3.4 Model-free reinforcement learning

The methods described before require knowledge of the system dynamics. However, this is not the case in model-free reinforcement learning, where the dynamics are unknown. As a result, it becomes necessary to develop solution methods that do not rely on this explicit knowledge.

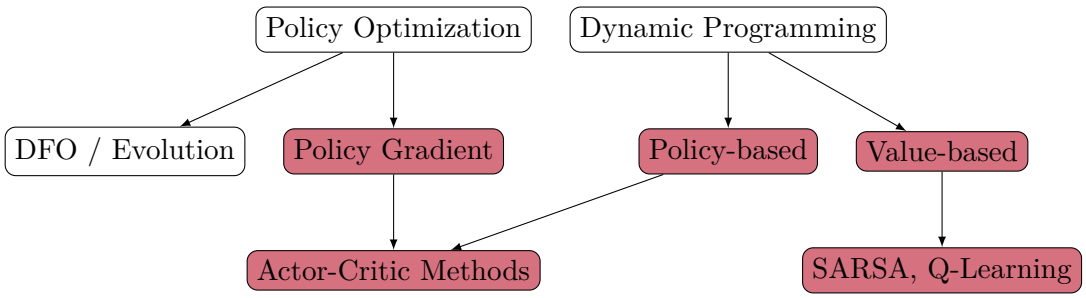


Figure 3.8: Classification of model-free RL algorithms [3.12, p. 4].

In reinforcement learning (RL), there are numerous options for approximating elements like policies, value functions, and dynamic models. Figure 3.8 provides a *classification of model-free RL algorithms*. There are two main approaches: (i.) *Policy Optimization* and (ii.) *(Approximate) Dynamic Programming*

Policy Optimization techniques focus on optimizing the policy, the function mapping the agent's state to its next control input. They consider reinforcement learning as a numerical optimization problem where we optimize the expected reward with respect to the policy's parameters. There are two ways to optimize a policy: *Derivative-free Optimization* (DFO) algorithms and policy gradient methods. DFO algorithms work by perturbing the policy parameters, assessing the performance, and then moving towards better performance. They're simple to implement but scale poorly with increased parameters. On the other hand, *Policy Gradient Methods* estimate the policy improvement direction using various quantities measured by the agent, thus avoiding parameter perturbation. They are more complex to implement but can optimize larger policies than DFO algorithms.

Approximate Dynamic Programming (ADP), is based on learning value functions, predicting the agent's potential reward. ADP algorithms strive to satisfy certain consistency equations that the true value functions obey. Known algorithms for exact solutions in finite state-input RL problems are *policy-based* and *value-based*. They can be combined with function approximation in multiple ways; currently, leading descendents of value iteration focus on *approximating Q-functions*.

Lastly, *Actor-Critic* methods incorporate elements from both policy optimization and dynamic programming. They optimize a policy using value functions for faster optimization and often utilize ideas from approximate dynamic programming for fitting the value functions.

We discuss five key model-free reinforcement learning algorithms. Table 3.5 summarizes the use of these model-free reinforcement learning algorithms. It categorizes these algorithms based on their applicability in discrete and continuous input spaces. On-policy and

off-policy learning refer to how an algorithm learns a policy. On-policy learning improves the policy used to make decisions, while off-policy learning improves a different policy from the one used to generate data.

Algorithm	Policy	Discrete	Continuous	Tabular
Monte Carlo	on	Yes	No	Yes
SARSA	on	Yes	No	Yes
Q-Learning	off	Yes	No	Yes
Deep Q-Network	off	Yes	Yes	No
Policy Gradient	on/off	Yes	Yes	No

Table 3.3: Use of reinforcement learning algorithms in different input spaces.

In the following section, we will specifically focus on what are known as *tabular solution methods*, see [3.7, p. 23], which are applicable in scenarios with an infinite horizon as well as discrete input and state spaces.

3.4.1 Generalized Policy Iteration

In model-free RL, the use of the action-value function $Q^\pi(\mathbf{x}, \mathbf{u})$ is favored over the state-value function $V^\pi(\mathbf{x})$ because $Q^\pi(\mathbf{x}, \mathbf{u})$ eliminates the need to know the state-transition and reward function explicitly, which is consistent with the goal of model-free RL to learn optimal policies without a model of the environment, cf. (3.74). Let us introduce the action-value table/matrix

$$\mathbf{Q}^\pi = \begin{bmatrix} Q^\pi(\mathbf{x}_0, \mathbf{u}_0) & Q^\pi(\mathbf{x}_0, \mathbf{u}_1) & \dots & Q^\pi\left(\mathbf{x}_0, \mathbf{u}_{|\mathcal{U}|-1}\right) \\ Q^\pi(\mathbf{x}_1, \mathbf{u}_0) & Q^\pi(\mathbf{x}_1, \mathbf{u}_1) & \dots & Q^\pi\left(\mathbf{x}_1, \mathbf{u}_{|\mathcal{U}|-1}\right) \\ \vdots & \ddots & \vdots & \\ Q^\pi\left(\mathbf{x}_{|\mathcal{X}|-1}, \mathbf{u}_0\right) & Q^\pi\left(\mathbf{x}_{|\mathcal{X}|-1}, \mathbf{u}_1\right) & \dots & Q^\pi\left(\mathbf{x}_{|\mathcal{X}|-1}, \mathbf{u}_{|\mathcal{U}|-1}\right) \end{bmatrix}. \tag{3.136}$$

The *Generalized Policy Iteration* (GPI) algorithm refers to an iterative procedure to improve the policy when combining policy evaluation and improvement, cf. Figure 3.9:

$$\pi^{(0)} \xrightarrow{\text{evaluate}} (\mathbf{Q}^\pi)^{(0)} \xrightarrow{\text{improve}} \pi^{(1)} \xrightarrow{\text{evaluate}} (\mathbf{Q}^\pi)^{(1)} \dots \xrightarrow{\text{improve}} \pi^* \xrightarrow{\text{evaluate}} (\mathbf{Q}^\pi)^{(*)} \tag{3.137}$$

In GPI, the action-value function $\mathbf{Q}^\pi$ is approximated repeatedly to be closer to the true value $(\mathbf{Q}^\pi)^{(*)}$ of the current policy and in the meantime, the policy π is improved repeatedly to approach optimality, i.e., π^*. This policy iteration process works and always converges to the optimality, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI [3.7, p. 86].

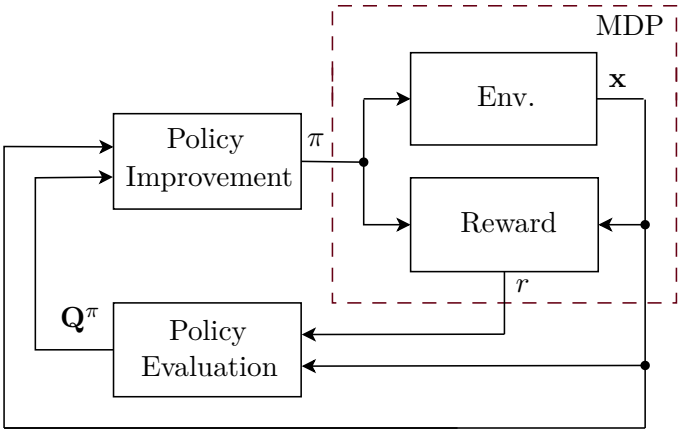


Figure 3.9: Generalized Policy Iteration.

Exploration vs. Exploitation

Efficient decision-making is often challenged by the need to balance exploration and exploitation. Exploration involves seeking new information to improve future decisions, while exploitation leverages current knowledge to maximize immediate rewards.

- Greedy input: When an agent chooses an input with the highest estimated value at the moment. The agent exploits its current knowledge by selecting the greedy input.
- Non-greedy input: When the agent does not choose the action with the highest estimated value and forgoes the immediate reward, hoping to gather more information about other inputs.
- Exploration: This enables the agent to improve its knowledge of each input, which hopefully leads to long-term benefits.
- Exploitation: The agent can choose the greedy input to obtain the highest reward for a short-term advantage. However, a purely greedy input selection can result in suboptimal behavior.

This creates a dilemma between exploration and exploitation, as an agent cannot explore and exploit simultaneously. Striking a balance between *exploration* and *exploitation* is crucial in model-free RL.

Policy Improvement

Policy improvement refers to acting greedily and exploiting current knowledge. In view of (3.74), given an action-value function $Q^\pi(\mathbf{x}, \mathbf{u})$, the greedy policy

$$\pi(\mathbf{u} \mid \mathbf{x}) \leftarrow \begin{cases} 1 & \text{if } \mathbf{u} = \arg\max_{\mathbf{w} \in \mathcal{U}} Q^\pi(\mathbf{x}, \mathbf{w}) \\ 0 & \text{else} \end{cases}, \tag{3.138}$$

is selected to maximize the expected reward based on current knowledge. Several strategies exist to balance exploration and exploitation:

ϵ-Greedy Method
With ϵ-greedy, at each step in state $\mathbf{x}$, the agent selects a random input with a fixed probability $\epsilon \in [0, 1]$:

- With probability $\frac{\epsilon}{|\mathcal{U}|} + 1 - \epsilon$, choose $\mathbf{u} = \arg\max_{\mathbf{w} \in \mathcal{U}} Q^\pi(\mathbf{x}, \mathbf{w})$ and
- with probability $\frac{\epsilon}{|\mathcal{U}|}$, select any input uniformly at random.

Hence, the ϵ-greedy policy is defined as

$$\pi(\mathbf{u} \mid \mathbf{x}) \leftarrow \begin{cases} \frac{\epsilon}{|\mathcal{U}|} + 1 - \epsilon & \text{if } \mathbf{u} = \arg\max_{\mathbf{w} \in \mathcal{U}} Q^\pi(\mathbf{x}, \mathbf{w}) \\ \frac{\epsilon}{|\mathcal{U}|} & \text{else} \end{cases}. \tag{3.139}$$

Compared to the greedy method (3.74), the ϵ-greedy method helps in exploring the input space. Pseudo code for the ϵ-greedy Algorithm is given in Algorithm 5.

