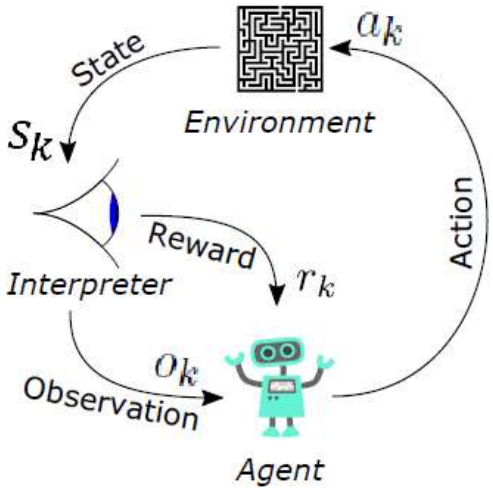
Content

Basic Terminology

1.1 Overview



At each step k the agent:

- Picks an action a_k.
- Receives an observation o_k.
- Receives a reward r_k.

At each step k the environment:

- Receives an action a_k.
- Emits an observation o_{k+1}.
- Emits a reward r_{k+1}.

The time increments $k \leftarrow k + 1$.

1.2 Reward

The reward is a simple random variable R_k with realizations r_k. It is often considered a real number $r_k \in \mathbb{R}$ or an integer $r_k \in \mathbb{Z}$. The reward gets passed from the environment to the agent. The agent's goal is to maximize the total/cumulative amount of reward it receives.

Theorem - All goals can be described by the maximization of the expected cumulative reward:

$$\max \mathbb{E}\left[\sum_{i=0}^{\infty} R_{k+i+1}\right]$$

In general, we seek to maximize the expected *return*.

Designing Reward Functions - For example, to make a robot learn to walk, researchers have provided reward on each time step proportional to the robot's forward motion. In making a robot learn how to escape from a maze, the reward is often −1 for every time step that passes prior to escape; this encourages the agent to escape as quickly as possible. To make a robot learn to find and collect empty soda cans for recycling, one might give it a reward of zero most of the time, and then a reward of +1 for each can collected. One might also want to give the robot negative rewards when it bumps into things or when somebody yells at it. For an agent to learn to play checkers or chess, the natural rewards are +1 for winning, −1 for losing, and 0 for drawing and for all nonterminal positions. The reward signal is your way of communicating to the agent what you want achieved, not how you want it achieved.

1.3 Return

The return g_t is defined as some specific function of the reward sequence. In the simplest case, this is the case for *episodic* tasks, the return is the sum of the rewards:

$$g_k = r_{k+1} + r_{k+2} + \cdots + r_N$$

where N is a final time step. This approach makes sense in applications in which there is a natural notion of final time step, that is, when the agent–environment interaction breaks naturally into subsequences, which we call *episodes*. Each episode ends in a special state called the terminal state, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. The next episode always begins independently of how the previous one ended. Thus, the episodes can all be considered to end in the same terminal state, with different rewards for the different outcomes.

. For continuing tasks, problems that lack a natural end, the return should be *discounted* to prevent infinite numbers.

$$g_k = \sum_{i=0}^{\infty} \gamma^i r_{k+i+1}$$

Here, $\gamma \in \{\mathbb{R} \mid 0 \leq \gamma \leq 1\}$, is the **discount rate**. The discount rate determines the present value of future rewards: a reward received i time steps in the future is worth only γ^{i-1} times what it would be worth if it were received immediately. If $\gamma < 1$, the infinite sum above has a finite value as long as the reward sequence $\{R_k\}$ is bounded. If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose a_k so as to maximize only R_{k+1}. As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted. Returns at successive time steps are related to each other in a way that is important for the theory and algorithms of reinforcement learning:

$$\begin{aligned} g_k &= r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \gamma^3 r_{k+4} + \cdots \\ &= r_{k+1} + \gamma(r_{k+2} + \gamma^2 r_{k+3} + \gamma^3 r_{k+4} + \cdots) \\ &= r_{k+1} + \gamma g_{k+1} \end{aligned}$$

Bold symbols are non-scalar multidimensional quantities e.g. vectors and matrices.

CAPITAL symbols denote random variables and small symbols their realizations.

1.4 Environment State

The environment state is a random variable $\boldsymbol{S}_k^e$ with realizations $\boldsymbol{s}_k^e$. It is an internal status representation of the environment. It can be fully, limited or not at all visible by the agent. In general, an agent's observations are a function of the environment state:

$$\boldsymbol{o}_k = \boldsymbol{f}(\boldsymbol{S}_k)$$

1.5 Agent State

The agent state is a random variable $\boldsymbol{S}_k^a$ with realizations $\boldsymbol{s}_k^a$. It is an internal status representation of the environment by the agent. In general: $\boldsymbol{s}_k^a \neq \boldsymbol{s}_k^e$, e.g., due to measurement noise. It can also be a condensed version, consisting only of information relevant for next action.

1.6 History and Information State

The history is the past sequence of all observations $\boldsymbol{o}_k$, actions $\boldsymbol{a}_k$ and rewards $\boldsymbol{r}_k$ up to time step k.

$$\mathbb{H}_k = \{\boldsymbol{o}_0, \boldsymbol{a}_0, \boldsymbol{r}_0, \dots, \boldsymbol{o}_k, \boldsymbol{a}_k, \boldsymbol{r}_k\}$$

If the current state $\boldsymbol{s}_k$ contains all useful information from the history, it is called an *information-* or *Markov state*:

$$\mathbb{P}[\boldsymbol{S_{k+1}}|\boldsymbol{S_k}] = \mathbb{P}[\boldsymbol{S_{k+1}}|\boldsymbol{S_0}, \boldsymbol{S_1}, \dots, \boldsymbol{S_k}]$$

The history is then fully condensed in $\boldsymbol{S}_k$, meaning that $\boldsymbol{S}_{k+1}$ is only depending on $\boldsymbol{S}_k$. A system can thereby fully be described by $\boldsymbol{S}_k$ and further past observations $\boldsymbol{S}_{k-1}, \boldsymbol{S}_{k-2}, \dots$ are irrelevant.

1.7 Action

An action $\boldsymbol{a}_k$ is the agent's degree of freedom in order to maximize its reward. Theres a distinction between a finite number of actions (*finite action set*, FAS) and infinite number of actions (continuous action set, CAS).

1.8 Policy

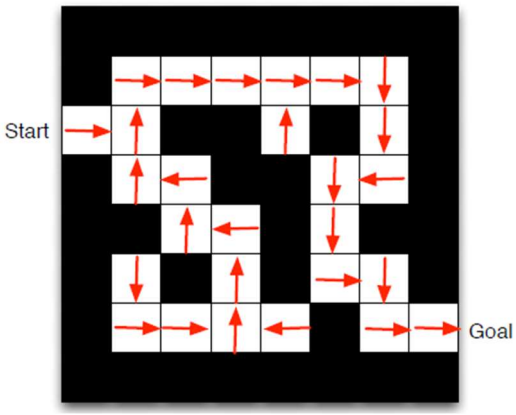
A policy $\boldsymbol{\pi}$ is the agent's internal strategy on picking actions. Deterministic policies map state and action directly:

$$\boldsymbol{a}_k = \boldsymbol{\pi}(\boldsymbol{s}_k)$$

Stochastic policies map a probability of the action given a state:

$$\boldsymbol{\pi}(\boldsymbol{A}_k|\boldsymbol{S}_k) = \mathbb{P}[\boldsymbol{A}_k|\boldsymbol{S}_k]$$

Reinforcement learning is all about changing $\boldsymbol{\pi}$ over time in order to maximize the expected return g_k.



Maze Example: RL-Solution by Policy
Arrows represent policy $\pi(s_k)$. Reward: r_k = –1

1.9 Model

A model predicts what will happen inside an environment. Such a model could be the:

State (transition) model $\boldsymbol{\mathcal{P}}$:

$$\boldsymbol{\mathcal{P}} = \mathbb{P}[\boldsymbol{s}_{k+1}|\boldsymbol{s}_k, \boldsymbol{a}_k]$$

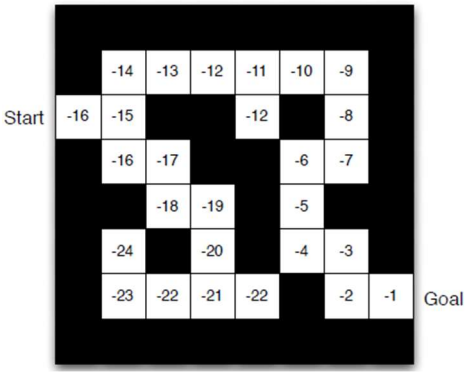
Or a reward model $\mathcal{R}$:

$$\mathcal{R} = \mathbb{P}[R_{k+1}|\boldsymbol{s}_k, \boldsymbol{a}_k\,]$$

In general, those models could be stochastic (as above) but in some problems relax to a deterministic form. Using data in order to fit a model is a learning problem of its own and often called *system identification*.

1.10 Value Functions

Almost all reinforcement learning algorithms involve estimating value functions—functions of states (or of state–action pairs) that estimate *how good* it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of "how good" here is defined in terms of future rewards that can be expected, or to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, called policies.



Maze Example: RL-Solution by Value Function
Numbers represent value $v_\pi(s_k)$
Reward: $r_k = -1$

The *state-value function* v_π is the expected return of being in state $\boldsymbol{s}_k$ following a policy $\boldsymbol{\pi}$. Assuming a MDP problem structure the state-value function is:

$$v_\pi(\boldsymbol{s}_k) = \mathbb{E}_\pi[G_k|\boldsymbol{s}_k] = \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} \gamma^i R_{k+i+1} \mid \boldsymbol{s}_k\right]$$

The *action-value function* q_π is the expected return of being in state $\boldsymbol{s}_k$ taking an action $\boldsymbol{a}_k$ and, thereafter, following a policy $\boldsymbol{\pi}$: $q_\pi(\boldsymbol{s}_k, \boldsymbol{a}_k)$. Assuming a MDP problem structure the action-value function is:

$$q_\pi(\boldsymbol{s}_k, \boldsymbol{a}_k) = \mathbb{E}_\pi[G_k|\boldsymbol{s}_k, \boldsymbol{a}_k] = \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} \gamma^i R_{k+i+1} \mid \boldsymbol{s}_k, \boldsymbol{a}_k\right]$$

A key task in reinforcement learning is to estimate $v_\pi(\boldsymbol{s}_k)$ and $q_\pi(\boldsymbol{s}_k, \boldsymbol{a}_k)$ based on sampled data.

1.11 Reinforcement Learning vs. Planning

There are two fundamental solutions to sequential decision making:

- Reinforcement *learning*: the environment is initially unknown, the agents interact with the environment, the policy is improved based on environment feedback (reward).
- *Planning*: An a priori environment model exists, the agent interacts with its own model, the policy is improved based on the model feedback ('virtual reward').

Above the two extreme cases are confronted:

- RL = only learning without using available pre-knowledge.
- Planning = iterating on a model without improving it based on data.

2 Markov Decision Processes

Markov decision processes (MDPs) are a *mathematically idealized* form of RL problems. They allow precise theoretical statements (e.g., on optimal solutions) and deliver insights into suitable RL algorithms since real-world problems can be abstracted as MDPs.

		States observable: Yes	States observable: No
Actions	Yes	Markov Decision Process (MDP)	Partially observable MDP
	No	Markov Chain	Hidden Markov Model

2.1 Finite Markov Chains

Given a Markov state $S_k = s$ and its successor $S_{k+1} = s'$, the *state transition probability* $\forall\{s, s'\} \in \mathcal{S}$ is defined by the matrix:

$$\boldsymbol{\mathcal{P}}_{ss'} = \mathbb{P}[S_{k+1} = s' | S_k = s]$$

Here, $\boldsymbol{\mathcal{P}}_{ss'} \in \mathbb{R}^{n \times n}$ has the form

$$\boldsymbol{\mathcal{P}}_{ss'} = \begin{bmatrix} p_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nn} \end{bmatrix}$$

with $p_{ij} \in \{\mathbb{R} | 0 \leq p_{ij} \leq 1\}$ (meaning that the probability $p_{ij}$is between 0 and 1) being the specific probability to go from state $s = S_i$ to state $s' = S_j$. Obviously, $\forall i$: $\sum_j p_{ij} = 1$ must hold (all transitions from one state [rows] sum to one).

A *finite Markov chain* is a tuple $\langle \mathcal{S}, \boldsymbol{\mathcal{P}} \rangle$ with

- S being a finite set of discrete-time states $S_k \in \mathcal{S} \rightarrow$ state space
- $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{P}}_{ss'} = \mathbb{P}[s'|s]$ is the state transition probability

This is a specific stochastic (state transitions are not deterministic) process model with a sequence of random variables S_k, S_{k+1}, … that is "memoryless" (only Markov states).

Define $p_k \in \mathbb{R}^n$ as a *probability row vector* where $p_{i,k}$ gives the probability of being in state $S_k \in \mathcal{S}$ at time-step k. The probability of being in state S_{k+m} at time-step $(k + m)$ starting from state s_k is given by:

$$\boldsymbol{p}_{k+m} = \boldsymbol{p}_k * (\boldsymbol{\mathcal{P}}_{ss'})^m$$

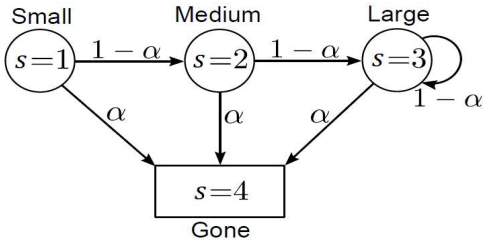
Hence, for $m \rightarrow \infty$, the *steady state* (when no state change occurs anymore for further time-steps), the following equation must hold:

$$\boldsymbol{p} = \boldsymbol{p}\boldsymbol{\mathcal{P}}_{ss'}$$

Solving above equation for $\boldsymbol{p}$ given the transitions probability $\boldsymbol{\mathcal{P}}_{ss'}$ gives insights which states of the Markov chain are visited in the long run.

Example

A growing forest tree modeled as a Markov chain.



$$s = s \in \{1,2,3,4\}$$
$$= \{\text{small, medium, large, gone}\}$$

$$\mathcal{P} = \begin{bmatrix} 0 & 1-\alpha & 0 & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- At $s = 1$ a small tree is planted (starting point)
- The tree grows with probability $p = 1 - \alpha$
- If it reaches $s = 3$ (large) its growth is limited
- With probability α a natural hazard destroys the tree
- The state $s = 4$ is terminal

❓ For $k = 0$ the initial state probability is given by $\boldsymbol{p}_{k=0} = [1 \quad 0 \quad 0 \quad 0]$, i.e., we are in state $s = 1$. What is the state probability after $k = 1,2,\ldots$ steps with $\alpha = 0.2$?

$$\boldsymbol{p}_1 = \begin{bmatrix} 0 & 0.8 & 0 & 0.2 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}}_{\boldsymbol{p}_0} \underbrace{\begin{bmatrix} 0 & 0.8 & 0 & 0.2 \\ 0 & 0 & 0.8 & 0.2 \\ 0 & 0 & 0.8 & 0.2 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathcal{P}},$$

$$\boldsymbol{p}_2 = \begin{bmatrix} 0 & 0 & 0.64 & 0.36 \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 0.8 & 0 & 0.2 \end{bmatrix}}_{\boldsymbol{p}_1} \underbrace{\begin{bmatrix} 0 & 0.8 & 0 & 0.2 \\ 0 & 0 & 0.8 & 0.2 \\ 0 & 0 & 0.8 & 0.2 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathcal{P}},$$

$$\boldsymbol{p}_3 = \begin{bmatrix} 0 & 0 & 0.51 & 0.49 \end{bmatrix}, \quad \boldsymbol{p}_{10} = \begin{bmatrix} 0 & 0 & 0.11 & 0.89 \end{bmatrix},$$
$$\boldsymbol{p}_\infty = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}.$$

💡 $\boldsymbol{p}_\infty$ is the *steady state* of the tree being "gone".

2.2 Finite Markov Reward Processes (MRP)

A *finite Markov reward process* (MRP) is a tuple $\langle \mathcal{S}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$ with

- S being a finite set of discrete-time states $S_k \in \mathcal{S} \rightarrow$ state space
- $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{P}}_{ss'} = \mathbb{P}[s'|s]$ is the state transition probability
- $\mathcal{R}$ is a reward function $\mathcal{R} = \mathcal{R}_s = \mathbb{E}[R_{k+1}|s_k]$
- γ is the discount factor $\gamma \in \{\mathbb{R}|0 \leq \gamma \leq 1\}$

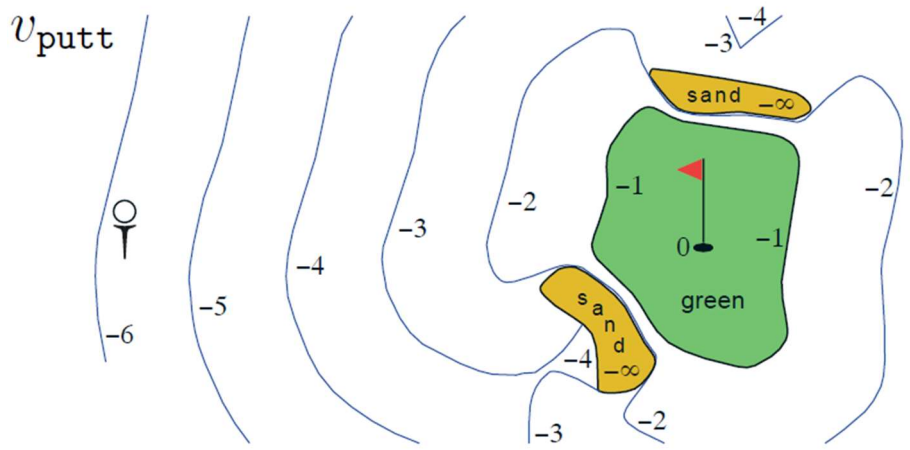
This extends a Markov chain with rewards. It's still an autonomous stochastic process without specific inputs. Rewards R_k only depends on state S_k.

2.2.1 Value Function in MRP

The *state-value function* $v(s_k)$ of an MRP is the expected return starting from state s_k:

$$v(s_k) = \mathbb{E}[G_k|S_k = s_k]$$

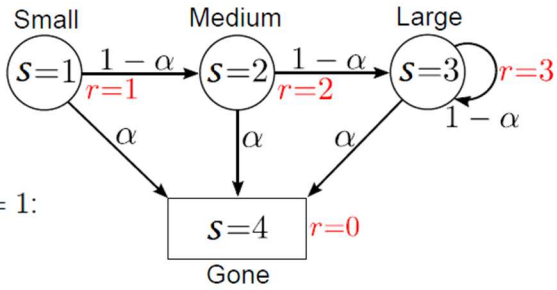
This represents the long-term value of being in state S_k.



Isolines indicate state value of different golf ball locations.

Example

- Growing larger trees is rewarded
- Losing a tree due to hazard is unrewarded



Sampling the value function to approximate it:

Exemplary samples $\hat{v}$ with $\gamma = 0.5$ starting in $s = 1$:

$s = 1 \rightarrow 4, \quad \hat{v} = 1,$

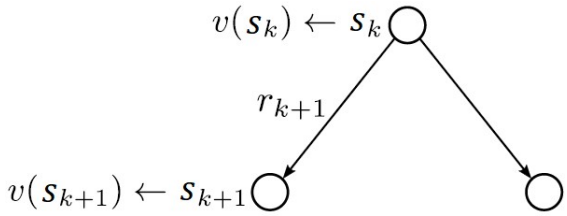
$s = 1 \rightarrow 2 \rightarrow 4, \quad \hat{v} = 1 + 0.5 \cdot 2 = 2.0,$

$s = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4, \quad \hat{v} = 1 + 0.5 \cdot 2 + 0.25 \cdot 3 = 3.75,$

$s = 1 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow 4, \quad \hat{v} = 1 + 0.5 \cdot 2 + 0.25 \cdot 3 + 0.125 \cdot 3 = 4.13.$

2.2.2 Bellman-Equation for MRPs

$$
\begin{aligned}
v(s_k) &= \mathbb{E}[G_k|S_k = s_k] \\
&= \mathbb{E}[R_{k+1} + \gamma R_{k+2} + \gamma^2 R_{k+3} + \cdots |S_k = s_k] \\
&= \mathbb{E}[R_{k+1} + \gamma(R_{k+2} + \gamma^2 R_{k+3} + \cdots)|S_k \\
&= s_k] = \mathbb{E}[R_{k+1} + \gamma G_{k+1}|S_k = s_k] \\
&= \mathbb{E}[R_{k+1} + \gamma v(S_{k+1})|S_k = s_k]
\end{aligned}
$$

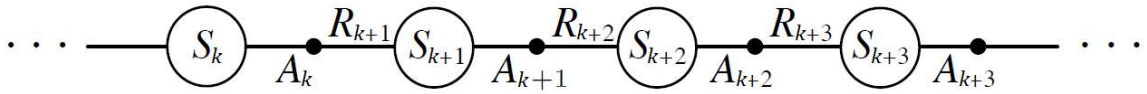


Backup diagram for $v(s_k)$

At time-step k the reward r_{k+1} is paid out, so, state $s = 1$ gets the reward r_2.

