	Estimation of the value function, using data from the policy currently being improved.
	Off-Policy Q-learning	Estimation of the value function, using data from a different policy than the one being optimized.
Policy-Based	**On-Policy** REINFORCE	Directly learns the policy function using data generated by the current policy.
	Off-Policy	Directly learns a policy function from experiences gathered by a different policy, allowing learning from past behaviors.
Actor-Critic	**On-Policy** PPO	Learn from experiences under current policy keeping policy and value function aligned.
	Off-Policy DDPG, SAC	Uses behaviour policy to learn, has to correct the differences in probability distribution using importance sampling.

Table 2: Categorization of Model-Free RL Methods. Typical algorithms are mentioned in the Subcategory: SARSA, Q-learning, and REINFORCE. Actor-Critic methods use a combination of both and are described in section 3.4.

ε-greedy exploration Model-free RL methods need a way to explore the state space. This can be done using *ε-greedy exploration*. An ε-greedy policy chooses a greedy action and with probability ε it chooses a random action:

$$\pi(a|s) = \begin{cases} \arg\max_{a \in \mathcal{A}(s)} Q(s,a) & \text{with probability } 1-\epsilon, \\ \text{a random action from } A(s) & \text{with probability } \epsilon. \end{cases} \tag{22}$$

Another exploration method which is generally superior to ε-greed exploration is explained in figure 8.

3.1.1 SARSA: On-policy learning

SARSA is an on-policy RL algorithm that updates the Q-values based on the current policy. The Q-value update rule is based on the Bellman Equation (18)

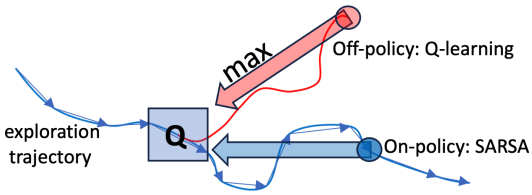


Fig. 6: Comparison between off-policy Q-learning and on-policy SARSA. It illustrates an exploration trajectory taken by an agent (blue), where the Q-learning approach selects the action that maximizes the Q-value (hence "off-policy" as it may choose an action not from the current policy), whereas SARSA selects the next action based on the current policy (hence "on-policy") and updates its Q-values based on the action actually taken.

and is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R + \gamma Q(s',a') - Q(s,a)\right] \tag{23}$$

In its general form SARSA uses ε-greedy exploration with decreasing ε during the training process.

SARSA is on-policy because it updates its Q-values based on the actions chosen by its current policy (ε-greedy). This makes SARSA more sensitive to the exploration-exploitation trade-off.

3.1.2 Q-learning: Off-policy learning

The Q-learning update rule is based on the Bellman Equation (20) and is given by

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \tag{24}$$

In its general form Q-learning also uses ε-greedy exploration with decreasing ε during the training process. However, in the update rule (24) the update is made with a greedy action, as opposed to a ε-greedy action in SARSA. The update rule therefore updates in a greedy way whereas the exploration is done with random actions with probability ε. Q-learning is therefore an off-policy RL algorithm, whereas SARSA learns the action-value function for an ε-greedy policy.

Figure 6 illustrates the differences between on-policy and off-policy learning. On-policy methods converge slower and are more conservative in the policy they evaluate. This is beneficial in environments where taking certain actions can lead to disastrous outcomes. Historic experiences are useless in the training process.

Off-policy methods converge faster because they can learn from experiences generated by a different policy. Replay buffers with historic data may be used effectively.

The following table 3 compares the two methods on-policy and off-policy.

	Off-Policy	**On-Policy**
Properties	Learns from experiences generated by a different policy than the one being improved.	Learns from the experiences generated by the current policy.
Advantages	- Can use data from previous policies - Efficient use of experience (replay buffer) - Suitable for Q-learning and DDPG - Can learn optimal policy while following an exploratory or safe policy	- Updated with the most recent data - Tight coupling between the policy being evaluated and the policy generating the data - Suitable for algorithms like A3C and PPO
Disadvantages	- Can be less stable due to the variance from different behavior policies - Requires careful handling of the importance sampling	- Less efficient use of data as it cannot learn from old experiences - Can be slower to converge as it continually updates from the same policy's experiences

Table 3: Comparison of Off-Policy and On-Policy Learning

3.2 Deep Q-learning

Q-learning has initially been used for problems with small state spaces for which grids or table based methods can be used to approximate the action value function. For large state spaces, artificial neural networks can be used to approximate the action value function.