Algorithm 5: ϵ-Greedy Policy Selection

Input: Current state $\mathbf{x}_k$, Q-function $\hat{Q}^\pi(\mathbf{x}, \mathbf{u})$, exploration parameter $\epsilon \in [0, 1]$, input space $\mathcal{U}$
Output: Selected input $\mathbf{u}_k$

```
/* Generate random number for exploration decision */
```
$p \leftarrow \text{uniform}(0, 1)$;
if $p < \epsilon$ **then**
```
    /* Exploration: select random input */
```
    $\mathbf{u}_k \leftarrow$ random sample from $\mathcal{U}$;
else
```
    /* Exploitation: select greedy input */
```
    $\mathbf{u}_k \leftarrow \arg\max_{\mathbf{w} \in \mathcal{U}} \hat{Q}^\pi(\mathbf{x}_k, \mathbf{w})$;
end
return $\mathbf{u}_k$;

Softmax Strategy
A downside of ϵ-greedy exploration is that it randomly picks any possible input for exploration. This means it's just as likely to pick the worst option as it is to pick the almost best one. In contrast, the softmax[7] strategy utilizes input-selection probabilities which are determined by ranking the action-value function estimates using a Boltzmann distribution. The softmax policy is defined as

$$\pi(\mathbf{u} \mid \mathbf{x}) \leftarrow \operatorname{soft}\max_{\mathbf{u}\in\mathcal{U}} \beta Q^{\pi}(\mathbf{x}, \mathbf{u}) = \frac{\exp\left(\frac{1}{\beta} Q^{\pi}(\mathbf{x}, \mathbf{u})\right)}{\sum_{\mathbf{w}\in\mathcal{U}} \exp\left(\frac{1}{\beta} Q^{\pi}(\mathbf{x}, \mathbf{w})\right)} \,. \tag{3.140}$$

Here, β is the temperature parameter that controls the stochasticity of the policy. A low β makes the policy more deterministic, choosing the action with the highest Q-value with higher probability. A high β makes the policy more exploratory, distributing the selection probability more uniformly across inputs. In practice, the agent uses the Softmax Strategy to randomly select the next input by sampling from the distribution.

> *Example* 3.10 (Comparison: ϵ-Greedy vs. Softmax Exploration for the Forzen Lake Example). For the same Q-values $Q^{\pi}(x, u(\leftarrow)) = 0.5$, $Q^{\pi}(x, u(\downarrow)) = 0.8$, $Q^{\pi}(x, u(\rightarrow)) = 0.3$, and $Q^{\pi}(x, u(\uparrow)) = 0.6$, we compare ϵ-Greedy vs. Softmax Exploration. Figure 3.10 illustrates the key difference: ϵ-greedy treats all non-optimal inputs equally during exploration, while softmax assigns higher probabilities to inputs with higher Q-values.

Policy Evaluation

If the dynamics of the environment are unknown, several methods can be used for *policy evaluation*. In policy evaluation, we apply the update rule

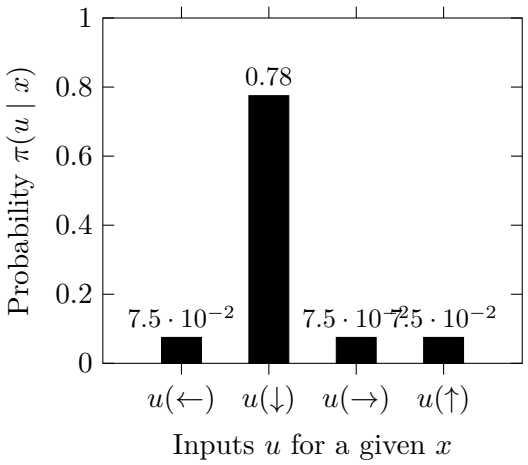
$$\hat{Q}^{\pi}(\mathbf{x}, \mathbf{u}) \leftarrow \hat{Q}^{\pi}(\mathbf{x}, \mathbf{u}) + \alpha\delta \tag{3.141}$$

to improve the estimated action-value function $\hat{Q}^{\pi}(\mathbf{x}, \mathbf{u})$. Here, $\alpha \in (0, 1]$ is the learning rate and δ denotes a residual. From a control engineering perspective, (3.141) is an integrator. It updates $\hat{Q}^{\pi}(\mathbf{x}, \mathbf{u})$ until the residual δ approaches zero.

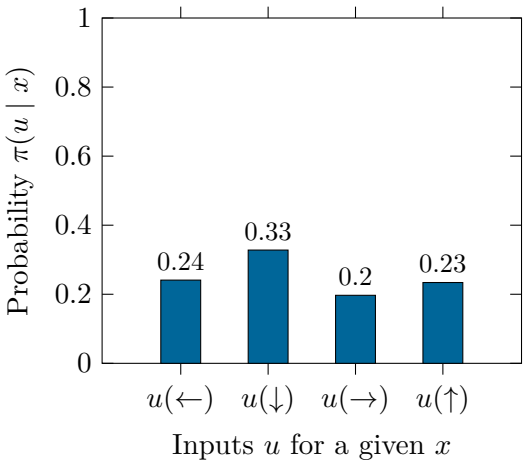
Monte Carlo methods
The principle of Monte Carlo (MC) methods is to sample rollouts $\boldsymbol{\tau}$ using the current policy π and approximate the expectation $Q^{\pi}(\mathbf{x}, \mathbf{u}) = \mathbb{E}_{\pi}\{\mathsf{R}_k \mid \mathbf{x}_k = \mathbf{x}, \mathbf{u}_k = \mathbf{u}\}$ to compute the empirical mean return[8]

$$\hat{R}_k = \sum_{j=k}^{N} \gamma^{j-k} r_j \tag{3.142}$$

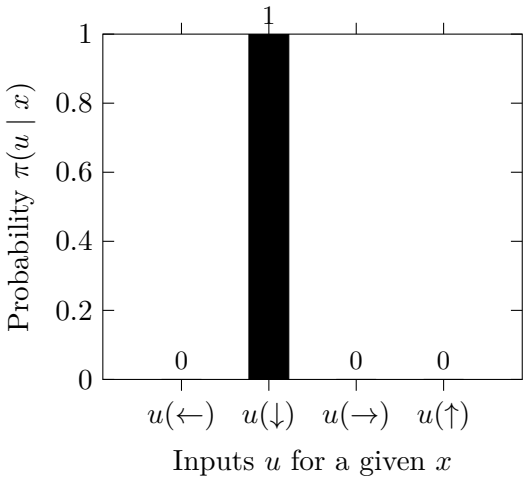
[7] Actually, the softmax function is a softened version of the argmax function. Therefore, it could be more appropriately termed softargmax.
[8] Also referred to as Monte Carlo return.



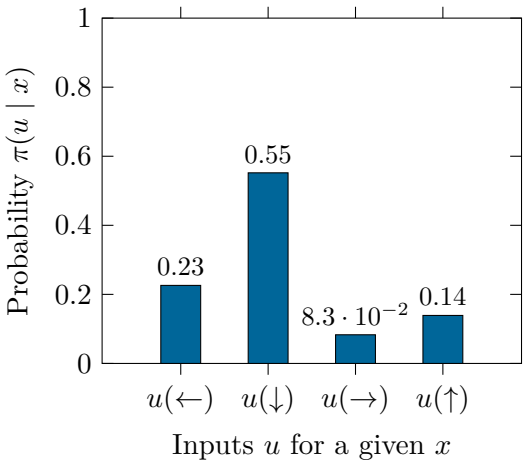
(a) ϵ-Greedy: $\epsilon = 0.3$ (exploration).



(b) Softmax: $\beta = 1.0$ (balanced).



(c) ϵ-Greedy: $\epsilon = 0.0$ (pure exploitation).



(d) Softmax: $\beta = 0.5$ (exploitation-focused).

Figure 3.10: Comparison of exploration strategies for the Frozen Lake game. Q-values: $Q^\pi(x, u(\leftarrow)) = 0.5$, $Q^\pi(x, u(\downarrow)) = 0.8$, $Q^\pi(x, u(\rightarrow)) = 0.3$, $Q^\pi(x, u(\uparrow)) = 0.6$. Key observation: ϵ-greedy distributes exploration probability uniformly among non-greedy actions, while softmax assigns probabilities proportional to Q-values.

for each state-input pair. Then, with residual

$$\delta = \frac{1}{N(\mathbf{x}, \mathbf{u})} \sum_{k=1}^{N(\mathbf{x},\mathbf{u})} \left[\hat{R}_k - \hat{Q}^\pi(\mathbf{x}, \mathbf{u}) \right], \tag{3.143}$$

where $N(\mathbf{x}, \mathbf{u})$ is the total number of times the state-input pair is visited, we get from (3.141) the Monte Carlo update rule

$$\hat{Q}^{\pi}(\mathbf{x},\mathbf{u}) \leftarrow \hat{Q}^{\pi}(\mathbf{x},\mathbf{u}) + \frac{1}{N(\mathbf{x},\mathbf{u})} \sum_{k=1}^{N(\mathbf{x},\mathbf{u})} \left[\hat{R}_k - \hat{Q}^{\pi}(\mathbf{x},\mathbf{u})\right] . \tag{3.144}$$

An update law for the visitation count $N(\mathbf{x},\mathbf{u})$ is mathematically formulated as

$$N(\mathbf{x},\mathbf{u}) \leftarrow N(\mathbf{x},\mathbf{u}) + \mathbb{1}_{\mathbf{x}_k=\mathbf{x},\mathbf{u}_k=\mathbf{u}} .$$

Here, $\mathbb{1}_{\mathbf{x}_k=\mathbf{x},\mathbf{u}_k=\mathbf{u}}$ is the binary indicator function[9]. The equation (3.144) represents a simple average of the returns. After each episode, for each state-input pair visited, we calculate the return $\hat{R}_k$ from that point until the end of the episode and then update the Q-value estimate for that state-action pair accordingly. This particular MC method is called every-visit MC because we compute the return every time a state is visited in the episode. MC methods need to learn from complete episodes to compute and all the episodes must eventually terminate.

SARSA: On-policy Temporal-Difference Learning
On-policy Temporal-Difference (TD) learning combines the principles of Monte Carlo methods with Dynamic Programming (DP). Under the certainty equivalence assumption, we get from the Bellman Expectation Equation, cf. (3.59),

$$Q^{\pi}(\mathbf{x},\mathbf{u}) = \bar{r}(\mathbf{x},\mathbf{u}) + \gamma Q^{\pi}(\mathbf{x}',\mathbf{u}') . \tag{3.145}$$

The key idea in On-policy TD learning is to update the action-value function $\hat{Q}^{\pi}(\mathbf{x},\mathbf{u})$ using the TD residual

$$\delta = \bar{r}(\mathbf{x},\mathbf{u}) + \gamma \hat{Q}^{\pi}(\mathbf{x}',\mathbf{u}') - \hat{Q}^{\pi}(\mathbf{x},\mathbf{u}) \tag{3.146}$$

and to follow the SARSA update rule, see, e.g., [3.7, p. 120],

$$\hat{Q}^{\pi}(\mathbf{x},\mathbf{u}) \leftarrow \hat{Q}^{\pi}(\mathbf{x},\mathbf{u}) + \alpha\left[\bar{r}(\mathbf{x},\mathbf{u}) + \gamma \hat{Q}^{\pi}(\mathbf{x}',\mathbf{u}') - \hat{Q}^{\pi}(\mathbf{x},\mathbf{u})\right] . \tag{3.147}$$

For $\alpha = 0$, the value $\hat{Q}^{\pi}(\mathbf{x},\mathbf{u})$ remains unchanged, while for $\alpha = 1$, the old value $\hat{Q}^{\pi}(\mathbf{x},\mathbf{u})$ is replaced entirely by the so-called TD-target[10] $\bar{r}(\mathbf{x},\mathbf{u}) + \gamma \hat{Q}^{\pi}(\mathbf{x}',\mathbf{u}')$. This becomes evident when we express (3.147) as a first-order low-pass filter in the form

$$\hat{Q}^{\pi}(\mathbf{x},\mathbf{u}) \leftarrow (1-\alpha)\hat{Q}^{\pi}(\mathbf{x},\mathbf{u}) + \alpha\left[\bar{r}(\mathbf{x},\mathbf{u}) + \gamma \hat{Q}^{\pi}(\mathbf{x}',\mathbf{u}')\right] . \tag{3.148}$$

On-policy Temporal-Difference learning is called SARSA because of this data sequence used for calculation – State, Action, Reward, State, Action. The estimate $\hat{Q}^{\pi}$ is updated based on its own estimation, which is a form of bootstrapping, as described in [3.7, p. 89]. TD learning can learn from incomplete rollouts and hence we don't need to track the rollouts up to termination.