The return g_k is defined (see: *1.3 Return*) as:

$$g_k = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \gamma^3 r_{k+4} + \cdots = r_{k+1} + \gamma g_{k+1}$$



which can be seen as the current reward plus the discounted future return. Don't get confused by the index shift in this step!

Assuming non-random rewards for every state $S = s \in \mathcal{S}$

$$\boldsymbol{r}_S = [\mathcal{R}(s_1) \ldots \mathcal{R}(s_n)]^{\mathrm{T}} = [\mathcal{R}_1 \ldots \mathcal{R}_n]^{\mathrm{T}}$$

for a finite number of n states with state-values

$$\boldsymbol{v}_S = [v(s_1) \ldots v(s_n)]^{\mathrm{T}} = [v_1 \ldots v_n]^{\mathrm{T}}$$

one can derive a linear equation system based on the backup diagram (graph above):

$$\boldsymbol{v}_S = \boldsymbol{r}_S + \gamma \boldsymbol{\mathcal{P}}_{ss'} \boldsymbol{v}_S$$

$$\begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & & \vdots \\ p_{n1} & \cdots & p_{nn} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

This is a normal equation in $\boldsymbol{v}_S$:

$$\boldsymbol{v}_S = \boldsymbol{r}_S + \gamma \boldsymbol{\mathcal{P}}_{ss'} \boldsymbol{v}_S$$

$$\Leftrightarrow \underbrace{(\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}_{ss'})}_{\boldsymbol{A}} \underbrace{\boldsymbol{v}_S}_{\boldsymbol{s}} = \underbrace{\boldsymbol{r}_S}_{\boldsymbol{b}}$$

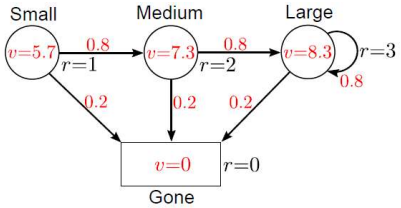
where $\boldsymbol{I}$ is the identity matrix (similar operation to $1 - x$). This equation can then be solved:

- By direct inversion (Gaussian elimination, $\mathcal{O}(n^3)$)
- By Matrix decomposition (QR, Cholesky, etc., $\mathcal{O}(n^3)$)
- By Iterative solutions (E.g., Krylov-subspaces, often better than $\mathcal{O}(n^3)$)

❗ In RL identifying and solving above equation is a key task, which is often realized only approximately for high-order state spaces. It is difficult to do directly!

Example

- MRP with state values
- Discount factor $\gamma = 0.8$
- Disaster probability $\alpha = 0.2$



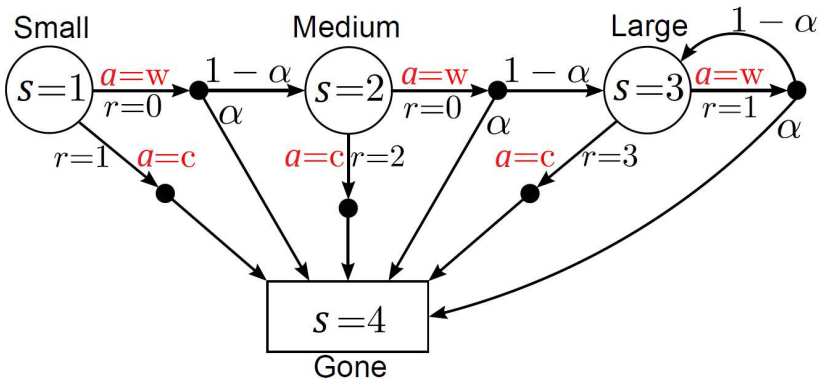
2.3 Finite Markov Decision Processes (MDP)

A *finite Markov decision process* (MDP) is a tuple $\langle \mathcal{S}, \mathcal{A}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$ with

- S being a finite set of discrete-time states $S_k \in \mathcal{S} \rightarrow$ state space
- $\mathcal{A}$ as a finite set of discrete-time actions $A_k \in \mathcal{A} \rightarrow$ action space
- $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{P}}_{ss'}^{a} = \mathbb{P}[S_{k+1} = s' | S_k = s, A_k = a_k]$ is the state transition probability; these are now multiple matrices (as opposed to MRPs), one for each possible action
- $\mathcal{R}$ is a reward function $\mathcal{R} = \mathcal{R}_s^a = \mathbb{E}[R_{k+1} | S_k = s, A_k = a_k]$
- γ is the discount factor $\gamma \in \{\mathbb{R} | 0 \leq \gamma \leq 1\}$

This extends a Markov Reward Process with actions/decisions. Now, rewards also depend on action A_k! MDPs can be transformed to MRPs if the policy is/actions are fixed.

Example



Two actions possible in each state:

- $a = w$ wait and let the tree grow
- $a = c$ cut it and gather the wood

With increasing tree size the wood reward increases as well!

The state transition probability matrix and reward function are given as:

$$\boldsymbol{\mathcal{P}}_{ss'}^{a=c} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{\mathcal{P}}_{ss'}^{a=w} = \begin{bmatrix} 0 & 1-\alpha & 0 & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$r_{\mathcal{S}}^{a=c} = \begin{bmatrix} 1 & 2 & 3 & 0 \end{bmatrix}^{\mathsf{T}}, \quad r_{\mathcal{S}}^{a=w} = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^{\mathsf{T}}.$$

The term $r_{\mathcal{S}}^{a}$ is the abbreviated form for receiving the output of $\mathcal{R}$ for the entire state space $\mathcal{S}$ given the action a.

2.3.1 Policies

In an MDP environment, a *policy* is a distribution over actions given states:

$$\pi(a_k | s_k) = \mathbb{P}[A_k = a_k | S_k = s_k]$$

In MDPs, policies depend only on the current state. A policy fully defines the agent's behavior.

Given an MDP $\langle \mathcal{S}, \mathcal{A}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$ and a policy π:

- The state sequence $S_k, S_{k+1}, \dots$ is a Markov chain $\langle \mathcal{S}, \boldsymbol{\mathcal{P}}^{\pi} \rangle$ since the state transition probability is only depending on the state:

$$\boldsymbol{\mathcal{P}}^{\pi}_{ss'} = \sum_{a_k \in \mathcal{A}} \pi(a_k|s_k) \boldsymbol{\mathcal{P}}^{a}_{ss'}$$

Conditionalizing on π, hence, summing out all actions.

- Consequently, the sequence $S_k, R_{k+1}, S_{k+1}, R_{k+2}, \dots$ of states and rewards is a Markov reward process $\langle \mathcal{S}, \boldsymbol{\mathcal{P}}^{\pi}, \mathcal{R}^{\pi}, \gamma \rangle$:

$$\mathcal{R}^{\pi}_{ss'} = \sum_{a_k \in \mathcal{A}} \pi(a_k|s_k) \mathcal{R}^{a}_{s}$$

Again, conditionalizing on π, hence, summing out all actions.

2.3.2 Value Functions

Almost all reinforcement learning algorithms involve estimating value functions—functions of states $v_{\pi}(s_k)$ (or of state–action pairs $q_{\pi}(s_k, a_k)$) that estimate *how good it is for the agent to be in a given state (*or how good it is to perform a given action in a given state*).*

Of course, the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to policies.

The *state-value function* of an MDP is the expected return starting in state s_k following policy π:

$$v_{\pi}(s_k) = \mathbb{E}_{\pi}[G_k|S_k = s_k] = \mathbb{E}_{\pi}\left[\sum_{i=0}^{\infty} \gamma^i R_{k+i+1} \,\middle|\, S_k\right]$$

The *action-value function* of an MDP is the expected return starting in state s_k taking an action a_k following a policy π:

$$q_{\pi}(s_k, a_k) = \mathbb{E}_{\pi}[G_k|S_k = s_k, A_k = a_k] = \mathbb{E}_{\pi}\left[\sum_{i=0}^{\infty} \gamma^i R_{k+i+1} \,\middle|\, S_k, A_k\right]$$

This function determines the value of choosing a single action a_k in the specific state s_k and the "just going back to normal" and following the instructions provided by the policy π.

2.3.3 Bellmann Expectation Equation

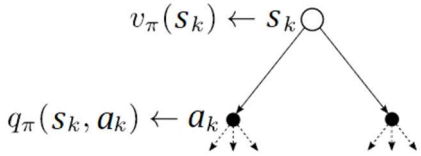
Analog to *2.2.2 Bellman-Equation for MRPs*, the state value of an MDP can be decomposed into a Bellman notation:

$$\begin{aligned} v_{\pi}(s_k) &= \mathbb{E}_{\pi}[G_k|S_k = s_k] \\ &= \mathbb{E}_{\pi}[R_{k+1} + \gamma G_{k+1}|S_k = s_k] \\ &= \mathbb{E}_{\pi}[R_{k+1} + \gamma v_{\pi}(S_{k+1})|S_k = s_k] \end{aligned}$$

This is done by using the definition: $g_k = r_{k+1} + \gamma g_{k+1}$ (see: *1.3 Return*).

In finite MPDs the state value can be directly linked to the action value by summing out the actions weighted by their transition probability $\pi(a_k|s_k)$.

$$v_\pi(s_k) = \sum_{a_k \in \mathcal{A}} \pi(a_k|s_k) * q_\pi(s_k, a_k)$$



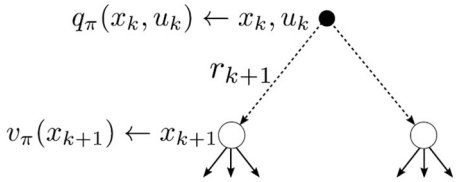
Backup diagram for $v_\pi(s_k)$

Likewise, the action value of an MDP can be decomposed into a Bellman notation:

$$\begin{aligned} q_\pi(s_k, a_k) &= \mathbb{E}_\pi[G_k | S_k = s_k, A_k = a_k] \\ &= \mathbb{E}_\pi[R_{k+1} + \gamma q_\pi(S_{k+1}, A_{k+1}) | S_k = s_k, A_k = a_k] \end{aligned}$$

In finite MPDs the action value can be directly linked to the state value by summing out the possible future states weighted by the transition probability $p^a_{ss'}$.

$$q_\pi(s_k, a_k) = \mathcal{R}^a_s + \gamma \sum_{s_{k+1} \in \mathcal{S}} p^a_{ss'} * v_\pi(s_{k+1})$$

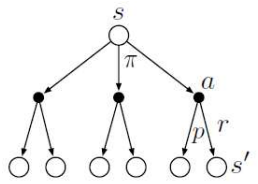


Backup diagram for $q_\pi(s_k, a_k)$

Inserting the summed-out q-/v-formulas above into each other directly results in the **Bellman Expectation Equation**:

$$v_\pi(s_k) = \sum_{a_k \in \mathcal{A}} \pi(a_k|s_k) * \left(\mathcal{R}^a_s + \gamma \sum_{s_{k+1} \in \mathcal{S}} p^a_{ss'} * v_\pi(s_{k+1}) \right)$$

It expresses a relationship between the value of a state and the values of its successor states. Think of looking ahead from a state to its possible successor states, as suggested by the diagram to the right. The Bellman equation averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.



Backup diagram for v_π

And, conversely, the action value becomes:

$$q_\pi(s_k, a_k) = \mathcal{R}^a_s + \gamma \sum_{s_{k+1} \in \mathcal{S}} p^a_{ss'} * \left(\sum_{a_k \in \mathcal{A}} \pi(a_{k+1}|s_{k+1}) * q_\pi(s_{k+1}, a_{k+1}) \right)$$

2.3.4 Bellman Expectation Equation in Matrix Form

Given a policy π and following the same assumptions as for *2.2.2 Bellman-Equation for MRPs*, the Bellman expectation equation can be expressed in Matrix form:

$$\boldsymbol{v}^\pi_S = \boldsymbol{r}^\pi_S + \gamma \boldsymbol{\mathcal{P}}^\pi_{ss'} \boldsymbol{v}^\pi_S$$

$$\begin{bmatrix} v_1^\pi \\ \vdots \\ v_n^\pi \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1^\pi \\ \vdots \\ \mathcal{R}_n^\pi \end{bmatrix} + \gamma \begin{bmatrix} p_{11}^\pi & \dots & p_{1n}^\pi \\ \vdots & & \vdots \\ p_{n1}^\pi & \dots & p_{nn}^\pi \end{bmatrix} \begin{bmatrix} v_1^\pi \\ \vdots \\ v_n^\pi \end{bmatrix}$$

Here, $\boldsymbol{r}_S^\pi$ and $\boldsymbol{\mathcal{P}}_{ss'}^\pi$ are the rewards and state transition probability following policy π. Hence, the state value can be calculated by solving above matrix for $\boldsymbol{v}_S^\pi$, e.g., by direct matrix inversion:

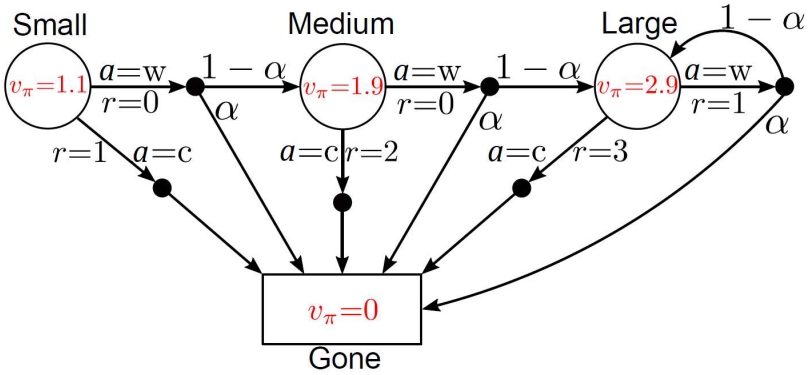
$$\boldsymbol{v}_S^\pi = \left(\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}_{ss'}^\pi\right)^{-1} * \boldsymbol{r}_S^\pi$$

Example

Let's assume following very simple policy ("50/50")

$$\pi(a = cut|s) = 0.5, \qquad \pi(a = wait|s) = 0.5 \; \forall s \in S$$

- Discount factor $\gamma = 0.8$
- Disaster probability $\alpha = 0.2$



Forest MDP with 50/50-policy including state values

The 50/50-policy applied to the given environment behavior,

$$\boldsymbol{\mathcal{P}}_{ss'}^{a=c} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{\mathcal{P}}_{ss'}^{a=w} = \begin{bmatrix} 0 & 1-\alpha & 0 & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$r_S^{a=c} = \begin{bmatrix} 1 & 2 & 3 & 0 \end{bmatrix}^\mathsf{T}, \quad r_S^{a=w} = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^\mathsf{T}.$$

results in:

$$\boldsymbol{\mathcal{P}}_{ss'}^\pi = \begin{bmatrix} 0 & \frac{1-\alpha}{2} & 0 & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{r}_S^\pi = \begin{bmatrix} 0.5 \\ 1 \\ 2 \\ 0 \end{bmatrix}$$

2.4 Optimal Policies and Value Functions

The optimal state-value function of an MDP is the maximum state-value function over all polices:

$$v^*(s) = \max_\pi v_\pi(s)$$

The optimal action-value function of an MDP is the maximum action-value function over all polices:

$$q^*(s,a) = \max_\pi q_\pi(s,a)$$

The optimal value function denotes the best possible agent's performance for a given MDP/environment. A finite MDP can be easily solved in an optimal way if $q^*(s,a)$ is known.

A policy π is defined to be *better than or equal to* a policy π', if its expected return is greater than or equal to that of π' for all states:

$$\pi \geq \pi' \text{ if: } \quad v_\pi(s) \geq v_{\pi'}(s)\ \forall s \in \mathcal{S}$$

For any finite MDP

- there exists an optimal policy $\pi^* \geq \pi$ that is better or equal to all other policies
- all optimal policies achieve the same optimal state-value function $v^*(s) = v_{\pi^*}(s)$
- all optimal policies achieve the same optimal action-value function $q^*(s,a) = q_{\pi^*}(s,a)$

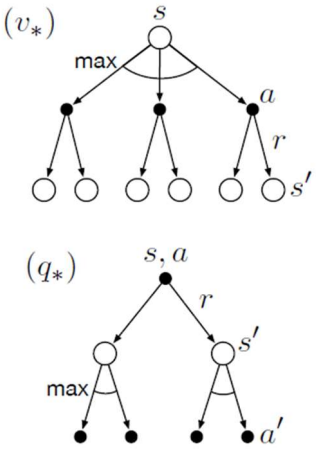
Principle of Optimality: "*An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*" - R.E. Bellman (Dynamic Programming, 1957)

The Bellman optimality equation for a finite MDP results in:

$$v^*(s_k) = \max_{a_k} \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a v_{\pi^*}(s_{k+1})$$

and for the action value:

$$q^*(s_k, a_k) = \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a * \max_{a_{k+1}} q^*(s_{k+1}, a_{k+1})$$



Backup diagrams for optimality equations

In finite MDPs with n states, above equations deliver an algebraic equation system with many unknowns and state-value equations. Due to max operator the equations sets are generally nonlinear. Direct, closed form solution rarely available. Hence, often approximate / iterative solutions are required.

If environment is exactly known, solving for v^* or q^* directly delivers optimal policy.

- If $v(s)$ is known, a one-step-ahead search is required to get $q(s)$.
- If $q(s,a)$ is known, directly choose q^*

Even though above decisions are very short sighted, by Bellman's principle of optimality one receives the long-term maximum of the expected reward.

2.5 Direct Numerical Value Calculation

This is possible only for small action- and state-space MDPs! RL addresses mainly two topics:

- Approximate solutions of complex decision problems

- Learning of such approximations based on environment interactions

2.6 Formula Overview

	Bellman Expectation Equation	Bellman Optimality Equation
state-value-function	$v_\pi(s_k) = \sum_{a_k \in \mathcal{A}} \pi(a_k \mid s_k) * \left(\mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a * v_\pi(s_{k+1}) \right)$	$v^*(s_k) = \max_{a_k} \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a v_{\pi^*}(s_{k+1})$
action-value-function	$q_\pi(s_k, a_k) = \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a * \left(\sum_{a_k \in \mathcal{A}} \pi(a_{k+1} \mid s_{k+1}) * q_\pi(s_{k+1}, a_{k+1}) \right)$	$q^*(s_k, a_k) = \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a * \max_{a_{k+1}} q^*(s_{k+1}, a_{k+1})$

3 Dynamic Programming

Dynamic programming (DP) is both a mathematical optimization method and a computer programming method. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. Likewise, in computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

Dynamic programming is a collection of algorithms to solve MDPs and neighboring problems. DP can be applied to problems with the following characteristics:

- Optimal substructure: Optimal solution can be derived from subproblems
- Overlapping subproblems: Subproblems may occur many times, hence, solutions can be cached and reused.

💡 This is connected to MDPs as they satisfy above properties. The Bellman equation provides recursive decomposition, and the value function stores and reuses solutions. DP is used for iterative planning in an MDP.

📄 Planning consists of two problems:

- Prediction
  - Input: MDP *and* policy π
  - Output: estimated value function $\hat{v}_\pi \approx v_\pi$ of given policy π
- Control
  - Input: MPD
  - Output: estimated optimal value function $\hat{v}_\pi^* \approx v_\pi^*$ or policy $\hat{\pi}^* \approx \hat{\pi}$

❗ In both applications DP requires *full knowledge* of the MDP structure. The feasibility in real-word engineering applications is therefore limited! But the following concepts are largely used in modern data-driven RL algorithms.

3.1 Policy Evaluation

Problem: *evaluate a given policy π to predict* v_π. Directly calculating v_π is numerically costly for high-dimensional state spaces (e.g., by matrix inversion).

General idea: apply *iterative approximations* $\hat{v}_i(s_k) = v_i(s_k)$ of $v_\pi(s_k)$ with decreasing errors.

$$\|v_i(s_k) - v_\pi\|_\infty \to 0 \text{ for } i = 1, 2, 3, \dots$$

In the MDP context, the Richardson iteration became the default solution approach to iteratively solve the Bellman equation:

$$\underbrace{(\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}^{\pi}_{ss'})}_{\boldsymbol{A}} \underbrace{\boldsymbol{v}^{\pi}_{\mathcal{S}}}_{\boldsymbol{\zeta}} = \underbrace{\boldsymbol{r}^{\pi}_{\mathcal{S}}}_{\boldsymbol{b}}$$

Bellman equation in matrix form

$$\boldsymbol{\zeta}_{i+1} = \boldsymbol{\zeta}_i + \omega(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{\zeta}_i)$$

which translates to:

$$\boldsymbol{v}^{\pi}_{\mathcal{S}\,i+1} = \boldsymbol{v}^{\pi}_{\mathcal{S}\,i} + \omega\left(\boldsymbol{r}^{\pi}_{\mathcal{S}} - \left(\left(\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}^{\pi}_{ss'}\right) * \boldsymbol{v}^{\pi}_{\mathcal{S}\,i}\right)\right)$$

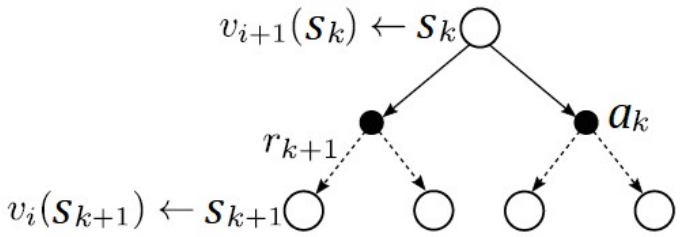
with ω being a scalar parameter that has to be chosen such that the sequence $\boldsymbol{\zeta}_i$ converges (learning rate; step-size.