Artificial neural networks (ANN) are designed to map input values to output values. In reinforcement learning, states (from observations) have to be mapped to values of an action value function. However, during the training process and the exploration phase the output values vary as new experiences are made. This leads to stability problems when training ANNs. Deep Q-learning [5] is an algorithm which accounts for the mentioned stability problems.

Figure 7 shows the topology of the ANN. The following components assist in making the training process stable:

- **Experience Replay Buffer.** Past experiences of the agent are stored in an experience replay buffer (state, action, reward-triplets). Random samples are taken from this buffer to train the ANN. This helps to reduce correlation between consecutive learning updates.
- **Target Network.** A separate target network with identical topology is used to generate Q-values for the training update. This makes the target more stable and therefore stabilizes the training process of the prediction network. Weights of the target network are updated with less frequency or gradually with a soft update strategy.
- **Fixed Replay buffer.** A certain percentage of the replay buffer may be kept fixed and is regularly shown to the ANN. This assures that the neural network does not forget learnings early in the training process.
- **Prioritized Experience Replay PER.** Experiences which incorporate a high TD error (difference between the predicted Q-value and the target Q-value) are shown to the ANN with a higher priority.

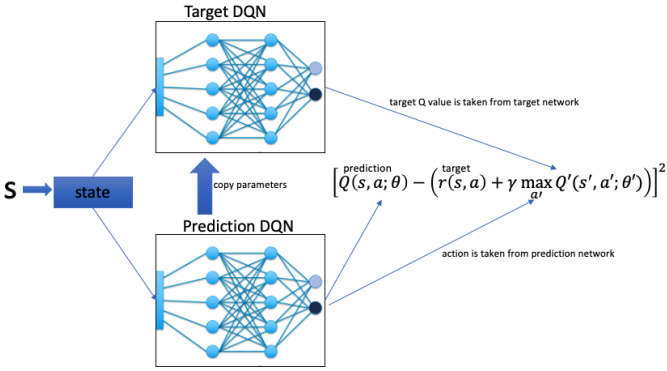


Fig. 7: The diagram illustrates the architecture of a Deep Q-Network (DQN), showing the interaction between two neural networks: the prediction network and the target network. The input state S is fed into both networks. The prediction network is responsible for estimating the Q-value $Q(S, a;\ \theta)$ for the current state and action. The target network, which has its parameters periodically updated from the prediction network, provides a stable target Q-value for the next state $Q(S', a';\ \theta')$. The learning process involves minimizing the loss, which is the squared difference between the prediction network's Q-value and the target Q-value (adjusted by the reward $r(S, a)$ and the discounted maximum Q-value of the next state from the target network). The action to be taken is determined by the prediction network, ensuring a continuous learning and updating cycle for the DQN.

- **Boltzmann Exploration (Soft-max Exploration).** Boltzmann Exploration chooses the next action based on a soft-max distribution over the Q-values. The temperature of this distribution is decreased during the training process, enforcing a more and more deterministic policy. This mitigrates the disadvantage of ε-greedy exploration which is, that it treats all non-greedy actions equally when choosing randomly and thus disregarding better knowledge about the current value function. See figure 8.

3.3 Policy Methods

Policy methods are a class of reinforcement learning algorithms that directly learn a policy, mapping states to actions. Two methods will be presented: Cross-Entropy Method and REINFORCE.

3.3.1 Cross-Entropy Method (CEM)

The cross-entropy method (CEM) is a Monte-Carlo, on-policy, model-free policy search method which iteratively refines the probability distribution over the space of possible policies. Policies are sampled and evaluated and the best ones are retained, improving the policy iteratively.

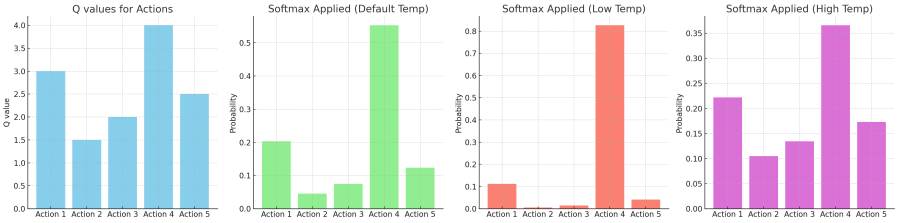


Fig. 8: Example of distributions for 5 actions for use in Boltzmann Exploration. The left diagram shows an example output of a Q-network which gives a value for every possible discrete action. The diagram with the green bars shows a soft-max function as applied to the Q-value from the first figure. This generates a probability distribution over the actions. The two diagrams on the right show this action probability distribution with a low and with a high temperature. A high temperature increases exploration, while a low temperature lets the agent behave more deterministically.