[9] If $\mathbf{x}_k = \mathbf{x}$ and $\mathbf{u}_k = \mathbf{u}$, the function $\mathbb{1}_{\mathbf{x}_k=\mathbf{x},\mathbf{u}_k=\mathbf{u}}$ returns 1, otherwise, it returns 0.
[10] In machine learning, a target value refers to the actual value you aim to predict with your model. It's the outcome or label that the algorithm is being trained to forecast in supervised learning.

Q-learning: Off-policy Temporal-Difference Learning
Q-learning is an extension of TD-learning that goes beyond predicting the value function Q^π to enable the determination of an optimal policy π^*. The update rule for Q-learning involves updating the estimate $\hat{Q}^\pi$ at each time step, considering both the observed state $\mathbf{x}$ and the corresponding action $\mathbf{u}$. Note that the Bellman Optimality Equations (3.62) reads as

$$Q^*(\mathbf{x},\mathbf{u}) = \bar{r}_k(\mathbf{x}_k,\mathbf{u}_k) + \gamma \mathbb{E}_{\mathbf{x}_{k+1}\sim \mathsf{p}_\times(\mathbf{x}_{k+1}|\mathbf{x}_k,\mathbf{u}_k)} \left\{ \max_{\mathbf{w}\in\mathcal{U}} Q^*(\mathbf{x}',\mathbf{w}) \right\} . \tag{3.149}$$

Since we do not have access to the optimal values Q^*, the idea in Off-policy TD learning is to update the estimate of the action-value function $\hat{Q}^\pi(\mathbf{x},\mathbf{u})$ using the residual

$$\delta = \bar{r}(\mathbf{x},\mathbf{u}) + \gamma \max_{\mathbf{w}\in\mathcal{U}} \hat{Q}^\pi(\mathbf{x}',\mathbf{w}) - \hat{Q}^\pi(\mathbf{x},\mathbf{u}) . \tag{3.150}$$

The Q-learning update law than becomes

$$\hat{Q}^\pi(\mathbf{x},\mathbf{u}) \leftarrow \hat{Q}^\pi(\mathbf{x},\mathbf{u}) + \alpha \left[\bar{r}(\mathbf{x},\mathbf{u}) + \gamma \max_{\mathbf{w}\in\mathcal{U}} \hat{Q}^\pi(\mathbf{x}',\mathbf{w}) - \hat{Q}^\pi(\mathbf{x},\mathbf{u}) \right] . \tag{3.151}$$

The estimated action-value function $\hat{Q}^\pi$ directly approximates optimal action-value function Q^*, independent of the policy being followed. Pseudo code for the Q-Learning Algorithm is given in Algorithm 6.

Algorithm 6: Q-Learning Algorithm

`/* Initialization */`
Initialize Q-function $\hat{Q}^\pi(\mathbf{x},\mathbf{u})$ arbitrarily for all $\mathbf{x}\in\mathcal{X}$, $\mathbf{u}\in\mathcal{U}$;
Initialize exploration parameter $\epsilon = 1.0$;
for $e = 0,1,2,\ldots,E$ *episodes* **do**
  Initialize state $\mathbf{x}_0$;
  for $k = 0,1,2,\ldots,N-1$ *steps* **do**
    `/* ϵ-greedy action selection */`
    if *random()* $< \epsilon$ **then**
      Select random action $\mathbf{u}_k \in \mathcal{U}$;
    else
      $\mathbf{u}_k = \arg\max_{\mathbf{w}\in\mathcal{U}} \hat{Q}^\pi(\mathbf{x}_k,\mathbf{w})$;
    end
    Execute $\mathbf{u}_k$, observe $\mathsf{r}_k = \mathsf{r}(\mathbf{x}_k,\mathbf{u}_k)$ and $\mathbf{x}_{k+1}$;
    `/* Q-Learning update rule */`
    $\hat{Q}^\pi(\mathbf{x}_k,\mathbf{u}_k) \leftarrow \hat{Q}^\pi(\mathbf{x}_k,\mathbf{u}_k) + \alpha\left[\mathsf{r}_k + \gamma \max_{\mathbf{w}\in\mathcal{U}} \hat{Q}^\pi(\mathbf{x}_{k+1},\mathbf{w}) - \hat{Q}^\pi(\mathbf{x}_k,\mathbf{u}_k)\right]$;
    $\mathbf{x}_k \leftarrow \mathbf{x}_{k+1}$; break if episode terminated;
  end
  $\epsilon \leftarrow \max(\epsilon - \epsilon_{\text{decay}}, \epsilon_{\min})$;
end

Finally, we compare Monte Carlo, SARSA and Q-Learning and draw some conclusions:

Aspect	Monte Carlo	SARSA	Q-Learning
Learning Type	Episodic	On-policy TD	Off-policy TD
Bootstrap	No (uses complete returns)	Yes (uses next state estimate)	Yes (uses next state estimate)
Policy Dependency	Learns from current policy	Learns and improves current policy	Learns optimal policy regardless of behavior policy
Exploration vs Exploitation	Policy-dependent	Conservative (follows current policy)	Aggressive (always seeks optimal)
Convergence	Guaranteed with sufficient exploration	Guaranteed to policy being followed	Guaranteed to optimal policy
Variance/Bias	High variance, unbiased	Lower variance, some bias	Moderate variance, some bias
Sample Efficiency	Low (requires complete episodes)	Moderate (learns during episodes)	High (learns from any experience)
Computational Cost	Low per episode, high memory	Moderate	Moderate
Suitability	Episodic tasks, simple environments	Safe exploration, real-world applications	Complex environments, optimal performance
Robustness	Sensitive to stochastic rewards	Robust to exploration strategy	Robust but may be overly optimistic

Table 3.4: Comparison of Monte Carlo, SARSA, and Q-Learning methods.

Note 3.11. A Jupyther Notebook of the *Monte Carlo Algorithm*, the *SARSA Algorithm* and *Q-Learning Algorithm* for the Frozen Lake Example can be found here. In addition, single file Python scripts for Monte Carlo Algorithm, the SARSA Algorithm, and the Q-Learning Algorithm are made available for the for the Frozen Lake Example.

There are a couple of extensions of TD- and Q-learning that are designed to address its limitations and improve its applicability and performance. Key extensions include:

- Double Q-Learning [3.7, p. 125]: Addresses the overestimation of action-state values by decoupling the improvement and evaluation of the input in the action-state value update.

- Eligibility Traces [3.7, p. 287]: The more theoretical view is that Eligibility Traces are a bridge from TD to Monte Carlo methods (forward view). According to the other view, an Eligibility Trace is a temporary record of the occurrence of an event, such as the visiting of a state or the taking of an action (backward view). The trace marks the memory parameters associated with the event as eligible for undergoing learning changes. TD(λ) [3.7, p. 293]: Extends the basic TD method by considering a series of TD errors for all time steps until the end of the episode, weighted by a factor λ. The factor $\lambda \in [0, 1]$ allows you to balance the trade-off between bias (accuracy) and variance (stability) in the learning updates.
- $Q(\lambda)$ [3.7, p. 312]: This extension applies the concept of Eligibility Traces to Q-Learning. In $Q(\lambda)$, every input-value pair (Q-value) has an associated eligibility trace, which decays over time but gets bumped up when visited. The action-state value updates are then applied to all pairs proportionally to their eligibility, allowing for quicker propagation of information through the state-input space.

3.4.2 Approximate solution methods

For large state and input spaces, methods where the state value function V^π or the action-state value function Q^π are represented by means of a table are no longer manageable. The reason for this is the high memory requirements of the table, since a values must be stored for each state or state-input pair. Moreover, for a continuous state space, an infinite number of values would have to be stored, and the computational cost is considerable has to be applied to each state or state-input pair, respectively. Instead of a table, a function approximator parameterized with the vector $\boldsymbol{\phi}$ can be used to represent the value functions in the form

$$V^\pi(\mathbf{x}) \approx \hat{V}^\pi(\mathbf{x};\boldsymbol{\phi}) = \hat{V}^\pi_{\boldsymbol{\phi}}(\mathbf{x}) \tag{3.152a}$$

$$Q^\pi(\mathbf{x},\mathbf{u}) \approx \hat{Q}^\pi(\mathbf{x},\mathbf{u};\boldsymbol{\phi}) = \hat{Q}^\pi_{\boldsymbol{\phi}}(\mathbf{x},\mathbf{u})\ . \tag{3.152b}$$

As a result, the memory requirement is significantly reduced to only the parameter vector $\boldsymbol{\phi}$. Additionally, by employing a function approximator, the agent can generalize from previously visited states or state-input pairs to unvisited ones. In an ideal scenario where $V^\pi(\mathbf{x})$ and $Q^\pi(\mathbf{x},\mathbf{u})$ are known, one could utilize supervised learning methods to approximate them with $\hat{V}^\pi$ and $\hat{Q}^\pi$. By generating rollouts

$$\boldsymbol{\tau}^{(e)} = \left(\mathbf{x}_0^{(e)}, \mathbf{u}_0^{(e)}, \mathbf{x}_1^{(e)}, \mathbf{u}_1^{(e)}, \ldots, \mathbf{x}_{N-1}^{(e)}, \mathbf{u}_{N-1}^{(e)}, \mathbf{x}_N^{(e)}\right) \tag{3.153}$$

for $e = 0, 1, \ldots, E$, with data

$$\mathcal{D} = \left\{\mathbf{x}_k^{(i)}, V^\pi\left(\mathbf{x}_k^{(e)}\right)\right\}_{e=0}^{E} \tag{3.154a}$$

$$\mathcal{D} = \left\{\left(\mathbf{x}_k^{(e)}, \mathbf{u}_k^{(e)}\right), Q^\pi\left(\mathbf{x}_k^{(e)}, \mathbf{u}_k^{(e)}\right)\right\}_{e=0}^{E}\ , \tag{3.154b}$$

the *regression task* can be formulated as follows

$$\min_{\boldsymbol{\phi}} \frac{1}{|\mathcal{D}|} \sum_{e\in\mathcal{D}} \left(\hat{V}^\pi_{\boldsymbol{\phi}}\left(\mathbf{x}_k^{(e)}\right) - V^\pi\left(\mathbf{x}_k^{(e)}\right)\right)^2 \tag{3.155}$$

and

$$\min_{\boldsymbol{\phi}} \frac{1}{|\mathcal{D}|} \sum_{e\in\mathcal{D}} \left(\hat{Q}^\pi_{\boldsymbol{\phi}}\left(\mathbf{x}_k^{(e)}, \mathbf{u}_k^{(e)}\right) - Q^\pi\left(\mathbf{x}_k^{(e)}, \mathbf{u}_k^{(e)}\right)\right)^2\ . \tag{3.156}$$

3.4.3 Deep Q-Network

Q-learning can experience instability and divergence when combined with nonlinear function approximators and bootstrapping. The Deep Q-Network (DQN) addresses these issues by introducing two key innovations to stabilize and enhance the training process:

- *Experience replay*: Instead of using sequential rollouts for updates, DQN stores a collection of steps $\mathbf{e}_k = (\mathbf{x}_k, \mathbf{u}_k, \mathsf{r}_k, \mathbf{x}_{k+1})$ in a replay memory $\mathcal{D}_k = \{\mathbf{e}_0, \ldots, \mathbf{e}_k\}$. This memory contains a diverse range of experiences from multiple past episodes. During training, mini-batches of experiences are randomly sampled from this memory

to update the Q-values. This technique not only improves the efficiency of data utilization but also breaks the correlation between consecutive samples and mitigates the non-stationary distribution issues, leading to more stable and reliable learning.