3.1.1 Richardson Iteration

The general form for any $s_k \in \mathcal{S}$ at iteration i is given as:

$$v_{i+1}(s_k) = \sum_{a_k \in \mathcal{A}} \pi(a_k|s_k) * \left(\mathcal{R}^a_s + \gamma \sum_{s_{k+1} \in \mathcal{S}} p^a_{ss'} * v_i(s_{k+1}) \right)$$

The matrix form then is: $\boldsymbol{v}^{\pi}_{S,i+1} = \boldsymbol{r}^{\pi}_{S} + \gamma \boldsymbol{\mathcal{P}}^{\pi}_{ss'} \boldsymbol{v}^{\pi}_{S,i}$

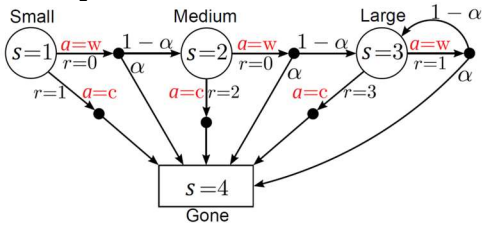


Backup diagram for iterative policy evaluation

During one Richardson iteration we update $v_{i+1}(s_k)$ from $v_i(s_{k+1})$. The old value of s_k, $v_i(s_k)$ is replaced with a new value $v_{i+1}(s_k)$ from the old values of the successor state $v_i(s_{k+1})$.

Updating estimates (v_{i+1}) on the basis of other estimates (v_i) is often called bootstrapping. Its also called the expected update because it is based on the expectation over all possible next states (utilizing full knowledge). In subsequent lectures, the expected update will be supplemented by data-driven samples from the environment.

Example



$$\mathcal{P}^{\pi}_{ss'} = \begin{bmatrix} 0 & \frac{1-\alpha}{2} & 0 & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad r^{\pi}_{\mathcal{S}} = \begin{bmatrix} 0.5 \\ 1 \\ 2 \\ 0 \end{bmatrix}$$

$$v_{i+1}(s_k) = \sum_{a_k \in \mathcal{A}} \pi(a_k|s_k) * \left(\mathcal{R}^a_s + \gamma \sum_{s_{k+1} \in \mathcal{S}} p^a_{ss'} * v_i(s_{k+1}) \right)$$

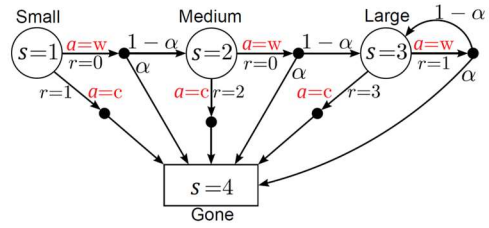
$\alpha = 0.2$ and $\gamma = 0.8$

i	$v_i(s=1)$	$v_i(s=2)$	$v_i(s=3)$	$v_i(s=4)$
0	0	0	0	0
1	$0.5 * (0 + 0.8 * (0.8 * 0 + 0.2 * 0)) + 0.5 * (1 + 0.8 * (1 * 0)) = \mathbf{0.5}$	$0.5 * (0 + 0.8 * (0.8 * 0 + 0.2 * 0)) + 0.5 * (2 + 0.8 * (1 * 0)) = \mathbf{1}$	$0.5 * (1 + 0.8 * (0.8 * 0 + 0.2 * 0)) + 0.5 * (3 + 0.8 * (1 * 0)) = \mathbf{2}$	0
2	$0.5 * (0 + 0.8 * (0.8 * 1 + 0.2 * 0)) + 0.5 * (1 + 0.8 * (1 * 0)) = \mathbf{0.82}$	$0.5 * (0 + 0.8 * (0.8 * 2 + 0.2 * 0)) + 0.5 * (2 + 0.8 * (1 * 0)) = \mathbf{1.64}$	$0.5 * (1 + 0.8 * (0.8 * 2 + 0.2 * 0)) + 0.5 * (3 + 0.8 * (1 * 0)) = \mathbf{2.64}$	0
3	$0.5 * (0 + 0.8 * (0.8 * 1.64 + 0.2 * 0)) + 0.5 * (1 + 0.8 * (1 * 0)) = \mathbf{1.03}$	$0.5 * (0 + 0.8 * (0.8 * 2.64 + 0.2 * 0)) + 0.5 * (2 + 0.8 * (1 * 0)) = \mathbf{1.85}$	→ use state values from **previous** row ...	0
...	...	...	...	...
∞	**1.12**	**1.94**	**2.94**	0

3.1.2 Variant: In-Place Updates

Instead of applying $(\boldsymbol{v}^{\pi}_{S,i+1} = \boldsymbol{r}^{\pi}_{S} + \gamma \boldsymbol{\mathcal{P}}^{\pi}_{ss'} \boldsymbol{v}^{\pi}_{S,i})$ to the entire vector $\boldsymbol{v}^{\pi}_{S,i+1}$ in 'one shot' (synchronous backup), an elementwise in-place version of the policy evaluation can be carried out by allowing to update states in a beneficial order. This may converge faster than regular Richardson iteration if state update order is chosen wisely (sweep through state space).

Example



$$\mathcal{P}^{\pi}_{ss'} = \begin{bmatrix} 0 & \frac{1-\alpha}{2} & 0 & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad r^{\pi}_{\mathcal{S}} = \begin{bmatrix} 0.5 \\ 1 \\ 2 \\ 0 \end{bmatrix}$$

$$v_{i+1}(s_k) = \sum_{a_k \in \mathcal{A}} \pi(a_k|s_k) * \left(\mathcal{R}^a_s + \gamma \sum_{s_{k+1} \in \mathcal{S}} p^a_{ss'} * v_i(s_{k+1}) \right)$$

$\alpha = 0.2$ and $\gamma = 0.8$

Reverse order, i.e., start with $s = 4$:

i	$v_i(s=1)$	$v_i(s=2)$	$v_i(s=3)$	$v_i(s=4)$
0	0	0	0	0
1	$0.5 * (0 + 0.8 * (0.8 * 1.64 + 0.2 * 0)) + 0.5 * (1 + 0.8 * (1 * 0)) = \mathbf{1.03}$	$0.5 * (0 + 0.8 * (0.8 * 2 + 0.2 * 0)) + 0.5 * (2 + 0.8 * (1 * 0)) = 1.64$	$0.5 * (1 + 0.8 * (0.8 * 0 + 0.2 * 0)) + 0.5 * (3 + 0.8 * (1 * 0)) = 2$	0
2	... ← use state values from **same** row	$0.5 * (0 + 0.8 * (0.8 * 2.64 + 0.2 * 0)) + 0.5 * (2 + 0.8 * (1 * 0)) = 1.85$	$0.5 * (1 + 0.8 * (0.8 * 2 + 0.2 * 0)) + 0.5 * (3 + 0.8 * (1 * 0)) = 2.64$	0
...	...	...	...	...
∞	**1.12**	**1.94**	**2.94**	0

3.2 Policy Improvement

❓ If we know v_π of a given MDP, how can we improve the policy?

- Consider a new (non-policy-conform) action $a \neq \pi(s_k)$
- Follow the current policy π thereafter
- Check the action-value of the "new move"

$$q_\pi(s_k, a_k) = \mathbb{E}[R_{k+1} + \gamma v_\pi(S_{k+1}) | S_k = s_k, A_k = a_k]$$

The key criterion is whether this is greater than or less than $v_\pi(s)$. If it is greater — that is, if it is better to select action a once in state s and thereafter follow π than it would be to follow π all the time — then one would expect it to be better still to select a every time state s is encountered, and that the new policy would in fact be a better one overall.

Theorem

If for any deterministic policy pair π and π':

$$q_\pi\big(s, \pi'(s)\big) \geq v_\pi(s) \qquad \forall s \in \mathcal{S}$$

applies, then the policy π' must be as good as or better than π. Hence, it obtains greater or equal expected return:

$$v_{\pi'}(s) \geq v_\pi(s) \qquad \forall s \in \mathcal{S}$$

Greedy Policy Improvement

So far, policy improvement addressed only changing the policy at a single state. Now, we extend this scheme to all states by selecting the best action according to $q_\pi(s_k, a_k)$ in every state:

$$\begin{aligned}\pi'(s_k) &= \arg\max_{a_k \in \mathcal{A}} q_\pi(s_k, a_k) \\ &= \arg\max_{a_k \in \mathcal{A}} \mathbb{E}[R_{k+1} + \gamma v_\pi(S_{k+1}) | S_k = s_k, A_k = a_k] \\ &= \arg\max_{a_k \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a * v_\pi(s_{k+1})\end{aligned}$$

Each greedy policy improvement takes the best action in a one-step look-ahead search and, therefore, satisfies above Theorem. If after a policy improvement step $v_{\pi'}(s) = v_\pi(s)$ it follows:

$$v_{\pi'}(s_k) = \max_{a_k \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a * v_\pi(s_{k+1})$$

which is the Bellman optimality equation. This guarantees that $\pi' = \pi$ must be optimal policies π^*. Although proof for policy improvement theorem was presented for deterministic policies, transfer to stochastic policies $\pi(a_k|s_k)$ is possible.

💡 The policy improvement theorem guarantees finding optimal policies in finite MDPs.

3.2.1 Policy Iteration

Policy Iteration combines *Policy Evaluation* and *3.2Policy Improvement* in an iterative seq.:

$$\pi_0 \rightarrow v_{\pi_0} \rightarrow \pi_1 \rightarrow v_{\pi_1} \rightarrow \cdots \rightarrow \pi^* \rightarrow v_{\pi^*}$$

$$startingPolicy \rightarrow evaluate \rightarrow improve \rightarrow evaluate \rightarrow improve \rightarrow \cdots$$

In the 'classic' policy iteration, each policy evaluation step is fully executed, i.e., for each policy π_i an exact estimate of v_{π_i} is provided either by iterative policy evaluation with a sufficiently high number of steps or by any other method that fully solves the Bellman equation.

Example: Tree Hater Policy

1. Assume $\alpha = 0.2$ and $\gamma = 0.8$ and start with tree hater initial policy:
$$\pi_0 = \pi(cut|s_k)\ \forall s_k \in S$$
2. First policy evaluation resulted in: $v_S^{\pi_0} = [1 \quad 2 \quad 3 \quad 0]^T$
3. Perform first greedy policy improvement (one-step lookahead):
$$\pi_1 = \{\pi(w|s_k = 1), \pi(c|s_k = 2), \pi(c|s_k = 3)\}$$
→ compared to π_0: action for state 1 changed
4. Second policy evaluation resulted in: $v_S^{\pi_1} = [1.28 \quad 2 \quad 3 \quad 0]^T$
5. Perform second greedy policy improvement (one-step lookahead):
$$\pi_2 = \{\pi(w|s_k = 1), \pi(c|s_k = 2), \pi(c|s_k = 3)\}$$
→ compared to π_1: nothing changed
6. Based on the policy improvement theorem we've found our best policy π′; which is for α = 0.2 and γ = 0.8 the optimal policy π*.

Pseudocode:

```
for s in states:
    for a in actions:
        if q(s,a) > π'[s,a]
            π'[s,a] = q(s,a)
```

Formula - Greedy Policy Improvement:

$$\pi'(s_k) = \underset{a_k \in \mathcal{A}}{arg\,max}\, q_\pi(s_k, a_k)$$
$$= \underset{a_k \in \mathcal{A}}{arg\,max}\, \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in S} p_{ss'}^a * v_\pi(s_{k+1})$$

Example: use $v_\pi(s_{k+1})$ from the previous evaluation

action / state	$a = cut$	$a = wait$
$s = 1$	$1 + 0.8 * (1 * 0) = 1$	$0 + 0.8 * (0.8 * 2 + 0.2 * 0) = \mathbf{1,28}$
$s = 2$	$2 + 0.8 * (1 * 0) = \mathbf{2}$	$0 + 0.8 * (0.8 * 3 + 0.2 * 0) = 1,92$
$s = 3$	$3 + 0.8 * (1 * 0) = \mathbf{3}$	$1 + 0.8 * (0.8 * 3 + 0.2 * 0) = 2,92$
$s = 4$	0	0

Since a new action is better than the old one:
→ update the next policy accordingly.

Example: Tree Lover Policy

1. Assume $\alpha = 0.2$ and $\gamma = 0.8$ and start with tree lover initial policy:
$$\pi_0 = \pi(wait|s_k)\ \forall s_k \in S$$
2. First policy evaluation resulted in: $v_S^{\pi_0} = [1.14 \quad 1.78 \quad 2.78 \quad 0]^T$
3. Perform first greedy policy improvement (one-step lookahead):
$$\pi_1 = \{\pi(w|s_k = 1), \pi(w|s_k = 2), \pi(c|s_k = 3)\}$$
→ compared to π_0: actions for states 2 and 3 changed
4. Second policy evaluation resulted in: $v_S^{\pi_1} = [1.28 \quad 2 \quad 3 \quad 0]^T$
5. Perform second greedy policy improvement (one-step lookahead):
$$\pi_2 = \{\pi(w|s_k = 1), \pi(c|s_k = 2), \pi(c|s_k = 3)\}$$
→ compared to π_1: nothing changed
6. Based on the policy improvement theorem we've found our best policy π′; which is for α = 0.2 and γ = 0.8 the optimal policy π*.

action / state	$a = cut$	$a = wait$
$s = 1$	$1 + 0.8 * (1 * 0) = 1$	$0 + 0.8 * (0.8 * 1.78 + 0.2 * 0) = \mathbf{1,14}$
$s = 2$	$2 + 0.8 * (1 * 0) = \mathbf{2}$	$0 + 0.8 * (0.8 * 2.78 + 0.2 * 0) = 1,78$
$s = 3$	$3 + 0.8 * (1 * 0) = \mathbf{3}$	$1 + 0.8 * (0.8 * 2.78 + 0.2 * 0) = 2,78$
$s = 4$	0	0

Since several actions are better than the old ones:
→ update the next policy accordingly.

action / state	$a = cut$	$a = wait$
$s = 1$	$1 + 0.8 * (1 * 0) = 1$	$0 + 0.8 * (0.8 * 2 + 0.2 * 0) = \mathbf{1,28}$
$s = 2$	$2 + 0.8 * (1 * 0) = \mathbf{2}$	$0 + 0.8 * (0.8 * 3 + 0.2 * 0) = 1,92$
$s = 3$	$3 + 0.8 * (1 * 0) = \mathbf{3}$	$1 + 0.8 * (0.8 * 3 + 0.2 * 0) = 2,92$
$s = 4$	0	0

No actions have changed from previous policy:
→ optimal policy has been found.

Regardless of the initial policy, this will always converge to optimal policy π^* given the chosen values for α and γ.

3.2.2 Value Iteration

Policy iteration involves full policy evaluation steps between policy improvements. In large state-space MDPs the full policy evaluation may be numerically very costly. Using a limited number of iterative policy evaluations steps and then apply policy improvement may speed up the entire DP process.

Value iteration is a one-step iterative policy evaluation followed by policy improvement. Allows simple update rule which combines policy improvement with truncated policy evaluation:

$$v_{i+1}(s_k) = \max_{a_k \in \mathcal{A}} \mathbb{E}[R_{k+1} + \gamma v_i(S_{k+1}) | S_k = s_k, A_k = a_k]$$

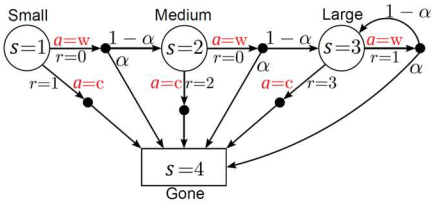
$$= \max_{a_k \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a v_i(s_{k+1})$$

Algorithm

input: full model of the MDP, i.e., $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
parameter: $\delta > 0$ as accuracy termination threshold
init: $v_0(s_k) \, \forall \, x \in \mathcal{S}$ arbitrary except $v_0(s_k) = 0$ if x is terminal
repeat
  $\Delta \leftarrow 0$;
  for $\forall \, s_k \in \mathcal{S}$ **do**
    $\tilde{v} \leftarrow \hat{v}(s_k)$;
    $\hat{v}(s_k) \leftarrow \max_{a_k \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a \hat{v}(s_{k+1}) \right)$;
    $\Delta \leftarrow \max(\Delta, |\tilde{v} - \hat{v}(s_k)|)$;
until $\Delta < \delta$;
output: Deterministic policy $\pi \approx \pi^*$, such that
$\pi(s_k) \leftarrow \arg\max_{a_k \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a \hat{v}(s_{k+1}) \right)$;

Compared to policy iteration, value iteration does not require an initial policy but only a state-value guess.

Example



$$\mathcal{P}^{\pi}_{ss'} = \begin{bmatrix} 0 & \frac{1-\alpha}{2} & 0 & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad r^{\pi}_{\mathcal{S}} = \begin{bmatrix} 0.5 \\ 1 \\ 2 \\ 0 \end{bmatrix}$$

$$v_{i+1}(s_k) = \max_{a_k \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a * v_i(s_{k+1})$$

$\alpha = 0.2$ and $\gamma = 0.8$

Like in-place update policy evaluation, reverse order and start value iteration with $s = 4$:

i	$v_i(s=1)$	$v_i(s=2)$	$v_i(s=3)$	$v_i(s=4)$
0	0	0	0	0
1	$\max_{a_k \in \{w,c\}} \begin{Bmatrix} 0 + 0.8 * (0.8 * 2 + 0.2 * 0) \\ 1 + 0.8 * (1 * 0) \end{Bmatrix} = \max_{a_k \in \{w,c\}} \begin{Bmatrix} 1.28 \\ 1 \end{Bmatrix} = \mathbf{1.28}$	$\max_{a_k \in \{w,c\}} \begin{Bmatrix} 0 + 0.8 * (0.8 * 3 + 0.2 * 0) \\ 2 + 0.8 * (1 * 0) \end{Bmatrix} = \max_{a_k \in \{w,c\}} \begin{Bmatrix} 1.92 \\ 2 \end{Bmatrix} = 2$	$\max_{a_k \in \{w,c\}} \begin{Bmatrix} 1 + 0.8 * (0.8 * 0 + 0.2 * 0) \\ 3 + 0.8 * (1 * 0) \end{Bmatrix} = \max_{a_k \in \{w,c\}} \begin{Bmatrix} 1 \\ 3 \end{Bmatrix} = 3$	0
2	... use state values from **same** row	$\max_{a_k \in \{w,c\}} \begin{Bmatrix} 0 + 0.8 * (0.8 * 3 + 0.2 * 0) \\ 2 + 0.8 * (1 * 0) \end{Bmatrix} = \max_{a_k \in \{w,c\}} \begin{Bmatrix} 1.92 \\ 2 \end{Bmatrix} = \mathbf{2}$	$\max_{a_k \in \{w,c\}} \begin{Bmatrix} 1 + 0.8 * (0.8 * 3 + 0.2 * 0) \\ 3 + 0.8 * (1 * 0) \end{Bmatrix} = \max_{a_k \in \{w,c\}} \begin{Bmatrix} 2.92 \\ 3 \end{Bmatrix} = 3$	0
...	...	...	...	...
∞	**1.28**	**2**	**3**	0

Choose policy based on actions connected to the *max*-values (after every calculation).

3.3 Further Aspects

All DP algorithms are based on the state-value $v(s)$.

- Complexity is $O(m * n^2)$ for m actions and n states.
- Evaluate all n^2 state transitions while considering up to m actions per state.

Could be also applied to action-values $q(s, a)$.

- Complexity is inferior with $O(m^2 * n^2)$.
- There are up to m^2 action-values which require n^2 state transition evaluations each.

Problem	Algorithm	Relevant Equations
Prediction	Policy Evaluation	Bellman Expectation
Control	Policy Iteration	Bellman Expectation
		Greedy Policy Improvement
	Value Iteration	Bellman Optimality

3.3.1 Asynchronous DP

❗ DP algorithms considered so far used *synchronous backups*. This means that in one iteration the entire state space is updated. This may be computational expensive for large MDPs. For some state-values or policy parts may converge faster than other but are updated as often as slowly converging states.

In contrast, *asynchronous backups* update states individually in an (arbitrary) order. Choosing a smart order achieves faster overall convergence rate. Simple example: in-place policy evaluation where only a subset of all states is updated in each iteration.

3.3.1.1 Prioritized Sweeping

This can be done by *prioritized sweeping*, where the state with the largest Bellman error gets updated first:

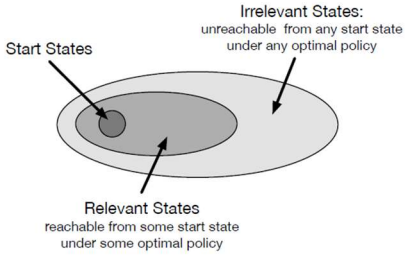
$$\arg\max_{s_k \in \mathcal{S}} \left| \max_{a_k \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a v_i(s_{k+1}) \right) - v_i(s_k) \right|$$

This builds up a priority queue of most relevant states by refreshing the Bellman error after each state update.