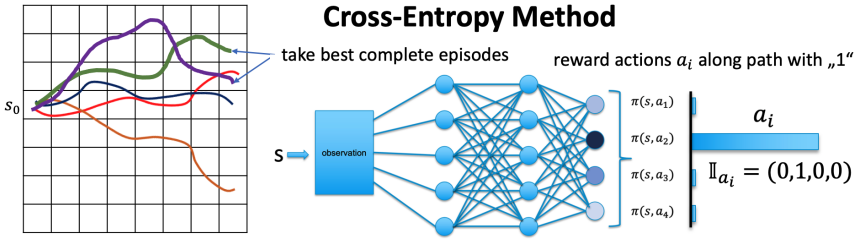


Fig. 9: This image depicts a policy network architecture utilized in the cross-entropy method CEM. The network takes an observation (state S) as input and outputs probabilities for each possible action. These probabilities $\pi(S, a_i)$ define the policy by indicating the likelihood of taking each action a_i given the current state. State-action pairs from top-performing trajectories are shown to the ANN. The respective actions a_i are then one-hot encoded and a multi-class cross-entropy loss function as in supervised learning is applied.

The algorithm is as follows:

1. **Initialize:** Randomly initialize the policy parameters.
2. **Generate Samples:** Collect trajectories using the current policy.
3. **Update Parameters:** Update the policy parameters using the elite trajectories (e.g. 30% top-performing samples).
4. **Repeat:** Iterate through steps 2 and 3 until convergence.

The ANN topology used for CEM is shown in figure 9 where also the training method is explained in more detail.

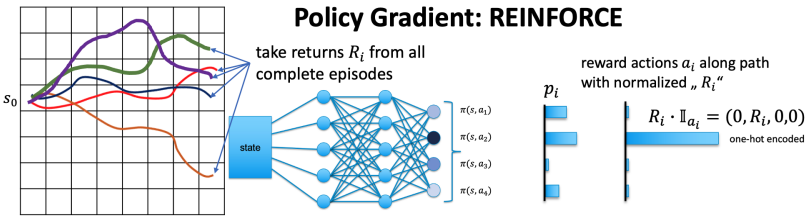


Fig. 10: REINFORCE: A policy gradient method which uses Monte-Carlo methods.

3.3.2 Monte-Carlo Policy Gradient: REINFORCE

REINFORCE (REward Increment = Nonnegative Factor × Offset Reinforcement × Characteristic Eligibility) is a Monte-Carlo, on-policy, model-free policy gradient method that directly maximizes the expected cumulative reward. Adding a baseline term helps reduce the variance of the gradient estimates.

Figure 10 illustrates the working of Monte-Carlo policy gradient methods. Complete episodes are sampled, returns are received, and the according actions are rewarded with the respective rewards.

Objective Function The algorithm maximizes the expected cumulative reward by adjusting the parameters of the policy in the direction of greater reward. The *objective function* for REINFORCE, denoted $J(\theta)$, where θ represents the policy parameters (weights of the ANN), is typically the expected return:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t) \right] \tag{25}$$

If the starting state s_0 is always the same, then $J(\theta) = V(s)$ (state-dependent objective), i.e. the objective function is the expected value in the starting state. If there is a probability distribution P_0 for the starting state, then $J(\theta) = \sum_s P_0(s) \cdot V(s)$ (state-agnostic objective).

REINFORCE uses gradient ascent on $J(\theta)$. The gradient of the objective function with respect to the policy parameters θ is estimated using complete (Monte-Carlo) episodes generated by the policy π_θ (on-policy). The update rule for the policy parameters involves taking steps proportional to the product of the return and the gradient of the log-probability of the trajectory under the policy:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} G_t \cdot \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{26}$$

where G_t is the return from time step t and α is the learning rate. The Algorithm is as follows:

1. **Initialize:** Randomly initialize the policy parameters.

2. **Collect Trajectories:** Execute the policy to collect state-action-reward trajectories.
3. **Compute Returns:** Calculate the (accumulated discounted) returns G_t for each time step.
4. **Update Policy:** Update the policy parameters according to (26).
5. **Repeat:** Iterate through steps 2-4 until convergence.