- *Periodically updated the target network*: In standard Q-learning, the same network estimates both the current and the target Q-values, leading to a moving target problem that can cause harmful correlations and oscillations. DQN addresses this by employing two separate networks: a primary network with parameters $\boldsymbol{\phi}$ for the current Q-value estimation and a target network with parameters $\boldsymbol{\phi}^-$ that remains fixed for a set number of steps C. The target network's sole purpose is to generate the stable target Q-values for the updates. Every C steps, the primary network's weights are copied to the target network, ensuring that the targets are only periodically updated, which significantly enhances the stability of the learning process.

The objective function for DQN is formulated as follows:

$$L(\boldsymbol{\phi}) = \mathbb{E}_{(\mathbf{x},\mathbf{u},\mathsf{r},\mathbf{x}')\sim U(\mathcal{D})}\left[\left(\mathsf{r}(\mathbf{x},\mathbf{u}) + \gamma \max_{\mathbf{w}\in\mathcal{U}} \hat{Q}^{\pi}_{\boldsymbol{\phi}^-}(\mathbf{x}',\mathbf{w}) - \hat{Q}^{\pi}_{\boldsymbol{\phi}}(\mathbf{x},\mathbf{u})\right)^2\right] \tag{3.157}$$

Here, $U(\mathcal{D})$ represents a uniform distribution over the replay memory $\mathcal{D}$ and $\boldsymbol{\phi}^-$ denotes the parameters of the target network. Pseudo Code of the Deep Q-Learning Algorithm is given in Algorithm 7.

Algorithm 7: Deep Q-Learning (DQN) Algorithm

```
/* Initialization                                                    */
```
Initialize policy network $\hat{Q}^{\pi}_{\boldsymbol{\phi}}$ and target network $\hat{Q}^{\pi}_{\boldsymbol{\phi}^-}$ with $\boldsymbol{\phi}^- = \boldsymbol{\phi}$;
Initialize replay memory $\mathcal{D}$ and exploration parameter $\epsilon = 1.0$;
for $e = 0, 1, 2, \ldots, E$ *episodes* **do**
  Initialize state $\mathbf{x}_0$;
  for $k = 0, 1, 2, \ldots, N-1$ *steps* **do**
    `/* ϵ-greedy action selection and execution */`
    if *random()* $< \epsilon$ **then**
      Select random action $\mathbf{u}_k$;
    else
      $\mathbf{u}_k = \arg\max_{\mathbf{w} \in \mathcal{U}} \hat{Q}^{\pi}_{\boldsymbol{\phi}}(\mathbf{x}_k, \mathbf{w})$;
    end
    Execute $\mathbf{u}_k$, observe $\mathsf{r}_k = \mathsf{r}(\mathbf{x}_k, \mathbf{u}_k)$ and $\mathbf{x}_{k+1}$;
    Store experience $\mathbf{e}_k = (\mathbf{x}_k, \mathbf{u}_k, \mathsf{r}_k, \mathbf{x}_{k+1})$ in $\mathcal{D}$;
    `/* Mini-batch learning update */`
    if $|\mathcal{D}| \geq$ *batch_size* **then**
      Sample mini-batch $(\mathbf{x}, \mathbf{u}, \mathsf{r}, \mathbf{x}') \sim U(\mathcal{D})$;
      Compute targets: $y = \mathsf{r} + \gamma \max_{\mathbf{w} \in \mathcal{U}} \hat{Q}^{\pi}_{\boldsymbol{\phi}^-}(\mathbf{x}', \mathbf{w})$ (if not terminal);
      Update: $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \alpha \nabla_{\boldsymbol{\phi}} L(\boldsymbol{\phi})$ where $L(\boldsymbol{\phi}) = \mathbb{E}[(y - \hat{Q}^{\pi}_{\boldsymbol{\phi}}(\mathbf{x}, \mathbf{u}))^2]$;
    end
    $\mathbf{x}_k \leftarrow \mathbf{x}_{k+1}$; break if episode terminated;
  end
  $\epsilon \leftarrow \max(\epsilon - \epsilon_{\text{decay}}, \epsilon_{\min})$;
  if $e \bmod C = 0$ **then**
    $\boldsymbol{\phi}^- \leftarrow \boldsymbol{\phi}$
  end
end

Note 3.12. Python code of the *Deep Q-Learning Algorithm* 7 for the Frozen Lake Example can be found here.

3.4.4 Policy Gradients

Policy gradients are a class of algorithms in reinforcement learning that optimize policies directly. They work by calculating the gradient of the expected reward concerning the policy parameters and then adjusting the parameters in the direction that increases the expected reward. This approach is particularly powerful for high-dimensional or continuous input spaces and allows for stochastic policies, offering a more nuanced way of exploring and exploiting the environment. They form the basis for many advanced reinforcement learning methods and are fundamental to understanding and developing new algorithms in the field.

The term *Policy Gradient* (PG) covers methods where a policy is parameterized by the

parameter vector $\boldsymbol{\theta}$, i.e.

$$\pi(\mathbf{u}_k \mid \mathbf{x}_k; \boldsymbol{\theta}) = \pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k) \ . \tag{3.158}$$

With this parameterized policy, a multivariate distribution

$$\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) = \mathsf{p}_0(\mathbf{x}_0) \prod_{k=0}^{N-1} \pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k) \mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \mathbf{u}_k) \tag{3.159}$$

can be introduced, which gives the probability density of observing a N-step rollout $\boldsymbol{\tau}$ under a the policy $\pi_{\boldsymbol{\theta}}$. Thus, with the *discounted return* (3.9), we obtain the parameterized action-value function

$$Q_k^{\pi_\theta}(\mathbf{x}_k, \mathbf{u}_k) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\{\mathsf{R}_k(\boldsymbol{\tau}) \mid \mathbf{x}_k = \mathbf{x}_k, \mathbf{u}_k = \mathbf{u}_k\} \ , \tag{3.160}$$

and analogously the parameterized state-value function

$$V_k^{\pi_\theta}(\mathbf{x}_k) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\{\mathsf{R}_k(\boldsymbol{\tau}) \mid \mathbf{x}_k = \mathbf{x}_k\} \ . \tag{3.161}$$

Now, the optimal control problem can be formulated as an optimization over the parameter vector $\boldsymbol{\theta}$, see [3.8, p. 96], in the form

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \ , \tag{3.162}$$

with expected return

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}_0 \sim \mathsf{p}_0(\mathbf{x}_0)}\{V_0^{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_0)\} = \int_{\mathcal{T}} \mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \mathsf{R}_0(\boldsymbol{\tau}) \mathrm{d}\boldsymbol{\tau} \ . \tag{3.163}$$

To solve (3.162) with (3.163), a gradient ascent method can be employed

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(i)}} \ , \tag{3.164}$$

where $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ represents the gradient of the objective function $J(\boldsymbol{\theta})$, and α denotes the step size or learning rate. Policy gradient methods optimize a policy directly by following the gradient of the expected reward with respect to policy parameters. The distinction between Deterministic and Stochastic Policy Gradients is in the way inputs are chosen by the policy:

- Deterministic Policy Gradients (DPG): The policy directly maps states to inputs deterministically. In Deep Deterministic Policy Gradient (DDPG), a noise process is added to the output of the actor network to promote exploration during learning. This controlled randomness in the output enables the deterministic policy to explore the action space effectively.
- Stochastic Policy Gradients (SPG): The policy outputs a probability distribution over inputs, meaning the input is sampled from this distribution, introducing randomness. SPG approaches are often advantageous in settings where exploration is essential, as the stochasticity encourages diverse input selection in the learning process.

3.4.5 Stochastic Policy Gradient

To obtain the Policy Gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ for the stochastic policy (3.158), the following procedure is performed. From (3.163), follows

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \int_{\mathcal{T}} \mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \mathsf{R}_0(\boldsymbol{\tau}) \mathrm{d}\boldsymbol{\tau} = \int_{\mathcal{T}} \nabla_{\boldsymbol{\theta}} (\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau})) \mathsf{R}_0(\boldsymbol{\tau}) \mathrm{d}\boldsymbol{\tau} \ . \tag{3.165}$$

Using the identity[11]

$$\nabla_{\boldsymbol{\theta}} (\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau})) = \mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \frac{\nabla_{\boldsymbol{\theta}} (\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}))}{\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau})} = \mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \nabla_{\boldsymbol{\theta}} \ln \left(\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \right) \ , \tag{3.166}$$

we obtain

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int_{\mathcal{T}} \nabla_{\boldsymbol{\theta}} (\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau})) \mathsf{R}_0(\boldsymbol{\tau}) \mathrm{d}\boldsymbol{\tau} = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \{ \nabla_{\boldsymbol{\theta}} \ln \left(\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \right) \mathsf{R}_0(\boldsymbol{\tau}) \} \ . \tag{3.167}$$

The term $\ln \left(\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \right)$ can be split into three components using (3.159), i.e.,

$$\ln \left(\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \right) = \ln \left(\mathsf{p}_0(\mathbf{x}_0) \right) + \sum_{k=0}^{N-1} \ln \left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k) \right) + \sum_{k=0}^{N-1} \ln \left(\mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \mathbf{u}_k) \right) \ . \tag{3.168}$$

Taking the gradient with respect to the parameter vector eliminates the terms that depend on the system dynamics results in

$$\nabla_{\boldsymbol{\theta}} \ln \left(\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \right) = \nabla_{\boldsymbol{\theta}} \sum_{k=0}^{N-1} \ln \left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k) \right) = \sum_{k=0}^{N-1} \nabla_{\boldsymbol{\theta}} \ln \left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k) \right) \ . \tag{3.169}$$

Substituting (3.169) into (3.167) with (3.160) yields the *Stochastic Policy Gradient* (SPG) for finite time horizons[12], see [3.13],

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left\{ \sum_{k=0}^{N-1} \nabla_{\boldsymbol{\theta}} \ln \left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k) \right) \mathsf{R}_0 \right\} \ . \tag{3.170}$$

Note 3.13. This does not work for deterministic policies. With a parameterized deterministic policy $\pi_{\boldsymbol{\theta}} : \mathbf{u}_k = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)$, the multivariate distribution (3.183) simplifies to

$$\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) = \mathsf{p}_0(\mathbf{x}_0) \prod_{k=0}^{N-1} \mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \mathbf{u}_k) \Big|_{\mathbf{u}_k = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)} \ . \tag{3.171}$$

Taking the logarithm yields

$$\ln \left(\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \right) = \ln \left(\mathsf{p}_0(\mathbf{x}_0) \right) + \sum_{k=0}^{N-1} \ln \left(\mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \mathbf{u}_k) \right) \Big|_{\mathbf{u}_k = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)} \ . \tag{3.172}$$

[11] Known as the log-derivative-trick.
[12] $\nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}$ is the so-called score function.