3.3.1.2 Real-Time Updates

Another approach would be to use *real-time updates* where those states which are frequently visited by the agent get updated. It utilizes the agent's experience to guide the asynchronous DP updates. After each time step $\langle s_k, a_k, r_{k+1} \rangle$ update s_k:

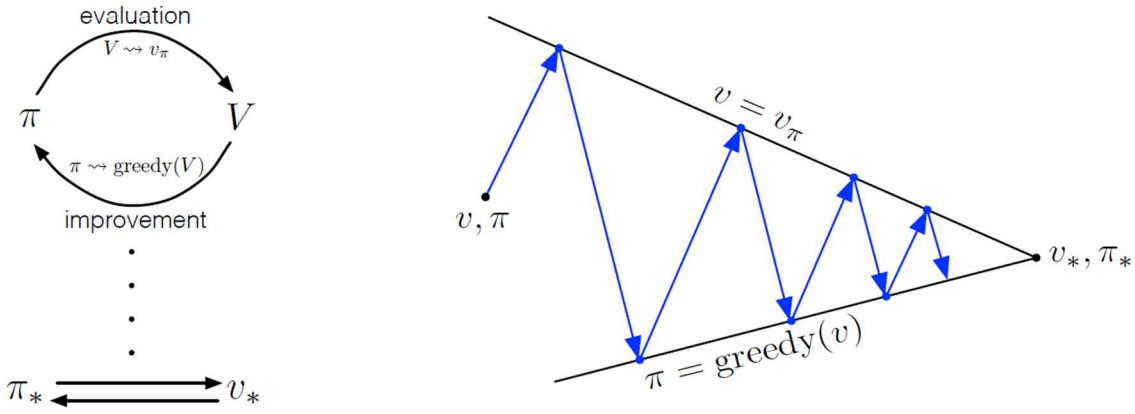
$$v_i(s_k) \leftarrow \max_{a_k \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a v_i(s_{k+1}) \right)$$



3.3.2 Generalized Policy Iteration (GPI)

The idea of updating the state values (i.e., through policy evaluation) followed by an improvement step is typically considered part of the GPI-Framework.

In a push-pull sort of manner were iteratively trying to make our state-value estimate v better and then improving the policy



Interpreting *Generalized Policy Iteration* to switch back and forth between (arbitrary) evaluations and improvement steps.

3.3.3 Curse of Dimensionality

DP is much more efficient than an exhaustive search (looking at all paths) over all n states and m actions in finite MDPs in order to find an optimal policy.

- Exhaustive search for deterministic policies: m^n evaluations (exponential)
- DP results in polynomial complexity regarding m and n

Nevertheless, DP uses full-width backups (for each state update, every successor state and action is considered). Hence, DP can be effective up to medium-sized MDPs (i.e., million states).

For large problems DP suffers from the *curse of dimensionality*:

- Number of finite states n grows exponentially with the number of state variables.
- Also: if continuous variables need quantization typically a large number of states results.
- Single state update may become computational infeasible.

4 Monte Carlo Methods

Dynamic Programming was *model-based* prediction and control. This means we assumed to have full knowledge of the MDP dynamics and structure.

Monte Carlo Methods allow model-free prediction and control. We therefore can estimate value functions and optimize policies in unknown MDPs. Although, we still assume to deal with finite MDP problems.

The key characteristic of MC methods is that the agents learn from *experience*, i.e., sequences of samples $\langle s_k, a_k, r_{k+1} \rangle$. The main concept is estimation by *averaging sample returns*.

The following chapter is limited to episodic tasks. As consequence we can estimate action/state values and update policies only in an episode-by-episode way.

4.1 Monte Carlo Prediction

We want to estimate a state value $v_\pi(s)$ for a given policy π. Available to us are samples $\langle s_{k,j}, a_{k,j}, r_{k+1,j} \rangle$ for episodes $j = 1, \dots, J$. To get the state value we can average the returns of visiting state s_k over all episodes:

$$v_\pi(s_k) \approx \hat{v}_\pi(s_k) = \frac{1}{J} \sum_{j=1}^{J} g_{k,j} = \frac{1}{J} \sum_{j=1}^{J} \sum_{i=1}^{T_j} \gamma^i * r_{k+i+1,j}$$

We sum over all episodes J, and for each episode over all steps of that episode. T_j denotes the terminating (last) time step of each episode j.

When iterating over different runs/episodes we can distinguish two flavors:

- *First-visit MC*: Apply above formula only to the first state visit per episode
- *Every-visit MC*: Apply above formula each time visiting a certain state per episode

Algorithm – First Visit

input: a policy π to be evaluated
output: estimate of $v_{\mathcal{S}}^{\pi}$ (i.e., value estimate for all states $s \in \mathcal{S}$)
init: $\hat{v}(s) \, \forall \, s \in \mathcal{S}$ arbitrary except $v_0(s) = 0$ if s is terminal
$l(s) \leftarrow$ an empty list for every $s \in \mathcal{S}$
for $j = 1, \dots, J$ *episodes* **do**
Generate an episode following π: $s_0, u_0, r_1, \dots, s_{T_j}, a_{T_j}, r_{T_j+1}$;
Set $g \leftarrow 0$;
for $k = T_j - 1, T_j - 2, T_j - 3, \dots, 0$ *time steps* **do**
$g \leftarrow \gamma g + r_{k+1}$;
if $s_k \notin \langle s_0, s_1, \dots, s_{k-1} \rangle$ **then**
Append g to list $l(s_k)$;
$\hat{v}(s_k) \leftarrow$ average($l(s_k)$);

We iterate over all episodes. For each episode, then iterate backwards over the time steps of current episode. For each time step, update the return. Then check whether the current state

we're in will also be visited earlier. If not, i.e., this is the first visit, append the return g to list $l(s)$ and calculate $\hat{v}(s_k)$ by averaging the returns in list $l(s)$.

❗ This algorithm is inefficient due to large memory demand (storing $l(s)$). To overcome this, we can use an incremental/recursive implementation.

4.1.1 Incremental Implementation

The sample mean $\mu_1, \mu_2, \ldots$ of an arbitrary sequence $g_1, g_2, \ldots$ is:

$$\begin{aligned}\mu_j &= \frac{1}{j}\sum_{i=1}^{j} g_i \\ &= \frac{1}{j}\left[g_j + \sum_{i=1}^{j-1} g_i\right] \\ &= \frac{1}{j}\left[g_j + (j-1)\mu_{j-1}\right] \\ &= \mu_{j-1} + \frac{1}{j}\left[g_j - \mu_{j-1}\right]\end{aligned}$$

With the resulting formula, we can calculate the new sample mean μ_j by updating the current sample mean μ_{j-1} with the newly observed return g_j. This formula does assume that the decision problem is stationary (μ_j is not changing over time). In case of a non-stationary problem, using a forgetting factor $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$ allows for dynamic adaptation:

$$\mu_j = \mu_{j-1} + \alpha\left[g_j - \mu_{j-1}\right]$$

This changes the weighting factor from $\frac{1}{j}$ (the number of episodes) to α.

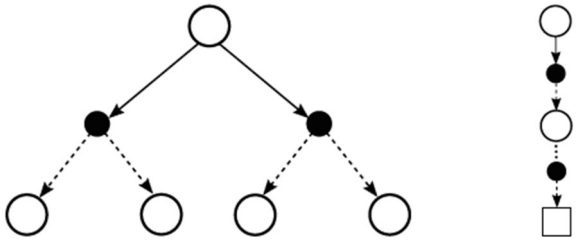
Statistical Properties

For the *first time visit* approach, each return sample g_j is independent from the others since they were drawn from separate episodes. One receives independent and identically distributed (i.i.d.) data to do this estimation. Consequently, this is bias-free. The estimates variance drops with $\frac{1}{n}$ where n are available samples.

For every *time visit* approach, each return sample g_j is *not* independent from the others since they might be obtained from same episodes. One receives non-i.i.d. data to estimate and consequently this is biased for any $n < \infty$. Only in the limit $n \to \infty$ one receives $(v_\pi(s) - \mathbb{E}[\hat{v}_\pi(s)]) \to 0$.

❗ This results in a bias-variance dilemma as for the *first-time* approach, bias is low, and variance is high. *Every-visit* MC is quicker to converge, as more states can be used, but has a certain bias.

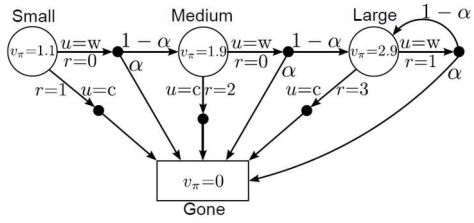
💡 MC is particularly attractive when one requires state-value knowledge of only one or few states. This is because one estimate does not rely on the estimate of other states (no bootstrapping as in DP).

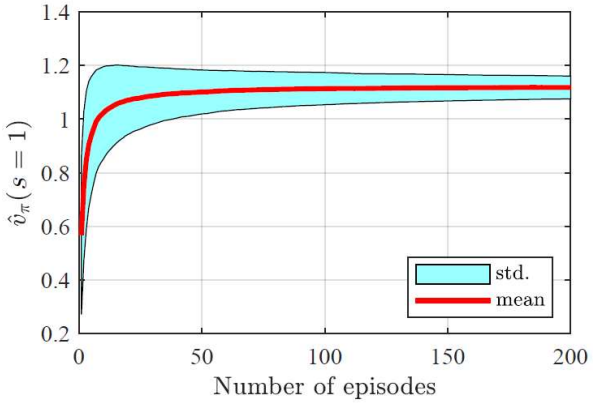


Backup diagram for DP (left) and MC (right) prediction

4.1.2 Example

Let's reuse the forest tree MDP example with *fifty-fifty policy* and discount factor $\gamma = 0.8$ plus disaster probability $\alpha = 0.2$:





State-value estimate of forest tree MDP for state $s = 1$ using MC-based prediction over the number of episodes being evaluated.

Estimation of Action Values

Action values are very useful to directly obtain optimal choices. Analog to above algorithm we can estimate $q_\pi(s, a)$ with a small extension: visits refer to state-action pairs (s, a). First-visit and every-visit variants also exist. Possible problems can arise because certain state-action combinations may never be visited when following a deterministic policy.

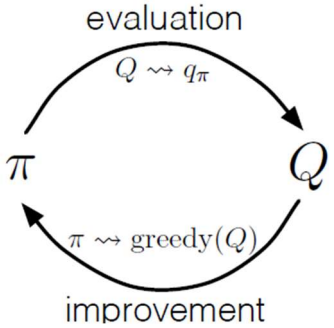
4.2 Monte Carlo Control

We want to find the optimal policy; for this, generalized policy iteration can be directly applied to the MC framework using action values:

$$\pi_0 \to \hat{q}_{\pi_0} \to \cdots \to \pi^* \to \hat{q}_{\pi^*}$$

This gives us another degree of freedom, choosing the number of episodes to approximate $\hat{q}_{\pi_i}$. Policy improvement is done by greedy choices:

$$\pi(s) = \arg\max_a q(s, a) \;\; \forall s \in \mathcal{S}$$



Assuming that one is operating in an unknown MDP, the policy improvement theorem (see *3.2 Policy Improvement*) is still valid for MC-based control.

Algorithm – First Visit

output: Optimal deterministic policy π^*
init: $\pi_{i=0}(s) \in \mathcal{A}$ arbitrarily $\forall s \in \mathcal{S}$
$\hat{q}(s,a)$ arbitrarily $\forall \{s \in \mathcal{S}, a \in \mathcal{A}\}$
$n(s,a) \leftarrow$ an empty list for state-action visits $\forall \{s \in \mathcal{S}, a \in \mathcal{A}\}$
repeat
$i \leftarrow i+1$;
Choose $\{s_0, a_0\}$ randomly such that all pairs have probability > 0 ;
Generate an episode from $\{s_0, a_0\}$ following π_i until termination step T_i;
Set $g \leftarrow 0$;
for $k = T_i - 1, T_i - 2, T_i - 3, \ldots, 0$ *time steps* **do**
$g \leftarrow \gamma g + r_{k+1}$;
if $\{s_k, a_k\} \notin \langle \{s_0, a_0\}, \ldots, \{s_{k-1}, a_{k-1}\} \rangle$ **then**
$n(s_k, a_k) \leftarrow n(s_k, a_k) + 1$;
$\hat{q}(s_k, a_k) \leftarrow \hat{q}(s_k, a_k) + 1/n(s_k, a_k) \cdot (g - \hat{q}(s_k, a_k))$;
$\pi_i(s_k) \leftarrow \arg\max_a \ \hat{q}(s_k, a)$;
until $\pi_{i+1} = \pi_i$;

We start with a random state-action pair $\{s_o, a_0\}$ and generate an episode following a certain policy π_i, until the termination of the episode at step T_i. In this generated episode, then iterate backwards over its time steps. For each time step, update the return. Then check whether the current state-action pair will also be visited earlier. If not, i.e., this is the first visit, increment the count $n(s_k, a_k)$ (over all episodes) of the state-action pair occurring. Update the q-value using the incremental update:

$$\hat{q}(s_k, a_k) \leftarrow \hat{q}(s_k, a_k) + \frac{1}{n(s_k, a_k)}[g - \hat{q}(s_k, a_k)]$$

Finally update the policy by a greedy policy improvement step: $\arg\max_a \hat{q}(s_k, a)$. We then repeat this procedure over a couple of episodes, as long as our derived policy does not change anymore from one iteration to the next $\pi_{i+1} = \pi$.

Note that we could exchange $\frac{1}{n(s_k, a_k)}$ by a constant learning factor α to allow dynamic adaptation.

4.3 Extensions to Monte Carlo On-Policy Control

In On-Policy Learning we evaluate /improve the policy that is used to make decisions. The agent picks its own actions. For MC we have to use exploring starts (ES) to ensure an sufficient level of exploration. ES is an on-policy method example; however, ES is a restrictive assumption and not always applicable. In some cases, the starting state-action pair cannot be chosen freely.

In Off-Policy Learning we evaluate or improve a policy that is different from the one used to generate data. The agent cannot apply own actions.

4.3.1 ϵ-Greedy

The general exploration requirement is to visit all state-action pairs with probability:

$$\pi(a|s) > 0 \ \forall \{s \in \mathcal{S}, a \in \mathcal{A}\}$$

Policies with this characteristic are called *soft*. The level of exploration can be tuned during the learning process.

With probability ϵ the agent's choice is overwritten with a random action. The probability of all non-greedy actions is:

$$\frac{\epsilon}{|\mathcal{A}|}$$

The probability of the greedy action is:

$$1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|}$$

$|\mathcal{A}|$ denotes the cardinality of the action space.

Algorithm - ϵ-Greedy MC-Control

output: Optimal ε-greedy policy $\pi^*(a|s)$
parameter: $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$
init: $\pi_{i=0}(a|s)$ arbitrarily soft $\forall \{s \in \mathcal{S}, a \in \mathcal{A}\}$
$\hat{q}(s,a)$ arbitrarily $\forall \{s \in \mathcal{S}, a \in \mathcal{A}\}$
$n(s,a) \leftarrow$ an empty list counting state-action visits $\forall \{s \in \mathcal{S}, a \in \mathcal{A}\}$
repeat
Generate an episode following π_i: $s_0, a_0, r_1, \ldots, s_{T_j}, a_{T_j}, r_{T_{j+1}}$;
$i \leftarrow i + 1$;
Set $g \leftarrow 0$;
for $k = T_i - 1, T_i - 2, T_i - 3, \ldots, 0$ *time steps* **do**
$g \leftarrow \gamma g + r_{k+1}$;
if $\{s_k, a_k\} \notin \langle \{s_0, a_0\}, \ldots, \{s_{k-1}, a_{k-1}\} \rangle$ **then**
$n(s_k, a_k) \leftarrow n(s_k, a_k) + 1$;
$\hat{q}(s_k, a_k) \leftarrow \hat{q}(s_k, a_k) + 1/n(s_k, a_k) \cdot (g - \hat{q}(s_k, a_k))$;
$\tilde{a} \leftarrow \arg\max_a \hat{q}(s_k, a)$;
$\pi_i(a|s_k) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}|, & a = \tilde{a} \\ \varepsilon/|\mathcal{A}|, & a \neq \tilde{a} \end{cases}$;
until $\pi_{i+1} = \pi_i$;

The output of this algorithm is an optimal ϵ-greedy policy, which differs from regular optimal policies. The key difference to the regular MC-Control algorithm (see *4.2*) is in the update of the policy, where we define the action by the ϵ-greedy approach.

ϵ-greedy Policy Improvement Theorem

Given an MDP, for any ϵ-greedy policy π the ϵ-greedy policy π' with respect to q_π is an improvement, i.e., $v_{\pi'} > v_\pi \; \forall s \in \mathcal{S}$. The original theorem still applies in a sense that we now only compare two ϵ-greedy policies with each other. There always might be a non- ϵ-greedy policy that is better.

Observations on the forest tree MDP example with ϵ-greedy MC-based control:

- Rather slow convergence rate: many episodes are required

- Significant uncertainty is present in a single sequence
- Later states are less often visited and therefore more uncertain
- Exploration is controlled by ϵ: in a completely greedy policy the state $s = 3$ is not visited at all as the optimal decision for state $s = 2$ is to cut the tree down. With ϵ-greedy this state is visited occasionally.

4.3.2 Greedy in the Limit with Infinite Exploration (GLIE)

A *learning policy* π is called GLIE if it satisfies the following two properties:

- If a state is visited infinitely often, then each action is chosen infinitely often:
$$\lim_{i\to\infty} \pi_i(a|s) = 1 \ \forall\{s \in \mathcal{S}, a \in \mathcal{A}\}$$
- In the limit ($i \to \infty$) the learning policy is greedy with respect to the learned action value:
$$\lim_{i\to\infty} \pi_i(s|a) = \pi(s) = \arg\max_a q(s,a) \ \forall s \in \mathcal{S}$$

To achieve the second requirement with ϵ-greedy, we adaptively changing the ϵ-greedy value.

Optimal decision using MC-control with ϵ-greedy

MC-based control using ϵ-greedy exploration is GLIE, if ϵ is decreased at rate:

$$\epsilon_i = \frac{1}{i}$$

with i being the increasing episode index. In this case,

$$\hat{q}(s,a) = q^*(s,a)$$

follows.

This is slightly infeasible as an infinite number of episodes are required.

4.4 Monte Carlo Off-Policy Prediction

The drawback of on-policy learning is that we'll only learn a near-optimal policy. This drawback comes with inherent exploration. The idea of off-policy learning is to use two separate policies:

- *Behavior policy* $b(a|s)$: Explores in order to generate experience
- *Target policy* $\pi(a|s)$: Learns from the generated experience to become the optimal policy

This allows to re-use experience generated from old policies ($\pi_0, \pi_1, \ldots$), and/or to learn about multiple policies while following just one. Also learning from observing humans or agents/controllers is possible.

We want to estimate a state value v_π and/or q_π for the target policy π while following the behavior policy $b(a|s)$. Both policies are considered fixed during the evaluation. To do so, we have to introduce a requirement:

- *Coverage*: Every action taken under π must be (at least occasionally) taken under b too. Hence, it follows:

$$\pi(a|s) > 0 \Longrightarrow b(a|s) > 0 \quad \forall\{s \in \mathcal{S}, a \in \mathcal{A}\}$$

Consequences from that are

- in any state b is not identical to π, b then must be stochastic.
- π could be deterministic or stochastic

❓ How can we map experiences from the behavior policy $b(a|s)$ to the target policy π?

4.4.1 Importance Sampling Ratio

Importance sampling is a general technique for estimating expected values under one distribution given samples from another.

But what is the probability of specifically observing a certain trajectory (sequence of random actions & states) on random variables $A_k, S_{k+1}, A_{k+1}, \dots, S_T$ starting in S_k while following π?

$$\begin{aligned}
&\mathbb{P}[A_k, S_{k+1}, A_{k+1}, \dots, S_T | S_k, \pi] \\
&\quad = \pi(A_k|S_k) * p(S_{k+1}|S_k, A_k) * \pi(A_{k+1}|S_{k+1}) * \cdots \\
&\quad = \prod_{k}^{T-1} \pi(A_k|S_k) * p(S_{k+1}|A_k)
\end{aligned}$$

Product over all time steps of an episode $k:t$

Policy; probability of choosing action A_k in state S_k

state transition probabililty

With $p(S_{k+1}|S_k, A_k)$ being the state transition probability of transitioning from state S_k to S_{k+1} when picking action A_k.

The relative probability of a trajectory under the target and behavior policy, the *importance sampling ratio*, from sample step k to T is:

$$\rho_{k:T} = \frac{\prod_k^{T-1} \pi(A_k|S_k) * p(S_{k+1}|A_k)}{\prod_k^{T-1} b(A_k|S_k) * p(S_{k+1}|A_k)} = \frac{\prod_k^{T-1} \pi(A_k|S_k)}{\prod_k^{T-1} b(A_k|S_k)}$$

This tells us, in other words, how big the likelihood of observing the one trajectory following the behavior policy and then mapping it to the target policy.