The policy gradient theorem shows how the update formula (26) is derived.
Policy Gradient Theorem: Given a policy π_θ parameterized by θ, the gradient of the expected return $J(\theta)$ with respect to the policy parameters is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t\right]$$

where G_t is the return following time t, $\pi_\theta(a_t|s_t)$ is the probability of taking action a_t in state s_t under policy π, and τ represents a trajectory of states and actions.
Proof: It has to be shown that

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t)\right] \stackrel{!}{=} \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t\right] \tag{27}$$

This means that the gradient of the expected return (with respect to the weights θ of the policy) is the expected value of the log-probabilities of the taken actions and the cumulative reward. In other words: The gradient of the objective function can be expressed as the gradient of local values of the policy $\nabla_\theta \log \pi_\theta(a_t|s_t)$, which in practice can be calculated with ML-optimizers like tensorflow/Keras, and the total return G_t which is determined empirically. The proof is composed of two steps.
Step 1 According to (12) the objective function $J(\theta)$ can be written as the expected total return:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi}[G(\tau)] = \sum_\tau P(\tau|\theta) \cdot G(\tau)$$

where $P(\tau|\theta)$ is the probability of trajectory τ under policy π according to the *policy induced trajectory distribution* (11), and $G(\tau)$ is the total return for trajectory τ. To find the gradient of $J(\theta)$, it has to be differentiated with respect to θ:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau P(\tau|\theta) \cdot G(\tau) = \sum_\tau \nabla_\theta P(\tau|\theta) \cdot G(\tau)$$

The chain rule for derivatives implies $\ln(f(x))' = f'(x) \cdot \frac{1}{f(x)}$. Applying this to $\nabla_\theta P$ gives $\nabla_\theta P = P \cdot \nabla_\theta \log P$, and thus:

$$\nabla_\theta J(\theta) = \sum_\tau P(\tau|\theta) \cdot \nabla_\theta \log P(\tau|\theta) \cdot G(\tau)$$

Using the definition of the expected value this gives

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[G(\tau)\cdot\nabla_\theta \log P(\tau|\theta)\right] \tag{28}$$

Step 2: Now the policy induced trajectory distribution (11) is substituted for $P(\tau|\theta)$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[G(\tau)\cdot\nabla_\theta \log \prod_{t=0}^{T-1}\pi(a_t|s_t)\cdot P(s_{t+1}|s_t,a_t)\right]$$

Note that $G(\tau)$ is independent of θ and can therefore be written in front of the gradient. Now the product rule for logarithm $\ln(a\cdot b) = \ln a + \ln b$ is applied to the product to receive

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[G(\tau)\cdot\sum_{t=0}^{T-1}\left(\nabla_\theta \log\pi(a_t|s_t) + \nabla_\theta \log P(s_{t+1}|s_t,a_t)\right)\right]$$

Since the transition probabilities $P(s_{t+1}|s_t,a_t)$ are generated by the system and are not dependent on θ, the gradient is zero. Therefore

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} G_t\cdot\nabla_\theta \log\pi_\theta(a_t|s_t)\right] \tag{29}$$

□

The policy gradient theorem shows, that the gradient of the policy depends on the log-gradient of the local policy values.

Implementing REINFORCE with Cross-Entropy Loss

The multi-class cross-entropy loss in a typical classification problem for classes c is given by:

$$L(\theta) = -\sum_c \mathbb{I}_{\bar{c}}(c)\cdot\log(p_c) = -\log(p_{\bar{c}}) \tag{30}$$

where $\mathbb{I}_{\bar{c}}(.)$ is the one-hot encoded vector for the true class $\bar{c}$, p_c is the predicted probability for class c. In order to be able to use ML-optimizers for REINFORCE, the *label* distribution $\mathbb{Y}$ has to be a probability distribution. This is achieved by creating a label probability distribution $\mathbb{Y}(.)$ for the respective state s_t by *pulling* the predicted probability distribution $\pi_\theta(.|s_t)$ with value G_t towards the action $\bar{a}$ which was taken in the episode

$$\mathbb{Y}_{s_t}(.) = (1-G_t)\cdot\pi_\theta(.|s_t) + G_t\cdot\mathbb{I}_{\bar{a}}(.) \tag{31}$$

This requires that the returns G_t for all states from the sampled episodes have been normalized to $G \leftarrow \frac{G-\mu}{\sigma}$. If the normalized $G_t = 0$, then $\mathbb{Y}_{s_t}(.) = \pi_\theta(.|s_t)$

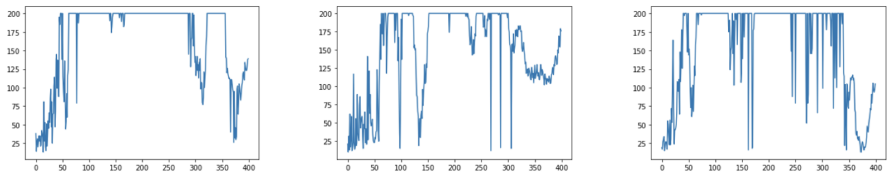


Fig. 11: Learning curves from training episodes using identical hyperparameters. The differences are due to stochasticity in the system and the policy. It can be seen, that the agent keeps forgetting its progress.

and no training occurs. If the normalized $G_t > 0$, then the respective action is rewarded, if $G_t < 0$ then the respective action is penalized.