The gradient with respect to $\boldsymbol{\theta}$ is computes as

$$\begin{aligned}\nabla_\theta \ln\left(\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau})\right) &= \nabla_{\boldsymbol{\theta}} \sum_{k=0}^{N-1} \ln(\mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \mathbf{u}_k))\Big|_{\boldsymbol{\theta}(\mathbf{x}_k)}, \\ &= \nabla_{\boldsymbol{\theta}} \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k) \nabla_{\mathbf{u}_k} \ln\left(\mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \mathbf{u}_k)\right)\big|_{\mathbf{u}_k = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)},\end{aligned} \tag{3.173}$$

where the term containing the system dynamics does not cancel out anymore, unlike in (3.169). Thus, the system dynamics must then be known to calculate the policy gradient.

There are two important variants of the Stochastic Policy Gradient, see [3.14] for more information:

- *Action-Value Function Policy Gradient*: Incorporates all return information, making it theoretically complete and accurate.
- *Advantage Function Policy Gradient*: Reduces variance in gradient estimates, leading to more stable and efficient learning.

Action-Value Function Policy Gradient

Firstly, the Stochastic Policy Gradient can be formulated in term of the return R_k according to (3.9). Causality requires that future inputs and policies at time k do not depend on past rewards at time i. Thus, for $i < k$, we have

$$0 = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left\{\nabla_{\boldsymbol{\theta}} \ln\left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k)\right)\mathsf{r}_i\right\}. \tag{3.174}$$

With the definition of the return R_k, we get from (3.170)

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left\{\sum_{k=0}^{N-1} \nabla_{\boldsymbol{\theta}} \ln\left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k)\right)\left(\sum_{i=0}^{k-1} \gamma^{i-k} r_i + \underbrace{\sum_{i=k}^{N-1} \gamma^{i-k} r_i}_{=\mathsf{R}_k}\right)\right\} \tag{3.175}$$

and with the causality condition (3.174), we find another formulation of the Stochastic Policy Gradient

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left\{\sum_{k=0}^{N-1} \nabla_{\boldsymbol{\theta}} \ln\left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k)\right)\mathsf{R}_k\right\}. \tag{3.176}$$

Secondly, the Stochastic Policy Gradient can be written in term of the state-value function $Q_k^{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k)$. Define $\boldsymbol{\tau}_{[:,k]} = (\mathbf{x}_0, \mathbf{u}_0, ..., \mathbf{x}_k, \mathbf{u}_k)$ as the rollout up to time k, and $\boldsymbol{\tau}_{[k,:]}$ as the remainder of the rollout after that. Using the *law of total expectation* (A.9), we can break up (3.170) into:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{k=0}^{N-1} \mathbb{E}_{\boldsymbol{\tau}_{[:,k]} \sim \pi_{\boldsymbol{\theta}}}\left\{\mathbb{E}_{\boldsymbol{\tau}_{[k,:]} \sim \pi_{\boldsymbol{\theta}}}\left\{\nabla_\theta \ln\left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k)\right)\mathsf{R}_k \mid \boldsymbol{\tau}_{[:,k]}\right\}\right\}. \tag{3.177}$$

The gradient of the log-probability is constant with respect to the inner expectation, due to its dependency on $\mathbf{x}_k$ and $\mathbf{u}_k$, and it can be extracted:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{k=0}^{N-1} \mathbb{E}_{\boldsymbol{\tau}_{[:,k]} \sim \pi_{\boldsymbol{\theta}}} \Big\{ \nabla_\theta \ln \left(\pi_\theta(\mathbf{u}_k \mid \mathbf{x}_k) \right) \mathbb{E}_{\boldsymbol{\tau}_{[k,:]} \sim \pi_{\boldsymbol{\theta}}} \Big\{ \mathsf{R}_k \mid \boldsymbol{\tau}_{[:,k]} \Big\} \Big\} \ . \tag{3.178}$$

Lastly, the Markov property implies

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left\{ \sum_{k=0}^{N-1} \nabla_{\boldsymbol{\theta}} \ln \left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k) \right) Q_k^{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k) \right\} \ . \tag{3.179}$$

Since $\mathsf{p}^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau})$ is unknown, the expectation $\mathbb{E}_{\pi_{\boldsymbol{\theta}}}\{\cdot\}$ is approximated from $e = 1, \dots E$ rollouts $\boldsymbol{\tau}^{(e)}$, collected in the set $\mathcal{D} = \{\boldsymbol{\tau}^{(e)}\}$, by the *empirical mean* as

$$\hat{\mathbf{g}} = \widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})} = \frac{1}{|\mathcal{D}|} \sum_{\boldsymbol{\tau} \in \mathcal{D}} \sum_{k=0}^{N-1} \nabla_{\boldsymbol{\theta}} \ln \left(\pi_{\boldsymbol{\theta}}(\mathbf{x}_k \mid \mathbf{u}_k) \right) Q_k^{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k) \ . \tag{3.180}$$

Advantage Function Policy Gradient

The estimated policy gradient has high variance[13], resulting in poor convergence properties [3.15]. To reduce variance, a *baseline* $b_k(\mathbf{x}_k) \in \mathbb{R}$ can be subtracted from Q_k^π in (3.170) without affecting the expectation, see [3.8, p. 98] for a proof, resulting in

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left\{ \sum_{k=0}^{N-1} \nabla_{\boldsymbol{\theta}} \big(\ln \pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k) \big) \Big(Q_k^{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k) - b_k(\mathbf{x}_k) \Big) \right\} \ . \tag{3.181}$$

Intuitively, the variance is reduced when subtracting a baseline because the operation effectively re-centers the reward signal around a mean value. A widely used baseline is the value function $b_k(\mathbf{x}_k) = V_k^\pi(\mathbf{x}_k)$. See [3.14] for different baselines and variants of the Stochastic Policy Gradient. From (3.181), we then get

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left\{ \sum_{k=0}^{N-1} \nabla_{\boldsymbol{\theta}} \ln \left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k) \right) A_k^{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k) \right\} \ , \tag{3.182}$$

with *Advantage function* according to (3.48). The Advantage function quantifies the relative benefit of taking a specific input in a given state compared to the average input for that state.

3.4.6 Deterministic Policy Gradient

The policy gradient (PG) for deterministic policy of the form $\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)$ is derived and documented in [3.16]. The authors demonstrated that the Deterministic Policy Gradient (DPG) acts as a special case of the Stochastic Policy Gradient (SPG). For a parameterized

[13] We say that a method has high variance if you get different results when you run it multiple times. A method with less variance will give you more similar results when you run it multiple times.

deterministic policy $\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)$, we introduce the multivariate distribution

$$\mathsf{p}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) = \mathsf{p}_0(\mathbf{x}_0) \prod_{k=0}^{N-1} \mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \mathbf{u}_k)\Big|_{\mathbf{u}_k=\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)} . \tag{3.183}$$

The cost function to be minimized

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}_0\sim\mathsf{p}_0(\mathbf{x}_0)}\{V_0^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_0)\} = \int_{\mathcal{T}} \mathsf{p}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\boldsymbol{\tau})\mathsf{R}_0(\boldsymbol{\tau})\mathrm{d}\boldsymbol{\tau} \tag{3.184}$$

is differentiated with respect to $\boldsymbol{\theta}$, to obtain

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}_0\sim\mathsf{p}_0(\mathbf{x}_0)}\{\nabla_{\boldsymbol{\theta}} V_0^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_0)\} . \tag{3.185}$$

To determine the DPG, $\nabla_{\boldsymbol{\theta}} V_0^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_0)$ must be known. The calculation is then carried out for the general case $\nabla_{\boldsymbol{\theta}} V_k^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k)$, following the proof in [3.16]:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} V_k^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k) &= \nabla_{\boldsymbol{\theta}} Q_k^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)) \\
&\overset{(3.59)}{=} \nabla_{\boldsymbol{\theta}}\Big(\bar{r}_k(\mathbf{x}_k, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)) + \gamma \mathbb{E}_{\mathbf{x}_{k+1}\sim\mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1}|\mathbf{x}_k,\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k))}\Big\{V_{k+1}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_{k+1})\Big\}\Big) \\
&= \nabla_{\boldsymbol{\theta}}\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)\nabla_{\mathbf{u}_k}\bar{r}_k(\mathbf{x}_k, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)) + \gamma\nabla_{\boldsymbol{\theta}} \int_{\mathcal{X}} \mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k))V_{k+1}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_{k+1})\mathrm{d}\mathbf{x}_{k+1} ,
\end{aligned} \tag{3.186}$$

The last part can be expanded taking the gradient inside the integral

$$\begin{aligned}
&\gamma \int_{\mathcal{X}} \nabla_{\boldsymbol{\theta}}\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)\nabla_{\mathbf{u}_k}\mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k))V_{k+1}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_{k+1}) \\
&+\mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k))\nabla_{\boldsymbol{\theta}} V_{k+1}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_{k+1})\mathrm{d}\mathbf{x}_{k+1} .
\end{aligned} \tag{3.187}$$

Putting things together gives

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} V_k^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k) = \nabla_{\boldsymbol{\theta}}\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)\nabla_{\mathbf{u}_k}\Bigg(&\bar{r}_k(\mathbf{x}_k, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)) + \gamma \underbrace{\int \mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k))V_k^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_{k+1})\mathrm{d}\mathbf{x}_{k+1}}_{=\mathbb{E}_{\mathbf{x}_{k+1}\sim\mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1}|\mathbf{x}_k,\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k))}\{V_{k+1}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_{k+1})\}} \Bigg) \\
&+ \gamma \underbrace{\int \mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1} \mid \mathbf{x}_k, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k))\nabla_{\boldsymbol{\theta}} V_{k+1}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_{k+1})\mathrm{d}\mathbf{x}_{k+1}}_{=\mathbb{E}_{\mathbf{x}_{k+1}\sim\mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1}|\mathbf{x}_k,\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k))}\{\nabla_{\boldsymbol{\theta}} V_{k+1}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_{k+1})\}} .
\end{aligned} \tag{3.188}$$