$\rho_{k:T}$ denotes the episodes we are considering, starting at sampling step k until termination T.

Interpretation

$\rho_{k:T} > 1$	Observed trajectory is *more likely* to be observed in the target policy π than the behavior policy b → very representative
$\rho_{k:T} < 1$	Observed trajectory is *less likely* to be observed in the behavior policy b than the target policy π → not very representative

4.4.2 Ordinary Importance Sampling (OIS)

We now apply this importance sampling ratio, also called ordinary importance sampling (OIS), to the Monte Carlo Prediction. It is convenient here to number time steps in a way that increases across episode boundaries. That is, if the first episode of the batch ends in a terminal state at time $k = 100$, then the next episode begins at time $k = 101$. This enables us to use time-step numbers to refer to particular steps in particular episodes. In particular, we can define the set of all time steps in which state s is visited, denoted $\mathcal{T}(s_k)$.

This is for an every-visit method; for a first-visit method, $\mathcal{T}(s_k)$ would only include time steps that were first visits to s within their episodes.

Estimating the state value v_π following a behavior policy b using ordinary importance sampling, results in scaling and averaging the sampled returns g_k (by policy b) by the importance sampling ratio $\rho_{k:T}$ per episode:

$$\hat{v}_\pi(s_k) = \frac{\sum_{k \in \mathcal{T}(s_k)} \rho_{k:T(k)} * g_k}{|\mathcal{T}(s_k)|}$$

Notation remark:

- $|\mathcal{T}(s_k)|$ is the amount of time steps in $\mathcal{T}(s_k)$
- $\mathcal{T}(s_k)$ denotes the set of all time steps in which the state s_k is visited
- $T(k)$ is the termination of a specific episode starting from time step k

❗ This mapping to $\hat{v}$ Is perfectly fine, meaning its bias-free (first-visit assumption). However, if ρ is large (distinctly different policies) the estimates variance is large.

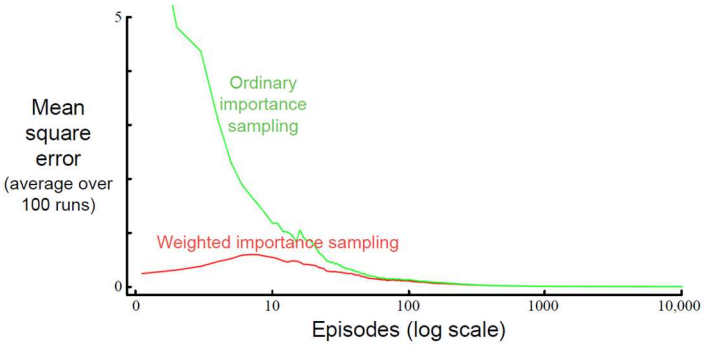
4.4.3 Weighted Importance Sampling (WIS)

Estimating the state value v_π following a behavior policy b using weighted importance sampling, results in scaling and a weighted averaging of the sampled returns g_k by the importance sampling ratio $\rho_{k:T}$ per episode.

$$\hat{v}_\pi(s_k) = \frac{\sum_{k \in \mathcal{T}(s_k)} \rho_{k:T(k)} * g_k}{\sum_{k \in \mathcal{T}(s_k)} \rho_{k:T(k)}}$$

or $\hat{v}_\pi(s_k) = 0$, if the denominator is zero.

❗ By this approach, we introduce bias (which vanishes to zero in the limit). But we limit variance while the OIS's variance is unbounded. → Bias–Variance Dilemma.



In the long run (*in this figure for ~episodes* > 100) OIS actually performs better than WIS.

4.4.4 Incremental Implementation (WIS)

MC prediction methods can be implemented incrementally, on an episode-by-episode basis, following the incremental/recursive implementation of the sample mean (*see: 4.1.1 Incremental Implementation*), whereas here we average returns.

For WIS an incremental implementation is also highly desirable to save memory and computational resources. For this, we rewrite WIS:

$$\hat{v}_i(s) = \frac{\sum_{k=1}^{i=1} w_k \, g_k}{\sum_{k=1}^{i=1} w_k} \quad with \quad w_k = \rho_{k:T(k)}$$

The recursive/incremental update rule is then:

$$\hat{v}_{i+1}(s) = \hat{v}_i(s) + \frac{w_i}{c_i}\big(g_i - \hat{v}_i(s)\big), \quad i \geq 1$$

$$c_{i+1} = c_i + w_{i+1} \quad with \quad c_0 = 0$$

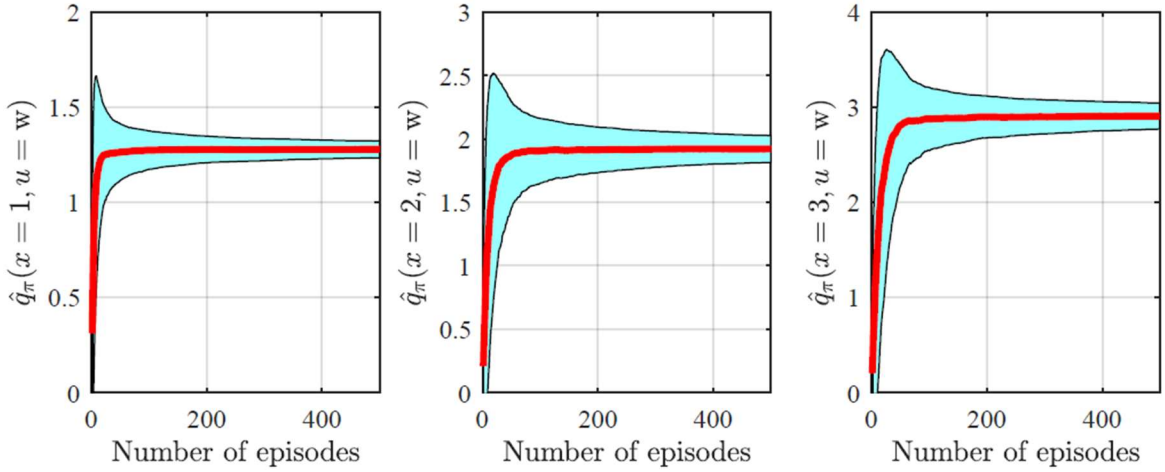
Above, c_i is the cumulative sum of weights over all considered state visits.

Algorithm

input: a target policy π to be evaluated
init: $\hat{q}(s,a)$ arbitrarily $\forall\ \{s \in \mathcal{S}, a \in \mathcal{A}\}$
$c(s,a) \leftarrow 0$ cumulative sum of WIS weights $\forall\ \{s \in \mathcal{S}, a \in \mathcal{A}\}$
for $j = 1, \ldots, J$ *episodes* **do**
Choose an arbitrary behavior policy b with coverage of π;
Generate an episode following b: $s_0, a_0, r_1, \ldots, s_{T_j}, a_{T_j}, r_{T_{j+1}}$;
Set $g \leftarrow 0$;
Set $w \leftarrow 1$;
for $k = T_i - 1, T_i - 2, T_i - 3, \ldots, 0$ *time steps while* $w \neq 0$ **do**
$g \leftarrow \gamma g + r_{k+1}$;
$c(s_k, a_k) \leftarrow c(s_k, a_k) + w$;
$\hat{q}(s_k, a_k) \leftarrow \hat{q}(s_k, a_k) + \frac{w}{c(s_k, a_k)}\,(g - \hat{q}(s_k, a_k))$;
$w \leftarrow w \frac{\pi(a_k|s_k)}{b(a_k|s_k)}$;

MC-based off-policy prediction using WIS

Example



Algorithm applied to the forest tree MDP with b being 50/50

4.5 Monte Carlo Off-Policy Control

This puts MC-based control utilizing GPI together with off-policy learning based on above algorithm. Requirement for off-policy MC-based control are:

- *Coverage*: behavior policy b has non-zero probability of selecting actions that might be taken by the target policy π. This results in the behavior policy b being *soft.*

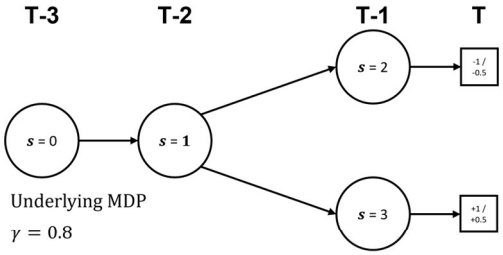
Algorithm

init: $\hat{q}(s,a)$ arbitrarily $\forall\ \{s \in \mathcal{S}, a \in \mathcal{A}\}$
$c(s,a) \leftarrow 0$ cumulative sum of WIS weights $\forall\ \{s \in \mathcal{S}, a \in \mathcal{A}\}$
$\pi(s) \leftarrow \arg\max_a \hat{q}(s,a)$ (with ties broken consistently)
for $j = 1, \ldots, J$ *episodes* **do**
Choose an arbitrary soft policy b;
Generate an episode following b: $s_0, a_0, r_1, \ldots, s_{T_j}, a_{T_j}, r_{T_{j+1}}$;
Set $g \leftarrow 0$;
Set $w \leftarrow 1$;
for $k = T_i - 1, T_i - 2, T_i - 3, \ldots, 0$ *time steps* **do**
$g \leftarrow \gamma g + r_{k+1}$;
$c(s_k, a_k) \leftarrow c(s_k, a_k) + w$;
$\hat{q}(s_k, a_k) \leftarrow \hat{q}(s_k, a_k) + \frac{w}{c(s_k, a_k)} (g - \hat{q}(s_k, a_k))$;
$\pi(s_k) \leftarrow \arg\max_{a_k} \hat{q}(s_k, a_k)$ (with ties broken consistently);
if $a_k \neq \pi(s_k)$ **then**
break/exit inner loop;
$w \leftarrow w \frac{1}{b(a_k|s_k)}$;

MC-based off-policy control using WIS

You may have been expecting the w update to have involved the importance-sampling ratio $\frac{\pi(A_t|S_t)}{\mathrm{b}(\mathrm{A_t}|\mathrm{S_t})}$, but instead it involves $\frac{1}{b(A_t|S_t)}$. This is nevertheless correct, because our policy $\pi(S_t)$ is a deterministic, greedy one, we are only observing trajectories where $\pi(A_t|S_t) = 1$, hence the numerator in the equation.

Example



Underlying MDP

$\gamma = 0.8$

$b(a|s)$ is epsilon greedy with $\epsilon = 0.1$ and sampled following trajectories:

- e_1: $\langle s_0, ANY, 0, \quad s_1, DOWN, 0, \quad s_3, ANY, \quad 1\rangle$
- e_2: $\langle s_0, ANY, 0, \quad s_1, DOWN, 0, \quad s_3, ANY, \quad 0.5\rangle$
- e_3: $\langle s_0, ANY, 0, \quad s_1, UP, 0, \quad s_2, ANY, -\ 1\rangle$
- e_4: $\langle s_0, ANY, 0, \quad s_1, UP, 0, \quad s_2, ANY, -0.5\rangle$

	$s = 0$	$s = 1$	$s = 2$	$s = 3$
e_1	$g = 0.8 * 0.8 + 0 = \mathbf{0.64}$ $c = 0 + 1 = 1$ $q = 0 + \frac{1.23}{1}(0.64 - 0) = \mathbf{0.78}$ $w = 1.23\frac{1}{0.9} = \mathbf{1.36}$	$g = 0.8 * 1 + 0 = \mathbf{0.8}$ $c = 0 + 1 = \mathbf{1}$ $q = 0 + \frac{1.11}{1}(0.8 - 0) = \mathbf{0.89}$ $w = 1.11\frac{1}{0.9} = \mathbf{1.23}$	not visited	$g = 0.8 * 0 + 1 = 1$ $c = 0 + 1 = \mathbf{1}$ $q = 0 + \frac{1}{1}(1 - 0) = \mathbf{1}$ $w = 1\frac{1}{0.9} = 1.\overline{11}$
e_2	$g = 0.8 * 0.4 + 0 = \mathbf{0.32}$ $c = 1 + 1 = \mathbf{2}$ $q = 0.78 + \frac{1.23}{2}(0.32 - 0.78) = \mathbf{0.5}$ $w = 1.23\frac{1}{0.9} = \mathbf{1.36}$	$g = 0.8 * 0.5 + 0 = \mathbf{0.4}$ $c = 1 + 1 = \mathbf{2}$ $q = 0.89 + \frac{1.11}{2}(0.4 - 0.89) = \mathbf{0.62}$ $w = 1.11\frac{1}{0.9} = \mathbf{1.23}$	not visited	$g = 0.8 * 0 + 0.5 = \mathbf{0.5}$ $c = 1 + 1 = \mathbf{2}$ $q = 1 + \frac{1}{2}(0.5 - 1) = \mathbf{0.75}$ $w = 1\frac{1}{0.9} = \mathbf{1.11}$
e_3	We stop calculating q_π for this state as in this following trajectory a suboptimal action, which would not have happend in π, was chosen by b. Therefore, g of the previous iteration would not be applicable to this state.	$g = 0.8 * -1 + 0 = \mathbf{-0.8}$ $c = 0 + 1 = \mathbf{1}$ $q = 0 + \frac{1.11}{1}(-0.8 - 0) = \mathbf{-0.89}$ break	$g = 0.8 * 0 - 1 = \mathbf{-1}$ $c = 0 + 1 = \mathbf{1}$ $q = 0 + \frac{1}{1}(-1 - 0) = \mathbf{-1}$ $w = 1\frac{1}{0.9} = \mathbf{1.11}$	not visited
e_4	see above	$g = 0.8 * -0.5 + 0 = \mathbf{-0.4}$ $c = 1 + 1 = \mathbf{2}$ $q = -0.89 + \frac{1.11}{2}(-0.4 - (-0.89)) = \mathbf{-0.62}$ Break	$g = 0.8 * 0 - 0.5 = \mathbf{-0.5}$ $c = 1 + 1 = \mathbf{2}$ $q = -1 + \frac{1}{2}(-0.5 - (-1)) = \mathbf{-0.75}$ $w = 1.11\frac{1}{0.9} = \mathbf{1.23}$	not visited

q_π-values for the greedy target policy π after four episodes:

$state$	$s = 0$	$s = 1$	$s = 2$	$s = 3$
$action$	$q_\pi = \mathbf{0.5}$	UP: $q_\pi = \mathbf{-0.62}$ $DOWN$: $q_\pi = \mathbf{0.62}$	$q_\pi = \mathbf{-0.75}$	$q_\pi = \mathbf{0.75}$

❗ Potential problems with MC-based off-policy control:

- Learns only 'from tails' of episodes if the remaining actions generated by b are greedy with respect to π (last if-statement in algo. above). Hence, a lot of samples generated by b remain unused for training π
- Slows down training
- Adds uncertainty to the training process

4.6 Summary

- MC methods allow model-free learning of value functions and optimal policies from experience in the form of sampled episodes.
- Using deep back-ups over full episodes, MC is largely based on averaging returns.
- MC-based control reuses GPI, i.e., mixing evaluation and improvement.

Maintaining sufficient exploration is important:

- Exploring starts: simple but not feasible in all applications
- On-policy ϵ-greedy learning: trade-off between optimality and exploration cannot be resolved easily.

- Off-policy learning: agent learns about a target policy from an exploratory, soft behavior policy

Importance sampling transforms expectations from the behavior to the target policy

- This estimation task comes with bias-variance dilemma
- Slow learning can result from ineffective experience usage in MC methods

5 Temporal-Difference Learning

Temporal-difference (TD) learning is a combination of Monte Carlo ideas and dynamic programming ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). Hence, TD characteristics are:

- Model-free prediction and control in unknown MDPs.
- Policy Evaluation and Improvement are updated in an online fashion (i.e., not per episodes) by bootstrapping)
- Assuming finite MDP problems

5.1 TD Prediction

Given some experience following a policy π, update the estimate $\hat{v}(s_k)$ for the nonterminal states occurring in that experience:

$$\hat{v}(s_k) \leftarrow \hat{v}(s_k) + \alpha[r_{k+1} + \gamma\hat{v}(s_{k+1}) - \hat{v}(s_k)]$$

The target of TD is $r_{k+1} + \gamma\hat{v}(s_{k+1})$. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $\hat{v}(s_k)$ (only then is g_k known), TD methods need to wait only until the next time step. This TD method is called `TD(0)`, or one-step TD.

Algorithm

input: a policy π to be evaluated
output: estimate of $v_{\mathcal{S}}^{\pi}$ (i.e., value estimates for all states $s \in \mathcal{S}$)
init: $\hat{v}(s) \forall\, s \in \mathcal{S}$ arbitrary except $v_0(s) = 0$ if x is terminal
for $j = 1, \ldots, J$ *episodes* **do**
    Initialize s_0;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $a_k \leftarrow$ apply action from $\pi(s_k)$;
        Observe s_{k+1} and r_{k+1};
        $\hat{v}(s_k) \leftarrow \hat{v}(s_k) + \alpha\left[r_{k+1} + \gamma\hat{v}(s_{k+1}) - \hat{v}(s_k)\right]$;
        Exit loop if s_{k+1} is terminal;

Note that the algorithm can be directly adapted to action-value prediction as it will be used for the later TD-based control approaches

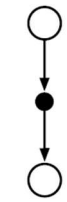
Because TD(0) bases its update in part on an existing estimate, we say that it is a *bootstrapping* method, like DP. We know from Chapter 3 that

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_k \mid S_k = s] && (6.3)\\ &= \mathbb{E}_\pi[R_{k+1} + \gamma G_{k+1} \mid S_k = s] \\ &= \mathbb{E}_\pi[R_{k+1} + \gamma v_\pi(S_{k+1}) \mid S_k = s]\,. && (6.4) \end{aligned}$$

Roughly speaking, Monte Carlo methods use an estimate of (6.3) as a target, whereas DP methods use an estimate of (6.4) as a target. The Monte Carlo target is an estimate because the expected value in (6.3) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v_\pi(S_{k+1})$ is not known and the current estimate, $V(S_{k+1})$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in (6.4) *and* it uses the current estimate V instead of the true v_π.

5.1.1 TD Error

Shown to the right is the backup diagram for tabular `TD(0)`. The value estimate for the state node at the top of the backup diagram is updated on the basis of the one sample transition from it to the immediately following state. We refer to TD and Monte Carlo updates as sample updates because they involve looking ahead to a sample successor state (or state–action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then updating the value of the original state (or state– action pair) accordingly.

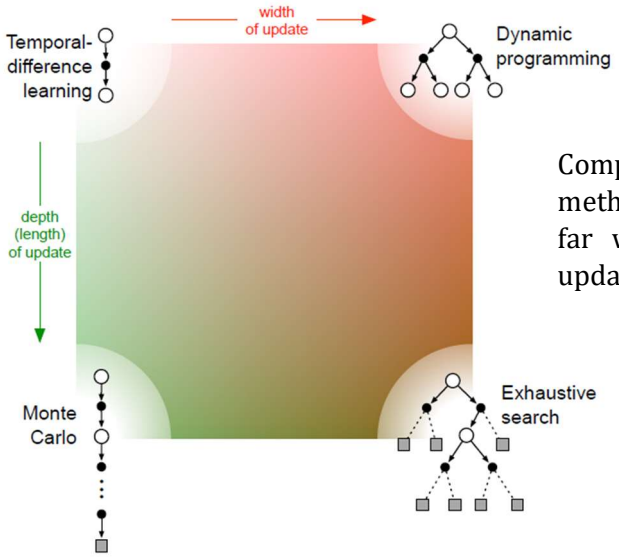


TD(0)

Finally, note that the quantity in brackets in the `TD(0)` update is a sort of error, measuring the difference between the estimated value of s_k and the better estimate $r_{k+1} + \gamma\hat{v}(s_{k+1})$. This quantity, called the *TD error*, arises in various forms throughout reinforcement learning:

$$\delta = r_{k+1} + \gamma\hat{v}(s_{k+1}) - \hat{v}(s_k)$$

Notice that the TD error at each time is the error in the estimate made at that time. Because the TD error depends on the next state and next reward, it is not actually available until one time step later. That is, (δ_k is the error in $\hat{v}(s_k)$, available at time $k + 1$).

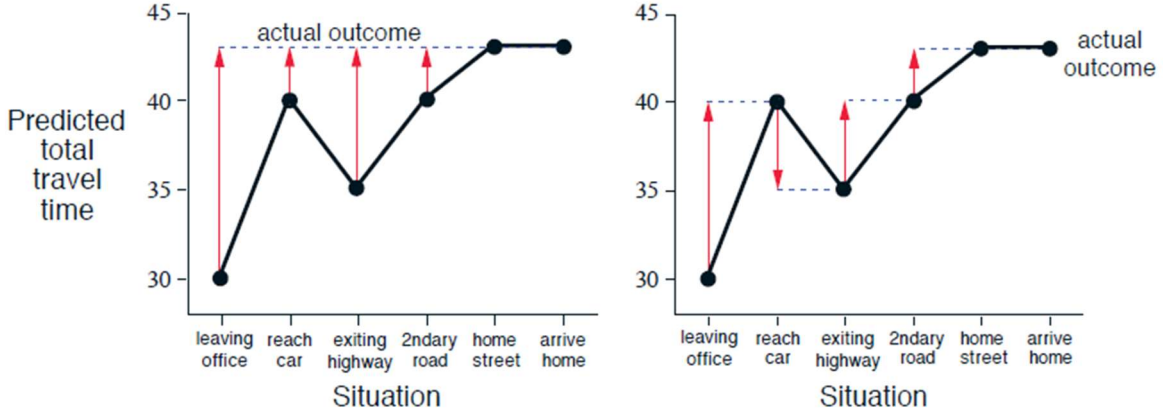


Comparison of the RL methods considered so far with regard to the update rules.