It can be shown that applying multi-class cross-entropy loss from (30) when using the label probability distribution $\mathbb{Y}$ from (31) leads to the gradient as calculated in equation (29), when calculated for the output neurons of the policy network before applying the soft-max activation function.

3.4 Actor-Critic Methods

Due to high variance of Monte-Carlo methods, REINFORCE generally doesn't have very good convergence properties. Figure 11 shows typical results from training sessions with identical hyperparameters for the cart pole problem in OpenAI gym.

Actor-Critic methods combine policy-based (Actor) and value-based (Critic) reinforcement learning approaches to optimize a policy. This introduces the concept of learning in temporal differences (bootstrapping) for policy gradient methods. As a consequence, variance will be reduced. This section will cover Advantage Critic, and Proximal Policy Optimization (PPO) as examples for Actor-Critic algorithms.

The following two equations illustrate the differences between an Actor-Critic (AC) and a Monte-Carlo Policy Gradient (MC-PG) method. See the policy gradient theorem and equation (28):

$$\begin{aligned} \text{Actor-Critic: } \nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}\left[A^\pi \cdot \nabla_\theta \log P(\tau|\theta)\right] \\ \text{MC-PG: } \nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}\left[G(\tau) \cdot \nabla_\theta \log P(\tau|\theta)\right] \end{aligned}$$

It can be seen that the difference lies in the factor in front of the gradient-logarithm. While MC-PG methods use the total return $G(\tau)$ which has a high variance, AC methods use a local *advantage* A^π as explained in the following section:

3.4.1 Advantage Actor-Critic A2C

In the Advantage Actor-Critic (A2C) algorithm, the Critic estimates the advantage function $(A(s, a))$, representing the advantage of taking action a in

state s over the baseline value. The definition of the advantage is

$$A^{\pi}(s_t, a_t) := Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{32}$$

The advantage A^{π} gives the relative quality of each action.

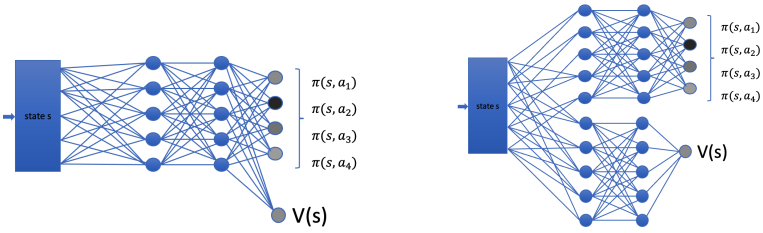
Equation (33) shows the gradient of the actor-critic method, based on the local action probabilities provided by the policy π_θ as derived in Step 2 of the proof of the policy gradient theorem.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A^{\pi_\theta}(s_t, a_t)\right] \tag{33}$$

The algorithm has the same basic form as the Monte-Carlo Policy Gradient method. However, step 3 is different in that the advantage is calculated.

Algorithm:

1. **Initialize:** Randomly initialize the policy (Actor) and the advantage function (Critic) parameters.
2. **Collect Data:** Interact with the environment to collect state-action-reward tuples.
3. **Update Advantage Function:** Minimize the advantage function's error to update the Critic.
4. **Update Policy:** Maximize the advantage values to update the policy.
5. **Repeat:** Iterate through steps 2-4 until convergence.



(a) Actor-critic with shared network. (b) Actor-critic with separate networks.

Fig. 12: Different architectures for Actor-Critic Networks. 12a: The shared network architecture has less parameters and may therefore learn faster. However, learning may be unstable, because the policy and the value function have different gradients. 12b: The separate network architecture learns slower because V depends on the policy π which therefore needs to be learned first. In both cases, the loss function of one part of the ANN (e.g. the part for the value approximation) can be scaled with a parameter to balance the two networks.

As can be seen in figure 12a and 12b, the state value function V is approximated by the neural network. In order to obtain A^{π}, the Q-value needs to be

calculated. This is done using the Bellman Principle (18) such that

$$Q(s,a) \leftarrow r + \gamma \cdot V(s')$$

where r is the immediate reward. s' is obtained from the policy network when applying action a: $s' \sim \pi(a|s)$. Substituting into the definition of the advantage gives

$$A(s,a) = r + \gamma \cdot V(s') - V(s).$$

3.4.2 Generalized Advantage Estimation (GAE)

Generalized Advantage Estimation (GAE) as introduced in [6] provides a way to estimate the advantage function with a balance between bias and variance. The formulation is as follows:

TD Residual The Temporal Difference (TD) residual at each time step is defined as:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Here, r_t is the reward at time t, $V(s)$ is the value function at state s, and γ is the discount factor.