And finally we get

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} V_k^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k) = \; &\nabla_{\boldsymbol{\theta}}\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)\nabla_{\mathbf{u}_k} Q_k^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k)\big|_{\mathbf{u}_k=\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)} \\
&+ \gamma\mathbb{E}_{\mathbf{x}_{k+1}\sim\mathsf{p}_{\mathsf{x}}(\mathbf{x}_{k+1}|\mathbf{x}_k,\boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k))}\Big\{\nabla_{\boldsymbol{\theta}} V_{k+1}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_{k+1})\Big\} .
\end{aligned} \tag{3.189}$$

Continuing the recursion in (3.189), substituting this into (3.185), and combining the expectation over $\boldsymbol{\tau}$, one obtains an expression for the DPG:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\tau}\sim \mathsf{p}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\boldsymbol{\tau})} \left\{ \sum_{k=0}^{N-1} \gamma^k \nabla_{\boldsymbol{\theta}} \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k) \nabla_{\mathbf{u}_k} Q_k^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k) \Big|_{\mathbf{u}_k = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)} \right\}. \tag{3.190}$$

Example 3.11 (Connection between Deterministic Policy Gradients and LQR). Next, we establish the connection between Deterministic Policy Gradients and Linear Quadratic Regulator (LQR). Consider a discrete-time linear dynamical system

$$\mathbf{x}_{k+1} = \boldsymbol{\Phi}\mathbf{x}_k + \boldsymbol{\Gamma}\mathbf{u}_k \ . \tag{3.191}$$

The action-value function reads as

$$Q^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k) = \bar{r}(\mathbf{x}_k, \mathbf{u}_k) + V^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_{k+1}) \ . \tag{3.192}$$

Assuming a quadratic reward and state-value function, i.e.,

$$\bar{r}(\mathbf{x}_k, \mathbf{u}_k) = \frac{1}{2}\Big(\mathbf{x}_k^\top \mathbf{Q}\mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R}\mathbf{u}_k\Big) \quad \text{and} \quad V^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k) = \frac{1}{2}\mathbf{x}_k^\top \mathbf{P}\mathbf{x}_k \ , \tag{3.193}$$

with positive definite matrix $\mathbf{P}$ and $\mathbf{R}$ as well as positive semi-definite matrix $\mathbf{Q}$, we get

$$Q^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k) = \frac{1}{2}\Big(\mathbf{x}_k^\top \mathbf{Q}\mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R}\mathbf{u}_k + (\boldsymbol{\Phi}\mathbf{x}_k + \boldsymbol{\Gamma}\mathbf{u}_k)^\top \mathbf{P}(\boldsymbol{\Phi}\mathbf{x}_k + \boldsymbol{\Gamma}\mathbf{u}_k)\Big) \ . \tag{3.194}$$

Taking the gradient of $Q^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}$ with respect to $\mathbf{u}_k$ results in

$$\nabla_{\mathbf{u}_k} Q^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k) = \mathbf{R}\mathbf{u}_k + \boldsymbol{\Gamma}^\top \mathbf{P}(\boldsymbol{\Phi}\mathbf{x}_k + \boldsymbol{\Gamma}\mathbf{u}_k) \tag{3.195}$$

and substituting $\mathbf{u}_k = -\mathbf{K}\mathbf{x}_k$ and evaluating at this point gives

$$\nabla_{\mathbf{u}_k} Q^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k)|_{\mathbf{u}_k = -\mathbf{K}\mathbf{x}_k} = -\mathbf{R}\mathbf{K}\mathbf{x}_k + \boldsymbol{\Gamma}^\top \mathbf{P}(\boldsymbol{\Phi} - \boldsymbol{\Gamma}\mathbf{K})\mathbf{x}_k \ . \tag{3.196}$$

Finally, the deterministic policy gradient is given by

$$\begin{aligned}
\nabla_{\mathbf{K}} J(\mathbf{K}) &= \mathbb{E}_{\boldsymbol{\tau}\sim \mathsf{p}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\boldsymbol{\tau})} \left\{ \sum_{k=0}^{N-1} \nabla_{\boldsymbol{\theta}} \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k) \nabla_{\mathbf{u}_k} Q_k^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k) \Big|_{\mathbf{u}_k = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)} \right\} && \text{(3.197a)} \\
&= \mathbb{E}_{\boldsymbol{\tau}\sim \mathsf{p}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\boldsymbol{\tau})} \left\{ \sum_{k=0}^{N-1} -\mathbf{x}_k \Big(-\mathbf{R}\mathbf{K}\mathbf{x}_k + \boldsymbol{\Gamma}^\top \mathbf{P}(\boldsymbol{\Phi} - \boldsymbol{\Gamma}\mathbf{K})\mathbf{x}_k\Big)^\top \right\} && \text{(3.197b)} \\
&= \mathbb{E}_{\boldsymbol{\tau}\sim \mathsf{p}^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\boldsymbol{\tau})} \left\{ \sum_{k=0}^{N-1} \mathbf{x}_k \mathbf{x}_k^\top \Big(\mathbf{K}^\top \mathbf{R} - (\boldsymbol{\Phi} - \boldsymbol{\Gamma}\mathbf{K})^\top \mathbf{P}\boldsymbol{\Gamma}\Big) \right\} . && \text{(3.197c)}
\end{aligned}$$

For the deterministic policy gradient framework to be mathematically self-consistent, the assumed quadratic form of the value function must be compatible with the derived optimal policy. This compatibility requires

$$V^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k) = Q^{\boldsymbol{\mu}_{\boldsymbol{\theta}}}(\mathbf{x}_k, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{x}_k)) \ , \tag{3.198}$$

which, upon substitution of our quadratic forms, becomes

$$\frac{1}{2}\mathbf{x}_k^\top \mathbf{P}\mathbf{x}_k = \frac{1}{2}\mathbf{x}_k^\top \Big[\mathbf{Q} + \mathbf{K}^\top \mathbf{R}\mathbf{K} + (\boldsymbol{\Phi} - \boldsymbol{\Gamma}\mathbf{K})^\top \mathbf{P}(\boldsymbol{\Phi} - \boldsymbol{\Gamma}\mathbf{K})\Big]\mathbf{x}_k \ . \tag{3.199}$$

Since this identity must hold for arbitrary state vectors $\mathbf{x}_k$, we can equate the quadratic coefficient matrices, yielding the Lyapunov Equation

$$\mathbf{P} = \mathbf{Q} + \mathbf{K}^\top \mathbf{R}\mathbf{K} + (\boldsymbol{\Phi} - \boldsymbol{\Gamma}\mathbf{K})^\top \mathbf{P}(\boldsymbol{\Phi} - \boldsymbol{\Gamma}\mathbf{K}) \ . \tag{3.200}$$

3.4.7 Actor-Critic Methods

Two main components in policy gradient are the policy model and the value function. It makes a lot of sense to learn the value function in addition to the policy, since knowing the value function can assist the policy update, such as by reducing gradient variance in policy gradients, and that is exactly what the Actor-Critic method does, see Figure 3.11. Actor-critic methods consist of two models:

- The *Critic* updates the value function parameters $\boldsymbol{\phi}$ and depending on the algorithm it could be action-value function $Q_{\boldsymbol{\phi}}^{\pi}$ or state-value function $V_{\boldsymbol{\phi}}^{\pi}$.
- The *Actor* updates the policy parameters $\boldsymbol{\theta}$ for $\pi_{\boldsymbol{\theta}}$, in the direction suggested by the critic.

Algorithm 8 presents the vanilla Actor-Critic algorithm with stochastic policy gradients. This foundational method is widely referred to as A2C (Advantage Actor-Critic) in the reinforcement learning literature.

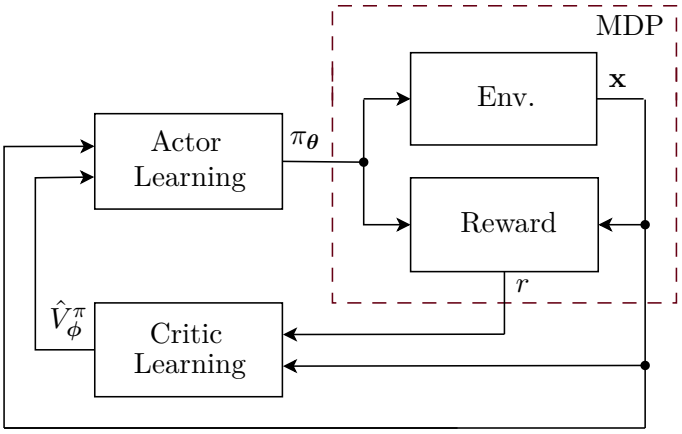


Figure 3.11: Actor-Critic Algorithm.

Algorithm 8: Vanilla Actor-Critic Stochastic Policy Gradient Algorithm

`/* Initialization */`

Initialize policy parameters $\boldsymbol{\theta}$ arbitrarily;

Initialize value function parameters $\boldsymbol{\phi}$ arbitrarily;

for $e = 0, 1, 2, \ldots, E$ *episodes* **do**

`/* Reset the environment and initial state */`

Initialize state $\mathbf{x}_0$;

`/* Collect rollouts using policy` $\pi^{(e)} = \pi\left(\boldsymbol{\theta}^{(e)}\right)$ `in environment */`

$\mathcal{D}^{(e)} = \boldsymbol{\tau}^{(1)}, \ldots, \boldsymbol{\tau}^{(E)}$;

for $k = 0, 1, 2, \ldots, N-1$ *time steps* **do**

`/* Compute Monte Carlo return */`

$\hat{R}_k^{(e)} \leftarrow \sum_{j=k}^{N-1} \gamma^{j-k} r_j$;

`/* Compute Advantage estimates */`

$\hat{A}_k^{(e)} \leftarrow \hat{R}_k^{(e)} - \hat{V}_{\boldsymbol{\phi}^{(e)}}^{\pi}$;

end

`/* Estimate policy gradient */`

$\hat{\mathbf{g}}^{(e)} \leftarrow \frac{1}{\left|\mathcal{D}^{(e)}\right|} \sum_{\boldsymbol{\tau} \in \mathcal{D}^{(e)}} \sum_{k=0}^{N-1} \nabla_{\boldsymbol{\theta}} \ln\left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k)\right)\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(e)}} \hat{A}_k^{(e)}$;

`/* Actor learning - compute policy update */`

$\boldsymbol{\theta}^{(e+1)} \leftarrow \boldsymbol{\theta}^{(e)} + \alpha \hat{\mathbf{g}}^{(e)}$;

`/* Critic learning - fit the value function by regression */`

$\boldsymbol{\phi}^{(e+1)} \leftarrow \arg\min_{\boldsymbol{\phi}} \frac{1}{\left|\mathcal{D}^{(e)}\right| N} \sum_{\boldsymbol{\tau} \in \mathcal{D}^{(e)}} \sum_{k=0}^{N-1} \left(\hat{V}_{\boldsymbol{\phi}}^{\pi} - \hat{R}_k^{(e)}\right)^2$;

end

3.4.8 Proximal Policy Optimization

Schulman proposed *Proximal Policy Optimization* (PPO) in 2017, and it has since become a widely used method in continuous model-free RL due to its simplicity and strong performance. PPO-clip is an actor-critic method and makes use of a generalized advantage estimate and a clipped surrogate objective function, see [3.17].