5.1.2 Example: Driving Home

Each day as you drive home from work, you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, the weather, and anything else that might be relevant.

state	elapsed time	predicted time to go	predicted total time
leaving office	0	30	30
reaching car, raining	5	35	40
exiting highway	20	15	35
behind truck	30	10	40
home street	40	3	43
arrive home	43	0	43



Changes recommended in the driving home example by MC (left) and TD methods (right).

The red arrows show the changes in predictions recommended by the constant-α MC method for $\alpha = 1$. These are exactly the errors between the estimated value (predicted time to go) in each state and the actual return (actual time to go). For example, when you exited the highway, you thought it would take only 15 minutes more to get home, but in fact it took 23 minutes.

5.1.3 Convergence of TD(0)

Suppose there is available only a finite amount of experience, say 10 episodes or 100 time steps. In this case, a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. Given an approximate value function the increments are computed for every time step at which a nonterminal state is visited, but the value function is changed only once, by the sum of all the increments. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges. We call this *batch updating* because updates are made only after processing each complete batch of training data.

Theorem

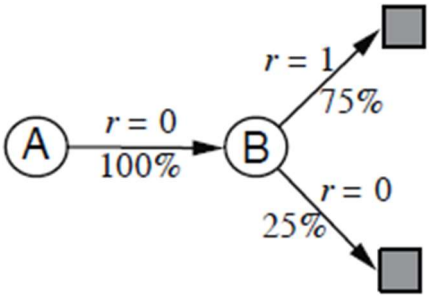
Given a finite MDP and a fixed policy π the state-value estimate of `TD(0)` converges to the true v_π:

- In the mean for a constant but sufficiently small step-size α and
- with probability 1 if the *step-size* α holds the condition:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \qquad and \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

Above k is the sample index (i.e., how often the TD update was applied). In particular, $\alpha_k = \frac{1}{k}$ meets the condition above.

5.1.4 Example: AB-Batch Training



A, 0, B, 0	B,1
B,1	B,1
B,1	B,1
B,1	B,0

episodes of experience available

This means that the first episode started in state A, transitioned to B with a reward of 0, and then terminated from B with a reward of 0. The other seven episodes were even shorter, starting from B and terminating immediately. Given this batch of data, what are the optimal state-value predictions?

Observe that 100% of the times the process was in state A it traversed immediately to B (with a reward of 0); and because we have already decided that B has value $\frac{3}{4}$, therefore A must have value $\frac{3}{4}$ as well. This is also the answer that batch `TD(0)` gives. The other reasonable answer is simply to observe that we have seen A once and the return that followed it was 0; we therefore estimate $V(A)$ as 0. This is the answer that batch Monte Carlo methods give.

Batch Monte Carlo methods always find the estimates that minimize mean square error (MSE) on the training set, whereas batch `TD(0)` always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process.

5.2 TD On-Policy Control: SARSA

With SARSA we follow the pattern of *Generalized Policy Iteration (GPI)*, only this time using TD methods for the evaluation or prediction part. The `TD(0)` action-value update is:

$$\hat{q}(s_k, a_k) \leftarrow \hat{q}(s_k, a_k) + \alpha[r_{k+1} + \gamma \hat{q}(s_{k+1}, a_{k+1}) - \hat{q}(s_k, a_k)]$$

In contrast to MC, we continuously perform online updates of policy evaluation and improvement. Every On-policy approach requires exploration, e.g., by an ϵ-Greedy policy.

Algorithm

parameter: $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}, \quad \alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: $\hat{q}(s, a)$ arbitrarily (except terminal states) $\forall \{s \in \mathcal{S}, a \in \mathcal{A}\}$
for $j = 1, 2, \ldots$ *episodes* **do**
    Initialize s_0;
    Choose a_0 from s_0 using a soft policy (e.g., ε-greedy) derived from $\hat{q}(s, a)$;
    $k \leftarrow 0$;
    repeat
        Take action a_k, observe r_{k+1} and s_{k+1};
        Choose a_{k+1} from s_{k+1} using a soft policy derived from $\hat{q}(s, a)$;
        $\hat{q}(s_k, a_k) \leftarrow \hat{q}(s_k, a_k) + \alpha\left[r_{k+1} + \gamma \hat{q}(s_{k+1}, a_{k+1}) - \hat{q}(s_k, a_k)\right]$;
        $k \leftarrow k + 1$;
    until s_k *is terminal*;

The $\hat{q}(s_k, a_k)$ update is done after every transition from a nonterminal state S_k. If S_{k+1} is terminal, then $\hat{q}(s_{k+1}, a_{k+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(S_k, A_k, R_{k+1}, S_{k+1}, A_{k+1})$, that make up a transition from one state–action pair to the next, hence the name SARSA for the algorithm.

5.3 TD Off-Policy Control: Q-Learning

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning.

In this case, the learned action-value function, Q, directly approximates q^*, the optimal action-value function, independent of the policy being followed.

$$\hat{q}(s_k, a_k) \leftarrow \hat{q}(s_k, a_k) + \alpha \left[r_{k+1} + \gamma \max_a \hat{q}(s_{k+1}, a) - \hat{q}(s_k, a_k) \right]$$

This is an off-policy update, since the optimal action-value function is updated independent of a given behavior policy. The policy still has an effect in that it determines which state–action pairs are visited and updated. Requirement for Q-learning control:

- Coverage: behavior policy b has nonzero probability of selecting actions that might be taken by the target policy $\pi \rightarrow$ behavior policy b is soft (e.g., ε-soft).
- Step-size requirements (*see: 5.1.3*) regarding α apply.

Algorithm

parameter: $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}, \quad \alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: $\hat{q}(s, a)$ arbitrarily (except terminal states) $\forall \{s \in \mathcal{S}, a \in \mathcal{A}\}$
for $j = 1, 2, \ldots$ *episodes* **do**
    Initialize s_0;
    $k \leftarrow 0$;
    repeat
        Choose a_k from s_k using a soft policy derived from $\hat{q}(s, a)$;
        Take action a_k, observe r_{k+1} and s_{k+1};
        $\hat{q}(s_k, a_k) \leftarrow \hat{q}(s_k, a_k) + \alpha \left[r_{k+1} + \gamma \max_a \hat{q}(s_{k+1}, a) - \hat{q}(s_k, a_k) \right]$;
        $k \leftarrow k + 1$;
    until s_k *is terminal*;

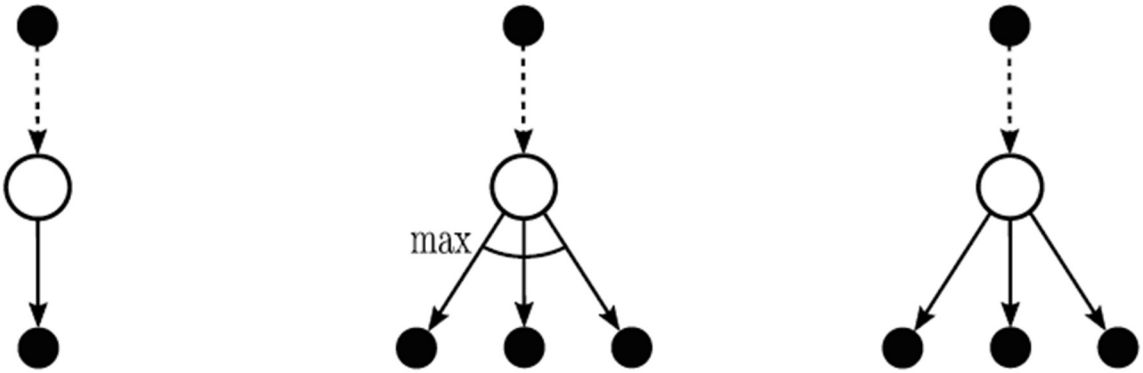
5.4 Expected SARSA

Consider a learning algorithm that is just like Q-learning except that instead of the maximum over next state–action pairs it uses the expected value, considering how likely each action is under the current policy. The expected Sarsa action-value update is:

$$\hat{q}(s_k, a_k) \leftarrow \hat{q}(s_k, a_k) + \alpha [r_{k+1} + \gamma \mathbb{E}_\pi [\hat{q}(s_{k+1}, a_{k+1}) | s_{k+1}] - \hat{q}(s_k, a_k)]$$

$$\leftarrow \hat{q}(s_k, a_k) + \alpha \left[r_{k+1} + \gamma \sum_a \pi(a | s_{k+1}) \hat{q}(s_{k+1}, a) - \hat{q}(s_k, a_k) \right]$$

This is an off or on-policy update, depending on whether the action leading to s_{k+1} was taken from the target or a varying behavior policy.



Back-up Diagrams for Sarsa (left), Q-learning (middle) and Expected Sarsa (right).

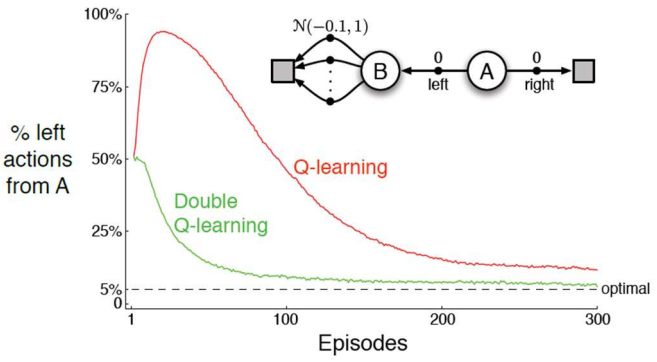
They all use sample updates based on a one-step look-ahead evaluation and improve estimates based on other estimates (bootstrap). While Sarsa updates based on specific state-action transitions, Q-Learning updates the optimal policy estimate q^* and Expected Sarsa updates considering the expected transitions.

5.5 Maximization Bias and Double Learning

All the control algorithms that we have discussed so far involve maximization in the construction of their target policies. For example, in Q-learning the target policy is the greedy policy given the current action values, which is defined with a max, and in Sarsa the policy is often "-greedy, which also involves a maximization operation. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias.

Example

The small MDP shown provides a simple example of how maximization bias can harm the performance of TD control algorithms. The MDP has two non-terminal states A and B. Episodes always start in A with a choice between two actions, left and right. The right action transitions immediately to the terminal state with a reward and return of zero.



The left action transitions to B, also with a reward of zero, from which there are many possible actions all of which cause immediate termination with a reward drawn from a normal distribution with mean −0.1 and variance 1.0.

Thus, the expected return for any trajectory starting with left is −0.1, and thus taking left in state A is always a mistake. Nevertheless, our control methods may favor left because of maximization bias making B appear to have a positive value. The figure shows that Q-learning with ϵ-greedy action selection initially learns to strongly favor the left action on this example.

💡 This happens because the samples generated will eventually be > 0 in the beginning of the training due to the high variance. This skews our q-estimates and results into favoring one action. After sampling action B often enough, so that its estimate drops below 0, we will then pick action A again.

Approach

When performing *Double Learning* were dividing sampled experience into two sets and use them to calculate independent estimates $\hat{q}_1(s,a)$ and $\hat{q}_2(s,a)$.

$\hat{q}_1(s,a)$ could be used to estimate the maximization action (classical greedy choice):

$$a^* = \arg\max_a \hat{q}_1(s,a)$$

while $\hat{q}_2(s,a)$ is used to estimate the corresponding action value:

$$q(s,a) \approx \hat{q}_2(s,a^*) = \hat{q}_2\left(s, \arg\max_a \hat{q}_1(s,a)\right)$$

This doubles the memory requirements, but the amount of computation per step remains the same.

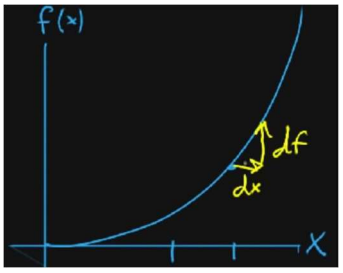
Algorithm

parameter: $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$, $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$
init: $\hat{q}_1(s,a)$, $\hat{q}_2(s,a)$ arbitrarily (except terminal states) $\forall \{s \in \mathcal{S}, a \in \mathcal{A}\}$
for $j = 1, 2, \ldots$ *episodes* **do**
  Initialize s_0;
  $k \leftarrow 0$;
  repeat
    Choose a_k from s_k using the policy ε-greedy based on $\hat{q}_1(s,a) + \hat{q}_2(s,a)$;
    Take action a_k, observe r_{k+1} and s_{k+1};
    if $n \sim \mathcal{N}(\mu = 0, \sigma) > 0$ **then**
      $\hat{q}_1(s_k, a_k) \leftarrow$
        $\hat{q}_1(s_k, a_k) + \alpha\,[r_{k+1} + \gamma \hat{q}_2(s_{k+1}, \arg\max_u \hat{q}_1(s_{k+1}, a)) - \hat{q}_1(s_k, a_k)]$;
    else
      $\hat{q}_2(s_k, a_k) \leftarrow$
        $\hat{q}_2(s_k, a_k) + \alpha\,[r_{k+1} + \gamma \hat{q}_1(s_{k+1}, \arg\max_u \hat{q}_2(s_{k+1}, a)) - \hat{q}_2(s_k, a_k)]$;
    $k \leftarrow k + 1$;
  until s_k *is terminal*;

We're splitting the experience generated randomly, so either $\hat{q}_1(s,a)$ or $\hat{q}_2(s,a)$ will be updated.

6 Recap: Basics

6.1 Partial Derivatives

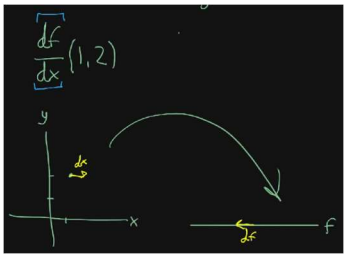


Ordinary derivatives are notated like this for example:

$$\frac{df}{dx} \text{ for } f(x) = x^2$$

This can be read as: "how does a change in the x-direction (df) influence the f-direction (df). The same can essentially applied to multivariable functions , e.g.:

$$\frac{df}{dx} \text{ for } f(x, y) = x^2 y + \sin(y)$$



Likewise, this his can be read as: "how does a change df influence the output (still a number)". The same can be done with the y-variable; except this time, dy would be a change in the y-direction.

Maybe changing f according to y it does something differently, e.g., increasing/decreasing the output a lot/a little.

The notation above is the same as $\frac{\partial f}{\partial x}$, where ∂ reads as "partial".

The example on the right, where we take the partial derivative of x with respect to f ($\frac{\partial f}{\partial x}$), only cares about the x-direction, treating y as a constant. Let's do it for a certain point (1,2):

$$\frac{\partial f}{\partial x}(1,2) = \frac{\partial}{\partial x}(x^2 * 2 + sin(2))\Big|_{x=1}$$
$$= 4x + 0 = 4$$

Most of the time we don't do it at a single point though.

$$\frac{\partial f}{\partial x}(x, y) = \frac{\partial}{\partial x}(x^2 * y + \sin(y))$$
$$= 2xy + 0$$
$$= 2xy$$

$$\frac{\partial f}{\partial y}(x, y) = \frac{\partial}{\partial y}(x^2 * y + \sin(y))$$
$$= x^2 + \cos(y)$$

6.2 Derivation Rules

6.2.1 Chain Rule

The chain rule says:

$$\frac{\partial}{\partial x}[f(g(x))] = f'(g(x)) * g'(x)$$

It tells us how to differentiate composite functions.

A function is composite if you can write it as $f(g(x))$. In other words, it is a function within a function, or a function of a function.

For example, $\cos(x^2)$ is composite and its derivative is $-\sin(x^2) * 2x$

6.3 Gradient

The gradient ∇ of a function f is a vector containing all partial derivatives of f. So, for example:

$$f(x) = x^2 \sin(y)$$

$$\frac{\partial f}{\partial x} = 2x * \sin(y)$$

$$\frac{\partial f}{\partial y} = x^2 * \cos(y)$$

$$\nabla f(x,y) = \begin{bmatrix} 2x * \sin(y) \\ x^2 * \cos(y) \end{bmatrix}$$

The function $\nabla f(x,y)$ takes in a point in two-dimensional space and outputs a two-dimensional vector.

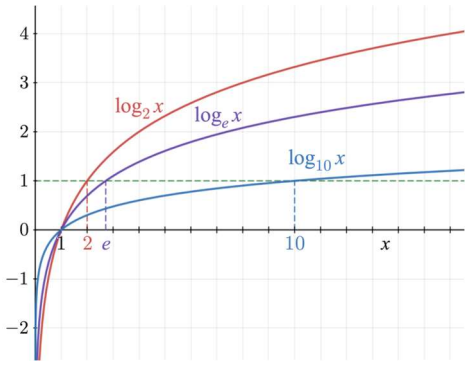
6.4 Logarithm

The logarithm is defined as the inverses of exponentials:

$$\log_a(x) = y \Leftrightarrow x = a^y$$

For example:

$$\log_2(4) = 2 \Leftrightarrow 4 = 2^2$$



The derivative of the log is defined as:

$$\frac{\partial}{\partial x}\ln(x) = \frac{1}{x}$$

$$\frac{\partial}{\partial x}\log_2(x) = \frac{1}{x * \ln(2)}$$

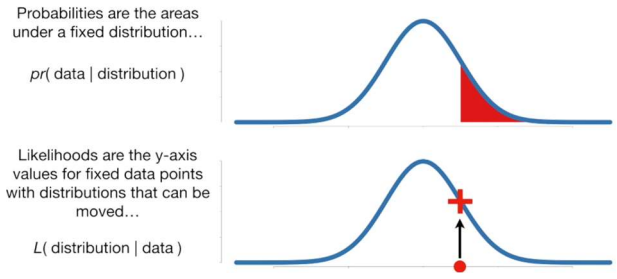
$$\frac{\partial}{\partial x}\log_{10}(x) = \frac{1}{x * \ln(10)}$$

Or, more generally:

$$\frac{\partial}{\partial x}\log_a(x) = \frac{1}{x * \ln(a)}$$

6.5 (Log-) Likelihood

A likelihood method is a measure of how well a particular model fits the data. They explain how well a parameter ($\boldsymbol{\theta}$) explains the observed data. The logarithms of likelihood, the *log likelihood function*, does the same job and is usually preferred for a few reasons:



- The log likelihood function in maximum likelihood estimations is usually computationally simpler
- Likelihoods are often tiny numbers (or large products) which makes them difficult to graph. Taking the natural logarithm results in a better graph with large sums instead of products.
- The log likelihood function is usually (not always!) easier to optimize.

7 Function Approximation

❗ Until further notice we assume that the state space is consisting of at least one continuous quantity or an unfeasible large amount of discrete states (quasi-continuous). The state is considered a vector:

$$s = [s_1\ s_2\ \dots]^{\mathrm{T}}$$

The action space remains discrete and feasible small; actions can be represented as a scalar.

💡 The novelty in this chapter is that the approximate value function is represented not as a table but in a *parameterized functional form* with weight vector $\boldsymbol{w} \in \mathbb{R}^d$. We will write

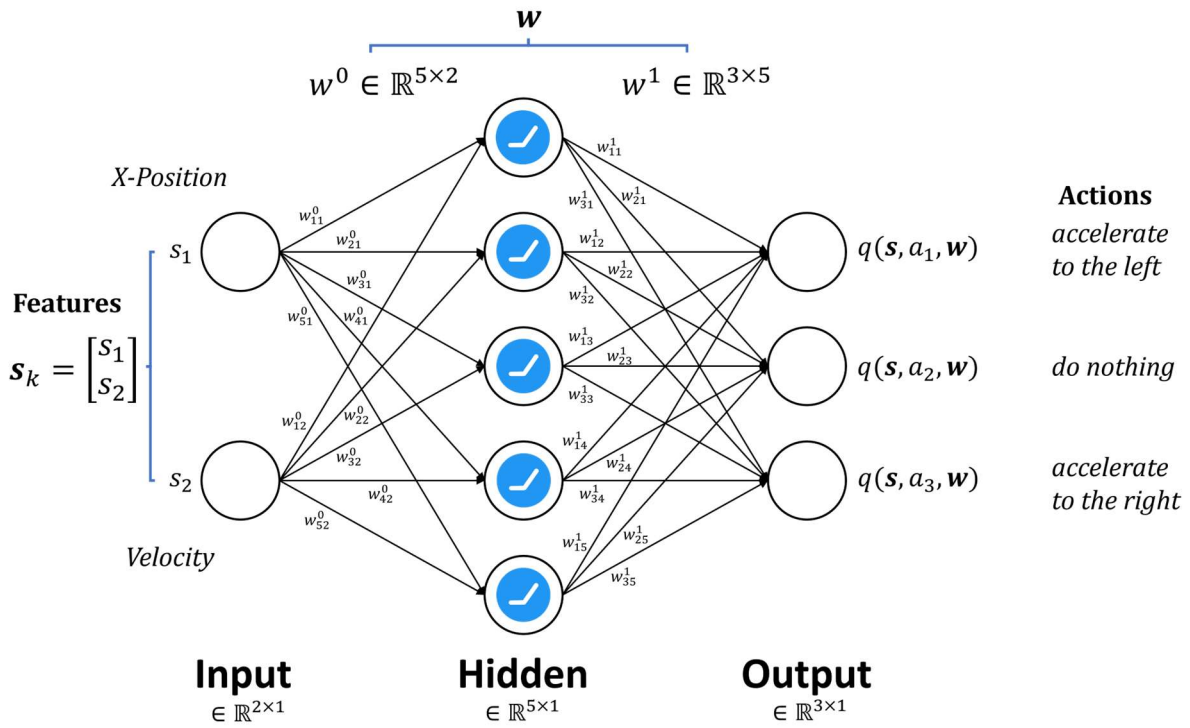
$$\hat{v}(s, \boldsymbol{w}) \approx v_\pi(s)$$

$$\hat{q}(s, a, \boldsymbol{w}) \approx q_\pi(s, a)$$

for the approximate value of state s given weight vector $\boldsymbol{w}$.