GAE Calculation: GAE is computed using these TD residuals, incorporating a decay parameter λ to adjust the bias-variance trade-off:

$$\hat{A}_t^{GAE} = \sum_{k=0}^{T-t-1} (\gamma\lambda)^k \delta_{t+k}$$

This equation represents a weighted sum of k-step TD residuals. Figure 13c illustrates different weightings for different path lengths.

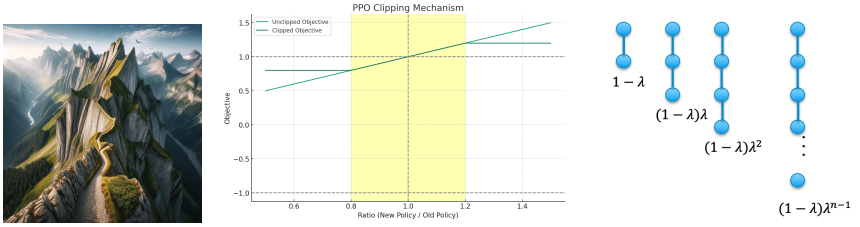
The parameters γ and λ in GAE control the trade-off between bias and variance. A lower λ value results in estimates with less variance but more bias, while a higher λ achieves the opposite. Adjusting λ allows GAE to provide a flexible approach to estimate the advantage function, facilitating more stable and efficient updates in reinforcement learning algorithms.

3.4.3 Proximal Policy Optimization (PPO)

The problem with A2C (Advantage Actor-Critic) and similar on-policy algorithms is that they can be sample inefficient and sensitive to the choice of hyperparameters. They often require a large number of samples to learn effectively because they can only use current policy samples to improve the policy and cannot reuse data from previous policies (due to being on-policy). Moreover, their performance can drastically change with different step sizes for updating the policy. Figure 13a illustrates the problem of finding an optimal policy.

The trick in PPO (Proximal Policy Optimization) (see [7]) is to address these issues by using a clipped objective function as illustrated in figure 13b, which prevents the policy from changing too much at each update. It allows

for using a larger step size while still maintaining stable and robust learning. PPO maintains the benefits of on-policy learning but with a more stable and reliable learning process.



(a) (b) (c)

Fig. 13: Finding a maximum in a policy requires to follow a narrow path (13a) in policy space. If an update step is taken which is too large, the agent *forgets* or drops off its path. This results in a learning progress as shown in figure 11. 13b: Illustration of the PPO Clipping Mechanism: Balancing Exploration and Exploitation in Reinforcement Learning. 13c: In Generalized Advantage Estimation (GAE) there is a balance between bias and variance.

In PPO, a policy improvement ratio is introduced, denoted by

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

r_t indicates how much the new policy differs from the old policy.

The **Objective Function** makes use of a clip function as shown in 13b to decide weather to update the policy

$$L(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right]$$

where A_t is the advantage function.

The **Algorithm** is similar to A2C, with the exception that the mentioned objective function is used. Exploration is usually done by choosing an action randomly given the current probability distribution of the prediction in the respective state.

The objective function of PPO prevents the policy from changing too much in a single update. The clipping mechanism limits the size of the policy update. This allows PPO to reuse data from multiple epochs which makes PPO more sample efficient, more stable and less sensitive to hyperparameters than A2C.

3.5 Off-policy Deep RL and Continuous Control

In this chapter, off-policy reinforcement learning methods that are effective for continuous control tasks are discussed. There will be a particular focus on SAC

(Soft Actor-Critic) and DDPG (Deep Deterministic Policy Gradient). These methods are distinguished by their ability to learn from experiences generated by a policy different from the one being improved (off-policy). They utilize a replay buffer, which stores past interactions with the environment and allows the algorithms to learn from this stored data multiple times.

For discrete action spaces, a neural network might use separate output neurons for each possible action, a technique commonly seen in methods like DQN. However, when dealing with continuous action spaces, SAC and DDPG use a different approach. They produce real-valued outputs that correspond to the specifics of the action space. An example for a continuous control is the steering angle of an autonomous vehicle.

DDPG is known for creating a deterministic policy, directly linking states to specific actions. This is in contrast to SAC, which creates a stochastic policy by outputting parameters that define a probability distribution, such as a Gaussian. This stochastic approach helps maintain a balance between trying out new actions (exploration) and using actions known to yield good results (exploitation).