Generalized Advantage Estimation

PPO employs a Generalized Advantage Estimation (GAE) as its advantage function. The purpose of GAE is to significantly reduce the variance of the estimator while keeping the bias introduced as low as possible [3.18, p. 136]. This goal mirrors the fundamental principles of Eligibility Traces, cf. [3.7, p. 125]. Our present discussion is fundamentally grounded in the methodologies outlined in the supplementary material provided by [3.19]. Generally, the Monte Carlo return (3.142) serves as an unbiased estimator of the expected return at a specific state. However, each reward r_k may vary due to the environment dynamics, leading to a high variance in the overall estimation. To mitigate this, an employe an n-step return

$$\begin{aligned}\hat{R}_{[k:k+n]} &= r_k + \gamma r_{k+1} + \gamma^2 r_{k+2} + \ldots + \gamma^n r_{k+n} \\ &= \sum_{j=k}^{n-1} \gamma^{j-k} r_j + \gamma^n \hat{V}^{\pi}_{k+n}(\mathbf{x}_{k+n}) \ ,\end{aligned} \tag{3.201}$$

where we estimate the remaining return using a value function estimate $\hat{V}^{\pi}(\mathbf{x})$. It offers a lower variance but slightly biased estimate by truncating the sum of returns after n steps. Particular instances such as $\hat{R}_{[k:k+1]} = r_k + \gamma \hat{V}^{\pi}_{k+1}(\mathbf{x}_{k+1})$, used in Q-learning, and $\hat{R}_{[k,\infty]} = \sum_{j=k}^{N} \gamma^{j-k} r_j = \hat{R}_k$, which reverts to the original Monte Carlo return, illustrate the variance-bias trade-off. An alternative for balancing bias and variance is the λ-return, computed as a weighted average of n-step returns, see [3.7, p. 289], and defined as

$$\hat{R}_k(\lambda) = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \hat{R}_{[k:k+n]} \ . \tag{3.202}$$

Assuming all rewards after step N are zero, such that $\hat{R}_{[k:k+n]} = \hat{R}_{[k:N]}$ for all $n \geq N-k$, the infinite sum can be calculated according to

$$\hat{R}_k(\lambda) = (1-\lambda) \sum_{n=1}^{N-k-1} \lambda^{n-1} \hat{R}_{[k:k+n]} + \lambda^{N-k-1} \hat{R}_{[k:N]} \ . \tag{3.203}$$

Here, $\lambda = 0$ yields $\hat{R}_{[k,k+1]}$, and $\lambda = 1$ provides the full Monte Carlo return $\hat{R}_{[k,\infty]}$. Intermediate values of $\lambda \in (0,1)$ produces interpolants that can be used to balance the bias and variance of the value estimator. Updating the value function with the λ-return leads to the so-called TD(λ) algorithm, and its application for advantage estimation results in the Generalized Advantage Estimator GAE(λ)

$$\hat{A}^{\pi}_k(\lambda) = \hat{R}_k(\lambda) - \hat{V}^{\pi}_k(\mathbf{x}_k) \ . \tag{3.204}$$

For instance, setting $\lambda = 0$ yields the expression

$$\hat{A}_k^\pi(0) = \hat{R}_{[k:k+1]} - \hat{V}_k^\pi(\mathbf{x}_k) = r_k + \gamma \hat{V}_{k+1}^\pi(\mathbf{x}_{k+1}) - \hat{V}_k^\pi(\mathbf{x}_k) = \delta_k \ , \tag{3.205}$$

which is equivalent to the TD residual (3.146). Notice that $\hat{R}_k(0) = \hat{R}_{[k:k+1]} = r_k + \gamma \hat{V}_{k+1}^\pi(\mathbf{x}_{k+1})$ is the 1-step estimator for $\hat{Q}_k^\pi(\mathbf{x}_k, \mathbf{u}_k)$. Choosing a value closer to zero introduces more bias due to the immediate approximation, resulting in lower variance, as discussed in [3.14]. Conversely, employing $\lambda = 1$ leads to high variance and nearly zero bias due to the summation of rewards, i.e.,

$$\hat{A}_k^\pi(1) = \hat{R}_{[k:N]} - \hat{V}_k^\pi(\mathbf{x}_k) = \sum_{j=k}^{N} \gamma^{j-k} r_j \ . \tag{3.206}$$

which is equivalent to the Monte Carlo return (3.142). To summarize, adjusting the value of λ allows control over the tradeoff between bias and variance. A smaller value, such as $\lambda = 0$, reduces variance at the expense of introducing more bias, while a larger value, like $\lambda = 1$, results in higher variance with minimal bias due to reward summation. The GAE can be efficiently computed using the parametrized TD residual

$$\delta_k = r_k + \gamma \hat{V}_{k+1}^\pi(\mathbf{x}_{k+1}) - \hat{V}_k^\pi(\mathbf{x}_k) \ . \tag{3.207}$$

The GAE formulation then becomes

$$\hat{A}_k(\lambda) = \sum_{j=k}^{N-1} (\gamma\lambda)^{j-k} \delta_j \ . \tag{3.208}$$

Rather than computing the full sum for each time step, GAE can be calculated recursively using the backward recurrence relation

$$\hat{A}_k(\lambda) = \delta_k + \gamma\lambda \hat{A}_{k+1}(\lambda) \ , \tag{3.209}$$

with boundary condition $\hat{A}_N(\lambda) = 0$. This recursion is evaluated in reverse chronological order for $k = N-1, N-2, \ldots, 0$, requiring only a single backward pass through the trajectory.

Clipping

PPO-clip updates policies via

$$\theta^{(i+1)} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{u}_k \sim \pi_{\boldsymbol{\theta}^{(i)}}} \left\{ L(\mathbf{x}_k, \mathbf{u}_k, \boldsymbol{\theta}^{(i)}, \boldsymbol{\theta}) \right\} \ , \tag{3.210}$$

typically taking multiple steps of (usually mini-batch) Stochastic Gradient Ascent (SGA) to maximize the objective function. The (clipped surrogate) objective function L is given by, see [3.17],

$$L\Big(\mathbf{x}_k, \mathbf{u}_k, \boldsymbol{\theta}^{(i)}, \boldsymbol{\theta}\Big) = \min\left(l_k A^{\pi_{\boldsymbol{\theta}^{(i)}}}(\mathbf{x}_k, \mathbf{u}_k),\ \mathrm{clip}(l_k, \epsilon) A^{\pi_{\boldsymbol{\theta}^{(i)}}}(\mathbf{x}_k, \mathbf{u}_k) \right) \ , \tag{3.211}$$

with likelihood ratio

$$l_k = l_k\Big(\boldsymbol{\theta}^{(i)}, \boldsymbol{\theta}\Big) = \frac{\pi_{\boldsymbol{\theta}}(\mathbf{x}_k, \mathbf{u}_k)}{\pi_{\boldsymbol{\theta}^{(i)}}(\mathbf{x}_k, \mathbf{u}_k)} = \frac{\text{new policy}}{\text{old policy}} \tag{3.212}$$

and clipping operator

$$\text{clip}(l_k, \epsilon) = \begin{cases} 1-\epsilon & \text{if} \quad l_k < 1-\epsilon \\ 1+\epsilon & \text{if} \quad l_k > 1+\epsilon \\ l_k & \text{else} \ . \end{cases} \tag{3.213}$$

The hyperparameter ϵ is a small positive constant, typically set to a value like 0.2, see Figure 3.12 for an illustration. The purpose of this ϵ-clipping is to prevent overly large policy updates, thereby maintaining stability in the learning process. This is achieved

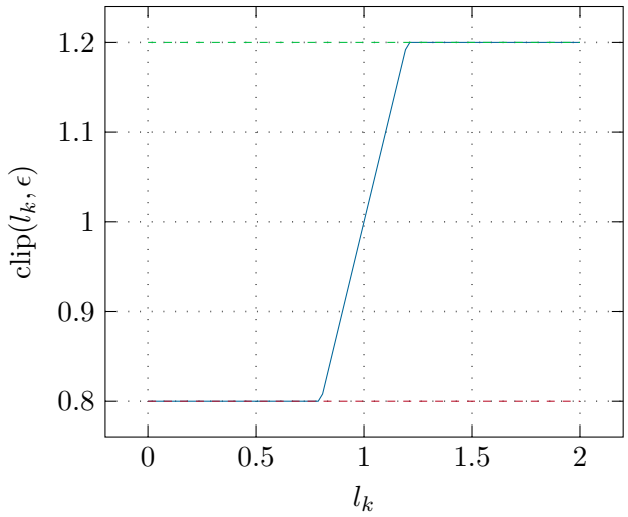


Figure 3.12: Clipping function (3.213) with $\epsilon = 0.2$.

by ensuring that the current policy does not deviate excessively from the older one. The term l_k represents the probability ratio between the current and old policy. Let's interpret the implications of l_k:

- If $l_k > 1$, it signifies that for a given state $\mathbf{x}_k$, the corresponding action $\mathbf{u}_k$ is more probable under the current policy than it was under the old policy.
- Conversely, if l_k is between 0 and 1, the action $\mathbf{u}_k$ is less likely under the current policy compared to the old one.

This likelihood ratio serves as a straightforward method to gauge the divergence between the old and current policy. Algorithm 9 shows vanilla Proximal Policy Optimization with Clipping.