More generally, $\hat{v}$ might be the function computed by a multi-layer artificial neural network, with $\boldsymbol{w}$ the vector of connection weights in all the layers. Typically, the number of weights (the dimensionality of $\boldsymbol{w}$) is much less than the number of states ($d \ll |S|$) and changing one weight changes the estimated value of many states.

Example: Mountain Car



Exemplary NN architecture for function approximation. Based on the mountain car environment. State $\boldsymbol{s}_k$ consist of multiple features, here two. Output neurons represent q-values. neurons are implicit.

$$w^0 = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{31} \\ w_{41} & w_{42} \\ w_{51} & w_{52} \end{bmatrix} \qquad w^1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \end{bmatrix}$$

Matrix representation of parameter vector $\boldsymbol{w}$ *for above example.*

Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states. Such generalization makes the learning potentially more powerful but also potentially more difficult to manage and understand.

7.1 Gradient-Based Prediction

7.1.1 Prediction Objective $\overline{\text{VE}}$

We must specify a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error in each state s. By the error in a state s we mean the square of the difference between the approximate value $\hat{v}(\tilde{s}, \boldsymbol{w})$ and the true value $v_\pi(s)$. Weighting this over the state space by μ, we obtain a natural objective function, the mean square value error, denoted $\overline{VE}$:

$$\overline{VE}(\boldsymbol{w}) \approx J(\boldsymbol{w}) = \sum_{s \in \mathcal{S}} \mu(s) * [v_\pi(s) - \hat{v}(\tilde{s}, w)]^2$$

$\sqrt{\overline{VE}}$ gives a rough measure of how much the approximate values differ from the true values and is often used in plots.

Often $\mu(s)$ is chosen to be the *fraction of time spent* in state s. For on-policy training, $\mu(s)$is called the *on-policy distribution*; for prediction we focus entirely on this.

On-Policy Distribution in Episodic Tasks

Let $h(s)$ denote the probability that an episode begins in each state s.

Let $\eta(s)$ denote the number of time steps spent, on average, in s, in a single episode. Time is spent in s if:

- episodes start in s, or
- transitions are made into s from a preceding state $\bar{s}$

$$\eta(s) = \underbrace{h(s)}_{\text{prob. of starting in } s} + \sum_{\bar{s}} \underbrace{\eta(\bar{s})}_{\text{avg. steps in } \bar{s}} \sum_a \underbrace{\pi(a|\bar{s})}_{\substack{\text{action selection} \\ \text{prob.}}} * \underbrace{p(s|\bar{s}, a)}_{\substack{\text{state transition} \\ \text{prob.}}}$$

$$\text{predecessors of } s$$

This system of equations can be solved for the expected number of visits $\eta(s)$. The on-policy distribution $\mu(s)$ is then normalized to sum to one:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}$$

→ It is essentially the probability of being in state s in the long run.

If we would perform off-policy prediction, we have to transform the sampled value (estimates) from the behavior to the target policy. Likewise, when doing this for tabular methods, this increases the prediction variance. In combination with generalization errors due to function approximation, the overall risk of diverging is significantly higher compared to the on-policy case

💡 To summarize, the overall goal is to find:

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w})$$

7.1.2 Gradient Descent

In gradient-descent methods, the weight vector is a column vector with a fixed number of real valued components, $\boldsymbol{w} = [w_1, w_2, \dots, w_d]^{\mathrm{T}}$ and the approximate value function $\hat{v}(\tilde{s}, \boldsymbol{w})$ is a differentiable function of $\boldsymbol{w}$ for all $s \in \mathcal{S}$. Will be updating $\boldsymbol{w}$ at each of a series of time steps t.

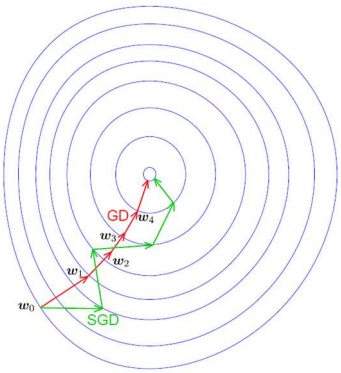
📄 Let $J(\boldsymbol{w})$ be a differentiable function, called the *cost function*, of parameter vector $\boldsymbol{w}$**.** We define the *gradient* of $J(\boldsymbol{w})$ to be the partial derivative of it with respect to each parameter $\mathbf{w}_{\mathrm{n}}$:

$$\nabla J(\boldsymbol{w}) = \left[\frac{\partial J(\boldsymbol{w})}{\partial w_1} \quad \frac{\partial J(\boldsymbol{w})}{\partial w_2} \quad \dots \quad \frac{\partial J(\boldsymbol{w})}{\partial w_n}\right]^{\mathrm{T}}$$

Above vector tells us the direction of steepest descent. Adjusting our parameters $\boldsymbol{w}$ in the direction of the gradient, will show us a local minimum of $J(\boldsymbol{w})$.

7.1.3 Finding the Gradient $\nabla J(w)$

For this task, there are two options. The first is the full calculation of $\nabla J(\boldsymbol{w})$, called *batch gradient* or *regular gradient descent*. This has a few drawbacks as it is computationally expensive, and we'd have to wait until the entire state sequence (in an episodic task) has been obtained. Furthermore, changing the policy in an online/GPI manner changes the representativeness of the batch.



This leaves *stochastic gradient descent* (SGD), where after taking a single step, the update is performed. SGD in expectation (averaging of samples) leads to the same result as regular gradient descent.

7.1.4 Gradient-Based Parameter Update

📄 The incremental learning update per step (for stochastic gradient descent) is:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha[v_\pi(s_k) - \hat{v}(\tilde{s}_k, \boldsymbol{w}_k)] * \nabla_{\mathrm{w}} \hat{v}(\tilde{s}_k, \boldsymbol{w}_k)$$

Gradient descent methods are called "stochastic" when the update is done, as here, only for a single example, which might have been selected stochastically.

💡 In practice, we substitute a target for $v_\pi(s_k)$.

- MC: $\Delta \mathbf{w} = \alpha[G_t - \hat{v}(\tilde{s}_k, \boldsymbol{w}_k)] * \nabla_{\mathrm{w}} \hat{v}(\tilde{s}_k, \boldsymbol{w}_k)$

- `TD(0)` : $\Delta \mathbf{w} = \alpha[r_{k+1} + \gamma \hat{v}(s_{k+1}, \boldsymbol{w}) - \hat{v}(\tilde{s}_k, \boldsymbol{w}_k)] * \nabla_{\mathrm{w}} \hat{v}(\tilde{s}_k, \boldsymbol{w}_k)$

Respectively for control, we substitute a target for $q_\pi(s, a)$:

- MC: $\Delta \mathbf{w} = \alpha[G_t - \hat{q}(\tilde{s}_k, a_k, \boldsymbol{w}_k)] * \nabla_{\mathrm{w}} \hat{q}(\tilde{s}_k, a_k, \boldsymbol{w}_k)$
- `TD(0)`: $\Delta \mathbf{w} = \alpha[r_{k+1} + \gamma \hat{q}(s_{k+1}, a_{k+1}, \boldsymbol{w}) - \hat{q}(\tilde{s}_k, a_k, \boldsymbol{w}_k)] * \nabla_{\mathrm{w}} \hat{q}(\tilde{s}_k, a_k, \boldsymbol{w}_k)$

7.1.5 Gradient MC Prediction

Direct transfer from tabular case to function approximation. The update target becomes the sampled return $v_\pi(\boldsymbol{s_k}) \approx g_k$:

input: a policy π to be evaluated, a feature representation $\tilde{\boldsymbol{s}} = \boldsymbol{f}(\boldsymbol{s})$
input: a differentiable function $\hat{v} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: value-function weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  generate an episode following π: $s_0, a_0, r_1, \ldots, s_T$;
  calculate every-visit return g_k;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g_k - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$;

Every-visit gradient MC

For the differentiable function $\hat{v}$ we can choose any function, such as a linear estimator or a decision tree, that works with the gradient-based parameter update step. We'll mostly use a ANN for it, though.

7.1.6 Semi-Gradient TD(0) Prediction

If bootstrapping is applied, then the true target $v_\pi(\boldsymbol{s}_k)$ is being approximated by a target depending on the estimate $\hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$.

❗ If $\hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$ does not perfectly fit $v_\pi(\boldsymbol{s}_k)$, the update target becomes a *biased estimate* of $v_\pi(\boldsymbol{s}_k)$.

The semi-gradient of $J(\boldsymbol{w})$ for `TD(0)` is then:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \approx -[r_{k+1} - \gamma \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})] \Delta_w \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$$

input: a policy π to be evaluated, a feature representation $\tilde{\boldsymbol{s}} = \boldsymbol{f}(\boldsymbol{s})$
input: a differentiable function $\hat{v} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$ with $\hat{v}(\tilde{\boldsymbol{s}}_T, \cdot) = 0$
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: value-function weights $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2, \ldots$ *episodes* **do**
  initialize $\boldsymbol{s}_0$;
  for $k = 0, 1, 2 \ldots$ *time steps* **do**
    $a_k \leftarrow$ apply action from $\pi(\boldsymbol{s}_k)$;
    observe $\boldsymbol{s}_{k+1}$ and r_{k+1};
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{s}}_{k+1}, \boldsymbol{w}) - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$;
    exit loop if $\boldsymbol{s}_{k+1}$ is terminal;

Semi-gradient `TD(0)`.Note that the state-value of a terminal state $\hat{v}(\tilde{s}_T, \cdot)$ has to be zero otherwise we'd make a systematic error.

When bootstrapping is applied, the gradient does not take into account any gradient component of the bootstrapped target estimate.

7.2 Batch Learning

❗ As already discussed: incremental learning is not data efficient, as during one incremental learning step we are not utilizing the given information to the maximum possible extent. This also applies to SGD-based updates with function approximation.

Batch learning methods provide an alternative:

- Given a fixed, consistent data set $\mathcal{D} = \{\langle \boldsymbol{s}_0, v_\pi(\boldsymbol{s}_0)\rangle, \langle \boldsymbol{s}_1, v_\pi(\boldsymbol{s}_1)\rangle, \ldots\}$, find the optimal parameter vector $\boldsymbol{w}^*$.
- As options we have experience replay and if $\hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$ is linear: closed-form least squares solution

7.2.1 Experience Replay

Based on the data set: $\mathcal{D} = \{\langle \boldsymbol{s}_0, v_\pi(\boldsymbol{s}_0)\rangle, \langle \boldsymbol{s}_1, v_\pi(\boldsymbol{s}_1)\rangle, \ldots\}$

Repeat:

1. Sample uniformly $i = 1, \ldots, b$ state-value pairs from experience (mini batch):

$$\langle \boldsymbol{s}_i, v_\pi(\boldsymbol{s}_i)\rangle \sim \mathcal{D}$$

2. Apply (semi) SGD update step:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \frac{\alpha}{b} \sum_{i=1}^{b} [v_\pi(s_i) - \hat{v}(\tilde{s}_i, \boldsymbol{w}_i)] * \nabla \hat{v}(\tilde{s}_i, \boldsymbol{w}_i)$$

The parameter update $\boldsymbol{w}_{k+1}$ is done by averaging over the b samples of our mini batch.

💡 Note that $\hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$ can be any differentiable function; the usual technical tuning requirements regarding α apply; the true target $v_\pi(\boldsymbol{s})$ is usually approximated by MC or TD targets.

7.2.2 (Ordinary) Least Squares

Assuming the following applies:

- $\hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$ is a linear estimator
- $\boldsymbol{\mathcal{D}}$ is a fixed, representative data set following the on-policy distribution

Then, minimizing the quadratic loss function $J(\boldsymbol{w})$ becomes an ordinary least squares (OLS) or linear regression problem. We focus on the combination of OLS and `TD(0)` (so called LSTD), but the following can be equally extended to n-step learning or MC:

Rewriting $J(\boldsymbol{w})$ using the `TD(0)` target:

$$v_\pi(\boldsymbol{s}_k) \approx r_{k+1} + \gamma \hat{v}(s_{k+1}) = r_{k+1} + \gamma \tilde{s}_{k+1}^{\mathbf{T}} \boldsymbol{w}$$

$$J(\boldsymbol{w}) = \sum_k [v_\pi(\boldsymbol{s}_k) - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})]^2 = \sum_k \left[r_{k+1} - \left(\tilde{s}_k^{\mathbf{T}} - \gamma \tilde{s}_{k+1}^{\mathbf{T}}\right) \boldsymbol{w}\right]^2$$

7.3 Gradient-Based Control (On-Policy)

📄 Transferring the objective $J(\boldsymbol{w})$ from on-policy *prediction* to *control* yields:

$$J(\boldsymbol{w}) = \sum_k [q_\pi(\boldsymbol{s}_k, a_k) - \hat{q}(\tilde{\boldsymbol{s}}_k, a_k, \boldsymbol{w})]^2$$

Analogously, the (semi-)gradient-based parameter update from *7.1.4* is also applied to action values:

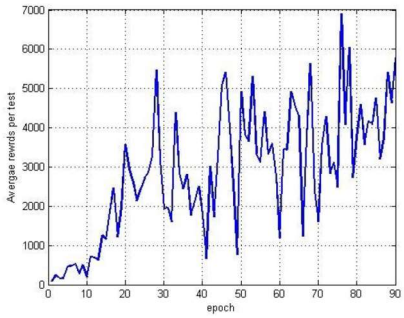
$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha [q_\pi(\boldsymbol{s}_k, a_k) - \hat{q}(\tilde{\boldsymbol{s}}_k, a_k, \boldsymbol{w})] * \nabla_{\mathbf{w}} \hat{q}(\tilde{\boldsymbol{s}}_k, a_k, \boldsymbol{w})$$

Depending on the control approach, the true target $q_\pi(\boldsymbol{s}_k, a_k)$ is approximated by:

- MC: full episodic return $q_\pi(\boldsymbol{s}_k, a_k) \approx g$
- Sarsa: one-step bootstrapped estimate $q_\pi(\boldsymbol{s}_k, a_k) \approx r_{k+1} + \gamma \hat{q}(\tilde{\boldsymbol{s}}_{k+1}, a_{k+1}, \boldsymbol{w}_k)$

7.3.1 Problems with GPI

❗ The policy improvement theorem for tabular methods guaranteed to find a globally better or equally good policy in each update step (cf. *3.2 Policy Improvement*). With parameter updates generalization applies; hence, when reacting to one specific state-action transition other parts of the state-action space within $\hat{q}$ are affected too. Therefore, the theorem is not applicable with function approximation.



Learning curves have drastic performance dips when applying function approximation.

7.3.2 Gradient MC Control

Here, the update target becomes the sampled return $q_\pi(s_k, a_k) \approx g_k$. If we're operating ε-greedy on $\hat{q}$, the baseline policy (given by $\boldsymbol{w}_0$) must guarantee to (successfully) terminate the episode! Otherwise $\hat{q}$ would never be updated.

input: a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$
input: a policy π (only if estimating q_π)
parameter: step size $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  generate episode following π or ε-greedy on $\hat{q}$: $s_0, a_0, r_1, \ldots, s_T$;
  calculate every-visit return g_k;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g_k - \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})$;

Every-visit gradient MC-based action-value estimation. Output is the p.-vector $\boldsymbol{w}$ *for* $\hat{q}_\pi$ *or* $\hat{q}^*$

7.3.3 Semi-Gradient Sarsa Control

input: a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$
input: a policy π (only if estimating q_π)
parameter: step size $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily
for $j = 1, 2, \ldots$ *episodes* **do**
  initialize $\boldsymbol{s}_0$;
  for $k = 0, 1, 2 \ldots$ *time steps* **do**
    $u_k \leftarrow$ apply action from $\pi(\boldsymbol{s}_k)$ or ε-greedy on $\hat{q}(\boldsymbol{s}_k, \cdot, \boldsymbol{w})$;
    observe $\boldsymbol{s}_{k+1}$ and r_{k+1};
    if $\boldsymbol{s}_{k+1}$ *is terminal* **then**
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} - \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})$;
      go to next episode;
    choose u' from $\pi(\boldsymbol{s}_{k+1})$ or ε-greedy on $\hat{q}(\boldsymbol{s}_{k+1}, \cdot, \boldsymbol{w})$;
    $\boldsymbol{w} \leftarrow$
      $\boldsymbol{w} + \alpha \left[r_{k+1} + \gamma \hat{q}(\boldsymbol{s}_{k+1}, a', \boldsymbol{w}) - \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})$;

Semi-gradient Sarsa action-value estimation. Output is the p.-vector $\boldsymbol{w}$ *for* $\hat{q}_\pi$ *or* $\hat{q}^*$

7.4 Deep Q-Networks (DQNs)

Recall the incremental learning step from tabular Q-learning:

$$\hat{q}(s,a) \leftarrow \hat{q}(s,a) + \alpha\left[r + \gamma \max_a \hat{q}(s',a) - \hat{q}(s,a)\right]$$

DQNs transfer this to an approximate solution:

$$\boldsymbol{w} = \boldsymbol{w} + \alpha\left[r + \gamma \max_a \hat{q}(\boldsymbol{s}',a,\boldsymbol{w}) - \hat{q}(\boldsymbol{s},a,\boldsymbol{w})\right]\nabla_{\boldsymbol{w}}\hat{q}(s,a,\boldsymbol{w})$$

Each part plays a distinct role:

- $\left[r + \gamma \max_a \hat{q}(\boldsymbol{s}',a,\boldsymbol{w}) - \hat{q}(s,a,\boldsymbol{w})\right]$: This is the TD error, which measures the difference between the estimated Q-value of the current state-action pair, $\hat{q}(s,a,\boldsymbol{w})$, and the TD target, $r + \gamma \max_a \hat{q}(\boldsymbol{s}',a,\boldsymbol{w})$. The TD target is the estimated Q-value of the best possible action at the next state, discounted by the factor γ, plus the immediate reward.

→ This TD error determines the magnitude of the update to the weights.

- $\nabla_{\boldsymbol{w}}\hat{q}(s,a,\boldsymbol{w})$: This is the gradient of the Q-value with respect to the weights, calculated at the current state-action pair.

→ This gradient determines the direction of the update to the weights.

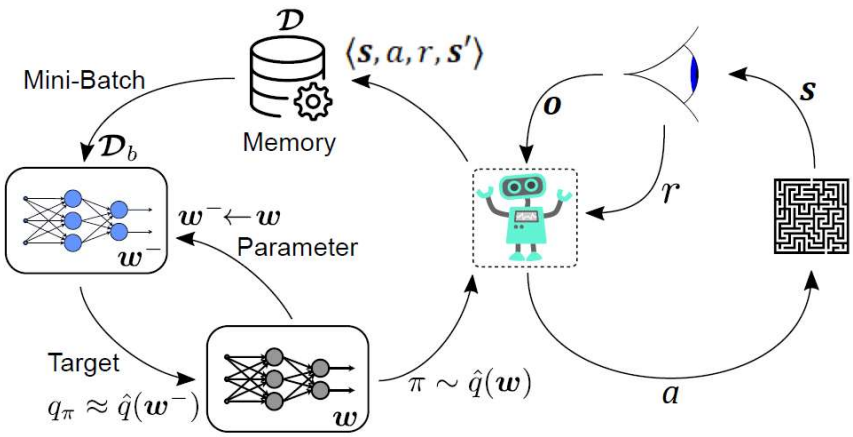
In essence, the weights are updated in the direction that would make the estimated Q-value of the current state-action pair closer to the TD target, and the size of the update is proportional to the TD error. The learning rate parameter α controls the step size of each update.

This formula effectively implements a form of Gradient Descent, where the goal is to minimize the difference between the estimated Q-values and the TD targets. By repeatedly adjusting the weights in this way over many experiences, the network learns to estimate the Q-values that lead to the highest cumulative future reward.