It should be noted that PPO, as outlined in section 3.4.3, can be adapted for continuous stochastic control tasks. This adaptation involves implementing a policy network that outputs parameters—specifically, a mean μ and standard deviation σ—for the action distribution.

Both SAC and DDPG are made to work with continuous action spaces and make good use of the off-policy learning approach.

3.5.1 Soft Actor-Critic SAC

SAC incorporates elements from DQN, like the use of two separate networks for Q-value estimation, enhancing stability in value function approximation. This dual-network approach, combined with direct policy learning, positions SAC as an off-policy method, enabling learning from experiences generated by various policies. Unlike DQN, SAC's architecture is tailored for continuous action spaces.

Soft Actor-Critic, as introduced in [8], integrates entropy regularization into its objective function. This encourages the policy to maintain a level of exploration by preferring higher entropy policies, preventing premature convergence to overly deterministic behavior. The difference to the method described in figure 8 is that Boltzmann exploration directly uses the value estimates to form a probability distribution for action selection while entropy regularization integrates exploration into the learning objective itself.

While standard RL methods maximize the expected sum of rewards $\sum \mathbb{E}_\pi[r(s_t, a_t)]$ (dropping the discount factor γ for clarity), in Soft Actor-Critic the expected entropy of the policy is included in the objective function, given by:

$$J(\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{T} (r(s_t, a_t) + \alpha \cdot \mathcal{H}(\pi(\cdot|s_t))) \right] \tag{34}$$

Where $\mathcal{H}(\pi(\cdot|s_t))$ is the entropy of the policy and α is a temperature parameter that balances the importance of the reward and entropy terms.

The entropy term is defined as

$$\mathcal{H}(\pi(\cdot|s_t)) := -\sum_{a \in A} \pi(a|s_t) \log(\pi(a|s_t)). \tag{35}$$

The policy network (actor) of SAC outputs a probability distribution for actions, typically a Gaussian distribution with parameters like mean and standard deviation. Actions are sampled from this distribution.

$$a \sim \pi(.|s) = \mathcal{N}(\mu(s), \sigma(s)) \tag{36}$$

where $\mathcal{N}$ denotes the Gaussian distribution with mean μ and variance σ.

The particular design of the policy network enables SAC to be used for continuous action spaces.

Soft Actor-Critic uses several neural networks to approximate

- the soft state value function $V_\psi(s)$
- the soft Q-function $Q_\theta(s, a)$
- the policy $\pi_\phi(a|s)$

where ψ, θ, ϕ denote the network parameters.

The soft state value function is defined as

$$\begin{aligned} V(s) &:= \mathbb{E}_{a \sim \pi}[Q(s,a) - \alpha \cdot \log \pi(a|s)] \text{ definition of soft } V \\ &= \sum \pi(a|s) \cdot [Q(s,a) - \alpha \cdot \log \pi(a|s)] \text{ definition expected value} \\ &= \sum \pi(a|s) \cdot Q(s,a) + \alpha \cdot \mathcal{H}(\pi(\cdot|s_t)) \text{ definition of entropy } \mathcal{H} \end{aligned}$$

This term encourages to select high-reward actions (as indicated by the Q-value) with the tendency to maintain diversity in action selection (as encouraged by the entropy in $-\mathbb{E} \log \pi$). The inclusion of the negative log probability of the chosen action encourages the policy to favor actions that are not only high in expected return but also high in entropy, promoting exploration.

The soft Q-function

$$Q_\theta(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_\pi[V_\psi(s_{t+1})] \tag{37}$$

is updated using the soft state value function V_ψ. Then the policy is updated using the soft Q-function. The objective function for the policy is given by

$$J_\pi(\phi) = \mathbb{E}\left[\alpha \cdot \log(\pi_\theta(a_t|s_t)) - Q_\phi(s_t, a_t)\right] \tag{38}$$

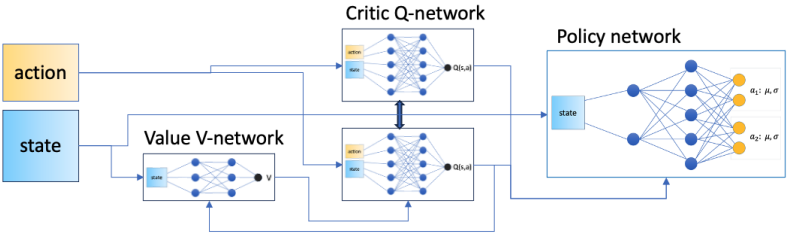


Fig. 14: This illustration shows the neural network architecture of the Soft Actor-Critic (SAC) algorithm. The architecture comprises three networks: the network for the state value function; the Q-network (Critic), consisting of two separate networks that estimate the Q-value function Q(s,a), and the Policy network, which outputs a probability distribution over actions, parameterized by mean μ and standard deviation σ, given the state. This structure enables SAC to balance exploration and exploitation in continuous action spaces.