Algorithm 9: Proximal Policy Optimization with Clipping

```
/* Initialization                                                    */
```
Initialize policy parameters $\boldsymbol{\theta}$, old policy parameters $\boldsymbol{\theta}_{\text{old}}$;
Initialize value function parameters $\boldsymbol{\phi}$;
for $e = 1, 2, 3, \ldots$ *episodes* **do**
```
    /* Collect rollouts using the old policy π^(e) = π(θ^(e)) in
       environment                                                   */
```
    $\mathcal{D}^{(e)} = \{\boldsymbol{\tau}^{(1)}, \ldots, \boldsymbol{\tau}^{E}\}$;
    for $k = 0, 1, 2, \ldots, N-1$ *time steps* **do**
```
        /* Compute TD residuals                                      */
```
        $\delta_k^{(e)} \leftarrow r_k^{(e)} + \gamma \hat{V}_{\boldsymbol{\phi}}^{\pi}\left(\mathbf{x}_{k+1}^{(e)}\right) - \hat{V}_{\boldsymbol{\phi}}^{\pi}\left(\mathbf{x}_k^{(e)}\right)$;
    end
```
    /* Compute generalized advantage estimates via backward recursion
       */
```
    $\hat{A}_N^{(e)}(\lambda) \leftarrow 0$; // Boundary condition
    for $k = N-1, N-2, \ldots, 0$; // Backward pass
    do
        $\hat{A}_k^{(e)}(\lambda) \leftarrow \delta_k^{(e)} + \gamma\lambda \hat{A}_{k+1}^{(e)}(\lambda)$;
    end
```
    /* Compute GAE returns for critic training                       */
```
    for $k = 0, 1, 2, \ldots, N-1$ *steps* **do**
        $\hat{R}_k^{(e)}(\lambda) \leftarrow \hat{A}_k^{(e)}(\lambda) + \hat{V}_{\boldsymbol{\phi}}^{\pi}\left(\mathbf{x}_k^{(e)}\right)$;
    end
```
    /* Actor learning - update policy parameters via SGA in
       mini-batch                                                    */
```
    $l_k^{(e)} \leftarrow \frac{\pi_{\boldsymbol{\theta}}(\mathbf{x}_k, \mathbf{u}_k)}{\pi_{\boldsymbol{\theta}^{(e)}}(\mathbf{x}_k, \mathbf{u}_k)}$;
    $\boldsymbol{\theta}^{(e+1)} \leftarrow \arg\max_{\boldsymbol{\theta}} \frac{1}{\left|\mathcal{D}^{(e)}\right|} \sum_{\boldsymbol{\tau} \in \mathcal{D}^{(e)}} \sum_{k=0}^{N-1} \min\left(l_k^{(e)} \hat{A}_k^{(e)}(\lambda),\ \text{clip}\left(\epsilon, l_k^{(e)}\right) \hat{A}_k^{(e)}(\lambda)\right)$;
```
    /* Critic learning - fit the value function by regression        */
```
    $\boldsymbol{\phi}^{(e+1)} \leftarrow \arg\min_{\boldsymbol{\phi}} \frac{1}{\left|\mathcal{D}^{(e)}\right| N} \sum_{\boldsymbol{\tau} \in \mathcal{D}^{(e)}} \sum_{k=0}^{N-1} \left(\hat{V}_{\boldsymbol{\phi}}^{\pi}(\mathbf{x}_k) - \hat{R}_k^{(e)}\right)^2$;
end

3.4.9 Off-Policy Policy Gradient

The vanilla versions of the actor-critic method are exclusively on-policy: training samples are collected according to the *target policy*, which is the same policy we aim to optimize. A simpler strategy involves two policies: the *target policy* π, which is studied and optimized, and the *behavior policy* β, which drives exploratory actions. Here, learning is based on data "off" the target policy, and the entire mechanism is labeled as off-policy learning. Off-policy methods offer several distinct advantages, including:

- The off-policy approach doesn't require a complete rollout, and it can reuse past

episodes (via experience replay), resulting in significantly improved sample efficiency[14].

- The sample collection employs a behavior policy different from the target policy, facilitating more effective exploration.

Nearly all off-policy methods employ *importance sampling* to estimate expected values from a distribution different than the sample source. In the context of off-policy learning, returns are weighted by the importance sampling ratio, which quantifies the relative likelihood of specific trajectories under the target and behavior policies. Given an initial state $\mathbf{x}_k$, the probability of a future state-input trajectory under any policy π is given by (3.42). Thus, the relative probability of the trajectory under the target and behavior policies, that it the *importance sampling ratio*, is

$$\rho_{[k:N-1]} \doteq \frac{\prod_{j=k}^{N-1} \mathsf{p}_{\mathsf{x}}(\mathbf{x}_{j+1} \mid \mathbf{x}_j, \mathbf{u}_j)\pi(\mathbf{u}_j \mid \mathbf{x}_j)}{\prod_{j=k}^{N-1} \mathsf{p}_{\mathsf{x}}(\mathbf{x}_{j+1} \mid \mathbf{x}_j, \mathbf{u}_j)\beta(\mathbf{u}_j \mid \mathbf{x}_j)} = \prod_{j=k}^{N-1} \frac{\pi(\mathbf{u}_j \mid \mathbf{x}_j)}{\beta(\mathbf{u}_j \mid \mathbf{x}_j)} \,. \tag{3.214}$$

Although the trajectory probabilities depend on the MDP's transition probabilities, which are generally unknown, they appear identically in both the numerator and denominator, and thus cancel. The importance sampling ratio ends up depending only on the two policies and the sequence, not on the MDP.

Remember that we aim to estimate the expected returns under the target policy π, but only possess returns R_k^β from the behavior policy β. These returns exhibit an incorrect expectation denoted by $V^\beta(\mathbf{x}) = \mathbb{E}_\beta\Big\{\mathsf{R}_k^\beta \mid \mathbf{x}_k = \mathbf{x}\Big\}$. To correct this, importance sampling is employed. Using the ratio $\rho_{[k:N-1]}$, we can transform these returns to align with the desired expectation

$$V^\pi(\mathbf{x}) = \mathbb{E}_\pi\Big\{\rho_{[k:N-1]}\mathsf{R}_k^\beta \mid \mathbf{x}_k = \mathbf{x}\Big\} \,. \tag{3.215}$$

Now let's see how off-policy policy gradient is computed. Since the training observations are sampled by $\mathbf{u}_k \sim \beta(\mathbf{u}_k \mid \mathbf{x}_k)$, we can rewrite the gradient following certain calculations, see [3.20] for a proof,

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\beta\left\{\sum_{k=0}^{N-1} \frac{\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k)}{\beta(\mathbf{u}_k \mid \mathbf{x}_k)} \nabla_{\boldsymbol{\theta}} \ln\left(\pi_{\boldsymbol{\theta}}(\mathbf{u}_k \mid \mathbf{x}_k)\right) Q_k^{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_k, \mathbf{u}_k)\right\} \,. \tag{3.216}$$

Here, $\frac{\pi_{\boldsymbol{\theta}}(\mathbf{u}_k|\mathbf{x}_k)}{\beta(\mathbf{u}_k|\mathbf{x}_k)}$ is the *importance weight*. In essence, when implementing policy gradient in the off-policy setting, we can adjust it with a weighted sum, where the weight is the ratio of the target policy to the behavior policy.

3.4.10 Practical implementations

Reinforcement learning has seen significant advances with the development of various Actor-Critic Algorithms, each addressing specific challenges in policy optimization and value

[14] Sample efficiency in the context of control technology and machine learning refers to the ability of a system to achieve high performance with a limited number of data samples.

function estimation. Table 3.5 provides a comprehensive comparison of five prominent deep reinforcement learning algorithms: A2C, PPO, DDPG, TD3, and SAC. These algorithms represent different approaches to balancing exploration-exploitation trade-offs, sample efficiency, and training stability in both discrete and continuous action spaces.

The comparison encompasses key algorithmic properties, architectural choices, and practical considerations that influence algorithm selection for specific applications. While A2C serves as a foundational synchronous actor-critic method, PPO introduces policy constraints for improved stability. The off-policy algorithms (DDPG, TD3, SAC) leverage experience replay for enhanced sample efficiency, with TD3 and SAC addressing specific limitations of DDPG through twin critics and entropy regularization, respectively. For practical implementation, these algorithms are readily available in established reinforcement learning libraries. Stable Baselines3 provides robust, well-documented implementations, while CleanRL offers minimalistic, research-oriented implementations that prioritize code clarity and reproducibility.

> *Note* 3.14. Python scripts are made available for the Pendulum Example using Stable Baselines3 and CleanRL.

Table 3.5: Comparison of Actor-Critic Algorithms.

Algorithm	**A2C**	**PPO**	**DDPG**	**TD3**	**SAC**
Full Name	Advantage Actor-Critic	Proximal Policy Optimization	Deep Deterministic Policy Gradient	Twin Delayed DDPG	Soft Actor-Critic
Action Space	Discrete & Continuous	Discrete & Continuous	Continuous	Continuous	Continuous
Policy Type	Stochastic	Stochastic	Deterministic	Deterministic	Stochastic
Learning Type	On-policy	On-policy	Off-policy	Off-policy	Off-policy
Key Innovation	Synchronous updates	Clipped surrogate objective	Continuous control with DPG	Delayed policy updates	Maximum entropy framework
Sample Efficiency	Low	Medium	High	High	High
Stability	Medium	High	Medium	High	High
Hyperparameter Sensitivity	Medium	Low	High	Medium	Low
Computational Cost	Low	Medium	Medium	Medium	High
Main Advantages	Simple, fast training	Stable, robust performance	Sample efficient for continuous control	Addresses overestimation bias	Automatic exploration, robust
Main Limitations	High variance, sample inefficient	Can be conservative	Sensitive to hyperparameters	Deterministic policy limits exploration	High computational overhead
Best Use Cases	Simple environments, fast prototyping	General purpose, stable training	Robotics, continuous control	Improved DDPG applications	Complex environments, robust learning

3.5 Literatur

[3.1] D. P. Bertsekas, *Reinforcement Learning and Optimal Control*. Athena Scientific, 2019.

[3.2] M. P. Deisenroth, A. Faisal, and C. S. Ong, *Mathematics for Machine Learning*. Cambridge University Press, 2020. [Online]. Available: https://mml-book.github.io/.

[3.3] E. T. Jaynes, *Probability Theory: The Logic of Science*. Cambridge University Press, 2013, vol. 1.

[3.4] D. P. Bertsekas, *Dynamic Programming and Optimal Control*. Athena Scientific, 2005, vol. 1.

[3.5] M. L. Puterman, *Markov Decision Processes*. John Wiley & Sons, 2005.

[3.6] C. Szepesvari, *Algorithms for Reinforcement Learning*. Morgan & Claypool, 2009.

[3.7] R. S. Sutton and A. G. Barto, *Reinforcement Learning*. MIT Press, 2018.

[3.8] M. Sugiyama, *Statistical Reinforcment Learning*. CRC Press, 2015.

[3.9] D. Silver, *Reinforcement learning: An introduction*, 2015. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.

[3.10] F. L. Lewis, D. L. Varbie, and V. Syrmos, *Optimal Control*. John Wiley & Sons, 2012.

[3.11] M. Vidyasagar, "A tutorial introduction to reinforcement learning," 2023. [Online]. Available: https://arxiv.org/abs/2304.00803v1.

[3.12] J. Schulman, "Optimizing expectations: From deep reinforcement learning to stochatic computation graphs," Ph.D. dissertation, University of California, Berkeley, 2016.

[3.13] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, pp. 1238–1274, 2013.

[3.14] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *ICLR 2015*, 2015. [Online]. Available: arXiv:1506.02438.

[3.15] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural Networks*, vol. 21, no. 4, pp. 682–697, 2008.

[3.16] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmille, "Deterministic policy gradient algorithms," in *Proceedings ofthe 31th International Conference on Machine Learning, ICML 32*, 2014.

[3.17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. [Online]. Available: arXiv:1707.06347.

[3.18] L. Graesser and W. Keng, *Foundation of Deep Reinforcement Learning*. Addison-Wesley, 2020.

[3.19] X. Peng, P. Abbeel, S. Levine, and M. van de Panne, "Deepmimic: Example-guided deep reinforcement learning of physics-based character skills," 2018. [Online]. Available: arXiv:1804.02717.

[3.20] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," in *Proceedings of the 29 th International Conference on Machine Learning*, Edinburgh, Scotland, UK, 2012.

[3.21] C. M. Bischop, *Pattern Recognition and Machine Learning*. Springer, 2006.