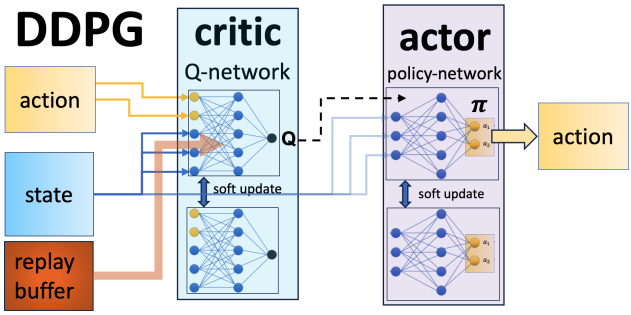


Fig. 15: Neural Network architecture involved in Deep Deterministic Policy Gradient. The state, the previously taken action, and data from a replay buffer are taken as input to a Q-network. The Q-value is taken to update the policy network. Both neural networks are copied and use soft target updates for stability. The ouput of the policy network is a continuous deterministic action.

also favouring a probability distribution with high entropy.

Figure 14 illustrates the neural network structure of Soft Actor-Critic.

3.5.2 Deep Deterministic Policy Gradients (DDPG)

Deep Deterministic Policy Gradients (DDPG) as described in [9] is an actor-critic algorithm designed for continuous action spaces. It combines ideas from both value-based methods and policy-based methods, leveraging a deterministic policy and a Q-function. It is particularly useful in robotic applications.

Figure 15 illustrates the fundamental architecture of DDPG, an actor-critic method. The policy network (actor) deterministically outputs continuous

action values for each state. The policy network is trained using a policy gradient approach, where the training involves a Q-value provided by a separate network, known as the critic. This Q-network concept shares similarities with DQN, as detailed in section 3.2. In DDPG, there are two sets of networks: the actor-critic networks and their corresponding target networks, which are softly updated to enhance training stability. The critic network takes the current state and action as inputs, along with data sampled from the replay buffer, and outputs a Q-value through a single linear neuron. This Q-value represents the estimated value of taking the given action in the current state.

3.6 Conclusion and Summary

The article provided an introduction to reinforcement learning, describing some of the most recent model-free methods like PPO, SAC, and DDPG. These algorithms have spawned various versions, enhancements, and practical implementations, such as the ML-agents framework in the Unity game engine.

Current research in reinforcement learning is expanding towards broader applications, though real-world problems pose significant challenges. One major difficulty is the slow acquisition of real-world data and the constraints on exploration in real-world settings. Off-line reinforcement learning addresses learning effective policies from fixed datasets without additional online interaction with the environment. This approach is particularly relevant for scenarios where real-time exploration is either impractical or risky.

Another area of interest are large and complex problems characterized by sparse rewards, where feedback is minimal and infrequent. Potential solutions may involve integrating generative AI, Large Language Models (LLMs), or transformers to better handle these challenges. Additionally, there is ongoing work on developing foundational models for robotics, aimed at equipping algorithms with the necessary tools to address complex, multifaceted problems.

As reinforcement learning continues to evolve, its application to real-world scenarios becomes increasingly feasible, opening up new avenues for innovation and problem-solving.

References

[1] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT press, Bradford Books; 2. edition, ISBN-13: 978-0262039246 (2018)

[2] Lapan, M.: Deep Reinforcement Learning Hands-On. Packt Publishing, Birmingham, UK (2018)

[3] Palmas, A., Ghelfi, E., Petre, A.G., Kulkarni, M., Anand, N., Nguyen, Q., Sen, A., So, A., Basak, S.: The The Reinforcement Learning Workshop: Learn How to Apply Cutting-edge Reinforcement Learning Algorithms to a Wide Range of Control Problems. Packt Publishing Ltd, Birmingham (2020)

[4] Graesser, L., Keng, W.L.: Foundations of Deep Reinforcement Learning. Addison-Wesley Professional, ISBN13: 9780135172384 (2019)

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)

[6] Schulman, J., Moritz, P., Levine, S., Jordan, M., Abbeel, P.: High-dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438 (2015)

[7] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)

[8] Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: International Conference on Machine Learning, pp. 1861–1870 (2018). PMLR

[9] Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. International Conference on Learning Representations, Puerto Rico 2016 (2016)