However, instead of using semi-gradient step-by-step updates, DQNs are characterized by:

- an *experience replay buffer* for batch learning (cf. *7.2.1* and *7.4.1*)
- a separate set of weights $\boldsymbol{w}^-$ for the bootstrapped Q-target (cf. ***Error! Reference source not found.***)

This allows to efficiently use available data through *experience replay* and stabilizes learning by trying to make targets and feature inputs more like i.i.d. data from a stationary process.



7.4.1 Experience Replay Buffer

A key reason for using this technique is to break the strong correlation between consecutive samples. If the network learned only from consecutive experience samples that occurred sequentially in the environment, the samples would be highly correlated which would lead to inefficient learning (target value changes). Taking random samples from replay memory breaks this correlation.

- We store experiences $\langle \boldsymbol{s}, a, r, \boldsymbol{s}' \rangle$ in the replay buffer $\mathcal{D}$ after each transition step
- The replay buffer $\mathcal{D}$ is of limited capacity, i.e., once it is full, it discards the oldest sample when updating (ring memory).
- This allows us to improve the Q-learning critic minimizing the *mean-squared Bellman error* (MSBE) :

$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma \max_a \hat{q}(\boldsymbol{s}', a, \boldsymbol{w})\right) - \hat{q}(\boldsymbol{s}, a, \boldsymbol{w})\right]^2_{\boldsymbol{D}_b}$$

7.4.2 (Fixed) Target Networks

When calculating the loss, we perform two passes through the DQN network using the sample obtained from the buffer. The first pass is used to calculate $q(\boldsymbol{s}, a, \boldsymbol{w})$, the q-value for the relevant action from the sample. Then we do the second pass with the successor state to calculate $\hat{q}(\boldsymbol{s}', a, \boldsymbol{w})$, the *target q-value*.

- Pass 1 (Current Q-Value): Feed the current state $\boldsymbol{s}$ through the network to get the Q-values for all actions. Then select the Q-value of the action a taken according to the stored experience in the buffer. This gives us $\hat{q}(s, a, \boldsymbol{w})$, where θ represents the current parameters of the network.
- Pass 2 (TD-Target): Next, feed the successor state $\boldsymbol{s}'$ through to get the Q-values for all actions. Pick the *maximum* Q-value among these. This is the Q-value of the best action at the successor state. The TD-Target is then computed as $r + \gamma \max_a \hat{q}(\boldsymbol{s}', a, \boldsymbol{w})$, where r is the immediate reward from the experience replay buffer.

If we were using the same network for these two passes, we'd update the weights in our learning step in a way that:

- moves our estimated $\hat{q}(s, a, \boldsymbol{w})$ values closer to our target $\max_a \hat{q}(\boldsymbol{s}', a, \boldsymbol{w})$
- moves our target $\max_a \hat{q}(\boldsymbol{s}', a, \boldsymbol{w})$ the same way as our estimated $\hat{q}(s, a, \boldsymbol{w})$ values

$$\hat{q}(\boldsymbol{s}, a, \boldsymbol{w}) \xrightarrow{\text{Direction of } \nabla_w \hat{q}(\boldsymbol{s}, a, \boldsymbol{w})} \max_a \hat{q}(\boldsymbol{s}', a, \boldsymbol{w}) \xrightarrow{\text{Direction of } \nabla_w \hat{q}(\boldsymbol{s}, a, \boldsymbol{w})}$$

This makes the optimization "chase its own tail" which introduces instability.

To solve this problem, we use different networks for DQN:

- Policy Network $\boldsymbol{w}$ for calculating $\hat{q}(s, a, \boldsymbol{w})$
- Target Network $\boldsymbol{w}^-$ for calculating $r + \gamma \max_a \hat{q}(\boldsymbol{s}', a, \boldsymbol{w}^-)$

The Target Network $\boldsymbol{w}^-$ is a clone of the Policy Network $\boldsymbol{w}$ that has its weights *frozen*. Based on a update frequency parameter $\boldsymbol{w}^-$ gets updated periodically to $\boldsymbol{w}^- = \boldsymbol{w}$.

7.4.3 Working Principle

repeat:

1. take actions a based on $\hat{q}(s, a, \boldsymbol{w})$ (e.g., ϵ-greedy)
2. store observed tuples $\langle \boldsymbol{s}, a, r, \boldsymbol{s}' \rangle$ in memory buffer $\boldsymbol{\mathcal{D}}$
3. Sample mini batches $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$
4. Calculate bootstrapped Q-target with a delayed parameter vector $\boldsymbol{w}^-$ (target network)
$$q_\pi(\boldsymbol{s}, a) \approx r + \gamma \max_a \hat{q}(\boldsymbol{s}', a, \boldsymbol{w}^-)$$
5. Optimize MSE loss between targets and the regular approximation $\hat{q}(\boldsymbol{s}', a, \boldsymbol{w})$ using $\boldsymbol{\mathcal{D}}_b$
$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma \max_a \hat{q}(\boldsymbol{s}', a, \boldsymbol{w}^-)\right) - \hat{q}(\boldsymbol{s}, a, \boldsymbol{w})\right]^2_{\boldsymbol{D}_b}$$

(6.) Update $\boldsymbol{w}^-$ based on $\boldsymbol{w}$ from time to time

7.4.4 Algorithm

input: a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$ (including feature eng.)
parameter: $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, update factor $k_w \in \{\mathbb{N} | 1 \leq k_w\}$
init: weights $\boldsymbol{w} = \boldsymbol{w}^- \in \mathbb{R}^\zeta$ arbitrarily, memory $\boldsymbol{\mathcal{D}}$ with certain capacity
for $j = 1, 2, \ldots$ *episodes* **do**
  initialize $\boldsymbol{s}_0$;
  for $k = 0, 1, 2 \ldots$ *time steps* **do**
    $a_k \leftarrow$ apply action ε-greedy w.r.t $\hat{q}(\boldsymbol{s}_k, \cdot, \boldsymbol{w})$;
    observe $\boldsymbol{s}_{k+1}$ and r_{k+1};
    store tuple $\langle \boldsymbol{s}_k, a_k, r_{k+1}, \boldsymbol{s}_{k+1} \rangle$ in $\boldsymbol{\mathcal{D}}$;
    sample mini-batch $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$ (after initial memory warmup);
    for $i = 1, \ldots, b$ *samples* **do** calculate Q-targets
      if $\boldsymbol{s}_{i+1}$ *is terminal* **then** $y_i = r_{i+1}$;
      else $y_i = r_{i+1} + \gamma \max_a \hat{q}(\boldsymbol{s}_{i+1}, a, \boldsymbol{w}^-)$;
    fit $\boldsymbol{w}$ on loss $\mathcal{L}(\boldsymbol{w}) = [y_i - \hat{q}(\boldsymbol{s}_i, a_i, \boldsymbol{w})]^2_{\boldsymbol{\mathcal{D}}_b}$;
    if $k \mod k_w = 0$ **then** $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$ (update target weights);

Remarks

Often 'deep' artificial neural networks are used as function approximation for DQN. Nevertheless, other model topologies are fully conceivable.

The fit of $\boldsymbol{w}$ on loss $\mathcal{L}$ is an intermediate supervised learning step.

- Comes with degrees of freedom regarding solver choice.
- Has own optimization parameters

Mini-batch sampling from $\mathcal{D}$ is often randomly distributed. Nevertheless, guided sampling with useful distributions for a specific control task can be beneficial.

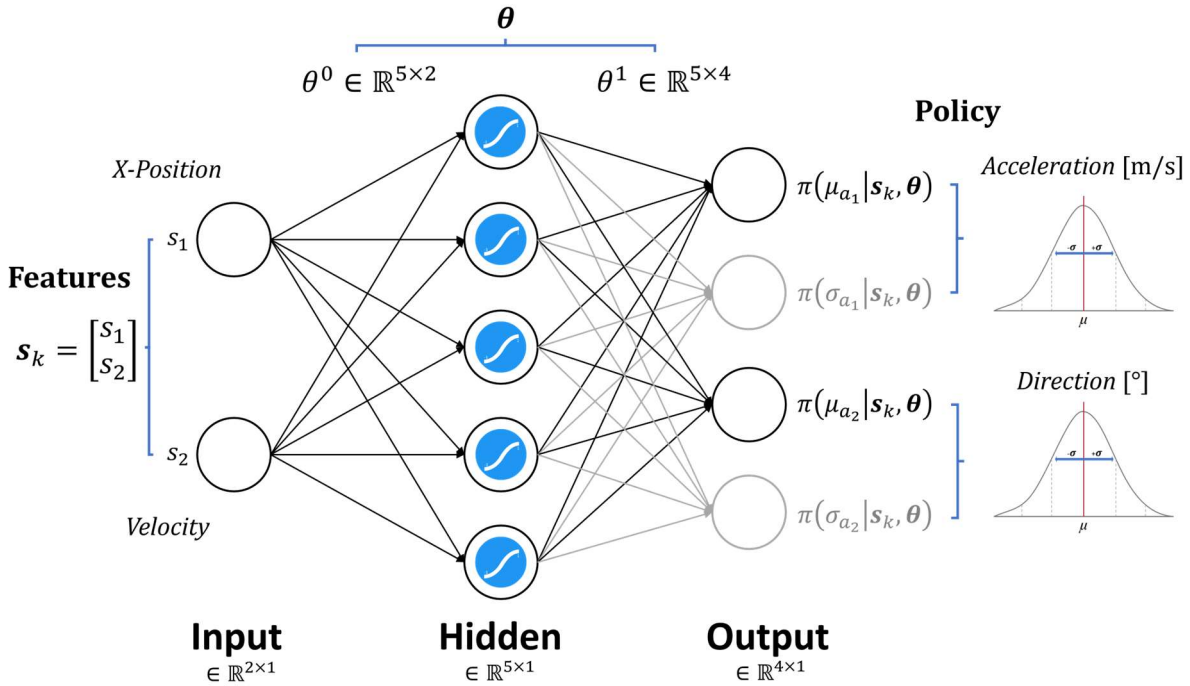
Likewise, the simple ε-greedy approach can be extended. Often a scheduled/annealed trajectory ε_k is used.

8 Policy Gradient Methods

Introduction

💡 So far, almost all the methods have been *action-value methods*; they learned the values of actions and then selected actions based on their estimated action values. Their policies would not even exist without the action-value estimates. In this chapter we consider methods that instead learn a *parameterized policy* that can select actions without consulting a value function. A value function may still be used to learn the *policy parameter* but is not required for action selection.

Example



*This example architecture shows how a **stochastic policy** is given by the ANN. The resulting μ and σ parameters can be used to model a gaussian distribution.*

On the x-axis of this function is the actual value of the action to execute while the y-axis corresponds to the probability of picking such action. To interact with the environment, actions then can be sampled from the distribution.

We write the following for the probability that action a is taken at time k given that the environment is in state s at time k with parameter $\boldsymbol{\theta}$.

$$\pi(a|s) = \mathbb{P}[A_k = a|S_k = s] \approx \pi(a|s, \boldsymbol{\theta})$$

We denote $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ for the policy's *parameter vector* and if a method uses a learned value function as well, then the value function's weight vector is denoted $\boldsymbol{w} \in \mathbb{R}^{d}$.

💡 Remember:

- $J(\boldsymbol{\theta})$ is the performance measure for the policy $\pi_{\boldsymbol{\theta}}$
- $\nabla J(\boldsymbol{\theta})$ a column vector containing all partial derivatives of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$

8.1 Action Space

8.1.1 SoftMax Action Preferences (Discrete & Stochastic)

If the action space is discrete and not too large, then a natural and common kind of parametrization is to form parameterized numerical preferences $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ for each state-action pair. The actions with the highest preferences are given the highest probabilities of being selected, for example, according to an exponential SoftMax distribution.

Soft-Max Action Preferences

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}$$

Example

a	$h(s, a, \boldsymbol{\theta})$	$\pi(a\|s, \boldsymbol{\theta})$
A	1	0,04
B	2	0,11
C	4	0,81
D	1	0,04

The denominator here is just what is required so that the action probabilities in each state sum to one. This kind of policy parameterization is called soft-max in action preferences.

The action preferences themselves can be parameterized arbitrarily. For example, they might be computed by a deep artificial neural network, where $\boldsymbol{\theta}$ is the vector of all the connection weights of the network.

Advantages

One advantage of parameterizing policies as above is that the approximate policy can approach a deterministic policy, whereas with ϵ-greedy action selection over action values there is always the probability of selecting a random action.

One could select according to a soft-max distribution based on action values $\hat{q}$, but this alone would not allow the policy to approach a deterministic policy. Instead, the action-value estimates would simply converge to their corresponding true values q, which would differ by a finite amount, translating to specific probabilities other than 0 and 1.

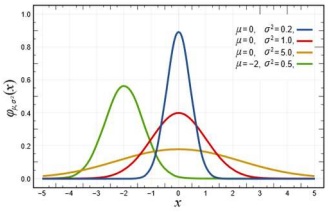
Parameterizing policies according to the *soft-max in action preferences* enables the selection of actions with arbitrary probabilities. In some problems, such as rock-paper-scissors, the best approximate policy may be stochastic. Action-value methods have no natural way of finding stochastic optimal policies, whereas policy approximating methods can.

Perhaps the simplest advantage that policy parameterization may have over action-value parameterization is that the policy may be a simpler function to approximate.

8.1.2 Gaussian Probability Density (Continuous & Stochastic)

Let's consider the case when the action space is continuous and there is only one scalar action $a \in \mathbb{R}$. A typical policy function is the *gaussian probability density*:

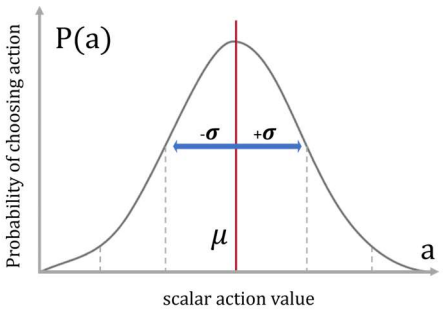
$$f(x) = \frac{1}{\sigma * \sqrt{2\pi}} * e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$



The function $f(x)$ yields a standard bell curve shape. The parameters μ and σ determine the expected value and variance.

By using neural networks, we can approximate these values ($\mu(s, \boldsymbol{\theta})$ and $\sigma(s, \boldsymbol{\theta})$.

If there are multiple, n actions, a typical policy function would be *multivariate gaussian probability density*. This is essentially the same as above but in n-dimensional space.



8.2 Policy Gradient Theorem

Policy Objective Function

The performance measure for the policy $\pi_{\boldsymbol{\theta}}$ is given by $J(\boldsymbol{\theta})$. It measures how good an agent is doing. Depending on whether we're considering the episodic or continuous case, it is defined differently:

Episodic	Continuous
$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{s}_0) = \mathbb{E}[v\|\boldsymbol{S} = \boldsymbol{s}_0, \boldsymbol{\theta}]$	$J(\boldsymbol{\theta}) = \bar{r}_{\pi_{\boldsymbol{\theta}}} = \int_{\mathcal{S}} \mu_\pi(s) \int_{\mathcal{A}} \pi(\boldsymbol{a}\|\boldsymbol{s}, \boldsymbol{\theta}) \int_{S,\mathcal{R}} p(\boldsymbol{s}', r\|\boldsymbol{s}, \boldsymbol{a}) * r$
Average total return per episode / State value of starting state	Average reward per step

Motivation

💡 We want to find $\boldsymbol{\theta}$ that maximizes $J(\boldsymbol{\theta})$. For this, we consider methods for learning $\boldsymbol{\theta}$ based on the gradient of $J(\boldsymbol{\theta})$. Similarly to function approximation, we define the *gradient* of $J(\boldsymbol{\theta})$ to be the partial derivative of it with respect to each parameter θ_{n}:

$$\nabla_{\theta} J(\boldsymbol{w}) = \left[\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} \quad \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_2} \quad \dots \quad \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_n}\right]^{\mathrm{T}}$$

These methods seek to maximize performance, so their updates approximate *gradient ascent*. In other words: by using gradient ascent we can find the best $\boldsymbol{\theta}$ that produces the highest return:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \widehat{\nabla J(\boldsymbol{\theta}_k)}$$

Where $\alpha \widehat{\nabla J(\boldsymbol{\theta}_k)} \in \mathbb{R}^{d'}$ is a stochastic estimate, whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_k$. We use a stochastic estimate, as the true gradient is not feasible to calculate. We obtain the gradient through the *policy gradient theorem*.

All methods that follow this general schema are called *policy gradient methods*, whether or not they also learn an approx. value function.

Policy Gradient Theorem

Given a metric $J(\boldsymbol{\theta})$ for the undiscounted episodic or continuing tasks (cf. *Policy Objective Function*) and a parameterizable policy $\pi(a|s, \boldsymbol{\theta})$ the policy gradient is:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \mathbb{E}_{\pi}\left[q_{\pi}(\boldsymbol{s},\boldsymbol{a}) * \frac{\nabla_{\boldsymbol{\theta}}\,\pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})}{\pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})}\right]$$

which can be rewritten to (cf. *Log Derivative Trick*):

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_{\pi}[q_{\pi}(\boldsymbol{s},\boldsymbol{a}) * \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})]$$

with $\nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})$, also called the *score function*, that can also be sampled. This gradient indicates the direction in which the parameters $\boldsymbol{\theta}$ should be adjusted to increase the log likelihood of the action that was taken, under the current policy.

Log Derivative Trick

Remember the derivation of the logarithm (cf. *7.4 Logarithm*):

$$\frac{\partial}{\partial x}\ln(x) = \frac{1}{x} \qquad \frac{\partial}{\partial x}\ln\big(f(x)\big) = \frac{1}{f(x)} * f'(x) \qquad \nabla\ln(f) = \frac{\nabla f}{f}$$

This allows us to transform this

$$\frac{\nabla\pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})}{\pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})} = \nabla \ln \pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})$$

Intuitive Interpretation

Inserting the policy gradient theorem into gradient ascent:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha\widehat{\nabla J(\boldsymbol{\theta}_k)}$$

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha\,\mathbb{E}_{\pi}\left[q_{\pi}(\boldsymbol{s},\boldsymbol{a}) * \frac{\nabla_{\boldsymbol{\theta}}\,\pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})}{\pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})}\right]$$

Blue: Move in the direction that favor actions that yield an increased value

Orange: Scale the update step size inversely to the action probability to compensate that some actions are selected more frequently

Value Based Solutions	Policy Based Solutions
Estimated value is an intuitive performance metric	Policy gradient theorem provides strong convergence properties.
Considered sample-efficient (cf. replay buffer or bootstrapping)	Seamless integration of stochastic and dynamic policies
	Straightforward applicable to large/continuous action spaces. In contrast, value-based approaches would require explicit optimization.

❗ Gradient-based optimization with (non-linear) function approximation is likely to deliver only suboptimal and local policy optima.

8.3 Monte Carlo Gradient (REINFORCE)

As of now, the score function $\nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})$ can be calculated analytically, but how do we retrieve $q_\pi(\boldsymbol{s},\boldsymbol{a})$? For MC policy gradient, we can use the episodic return g_k to *approximate* $q_\pi(\boldsymbol{s},\boldsymbol{a})$:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha\, \gamma^k g_k\, \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}_k|\boldsymbol{s}_k,\boldsymbol{\theta}_k)$$

This is known as the *REINFORCE* approach. Using the sampled return provides an unbiased estimate but also yields a high variance leading to slow learning!

8.3.1 Baseline

We want to reduce the variance of the learning process while not affecting the expectation of the policy gradient. For this we introduce the baseline $b(\boldsymbol{s})$ as a comparison term.

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_\pi\big[\big(q_\pi(\boldsymbol{s},\boldsymbol{a}) - b(\boldsymbol{s})\big) * \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})\big]$$

Proof: In general, this is true:

$$\begin{aligned}
&\mathbb{E}[b * \nabla \ln \pi(A_t|S_t)] = \mathbb{E}\left[b * \sum_a \pi(a|S_t) * \nabla \ln \pi(a|S_t)\right] \\
&= \mathbb{E}\left[b * \sum_a \pi(a|S_t) * \frac{\nabla \pi(a|S_t)}{\pi(a|S_t)}\right] \\
&= \mathbb{E}\left[b * \sum_a \nabla \pi(a|S_t)\right] \\
&= \mathbb{E}\left[b * \nabla \sum_a \pi(a|S_t)\right] \\
&= \mathbb{E}[b \nabla 1] = \mathbb{E}[b * 0] \\
&= 0
\end{aligned}$$

If b does not depend on the action (can depend on the state).

💡 This implies, that we can subtract a *baseline* to reduce the variance without changing the expected value of the update.

Consequently, the Monte Carlo policy parameter update yields:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha\, \gamma^k \big(g_k - b(s_k)\big)\, \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}_k|\boldsymbol{s}_k,\boldsymbol{\theta}_k)$$

8.3.2 Advantage Function

An intuitive choice of the baseline is the state value $b(\boldsymbol{s}) = v_\pi(\boldsymbol{s})$ which results in

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_\pi\big[\big(q_\pi(\boldsymbol{s},\boldsymbol{a}) - v_\pi(\boldsymbol{s})\big) * \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})\big]$$

Here, the difference between action and state value is the *advantage function*:

$$a_\pi(\boldsymbol{s},\boldsymbol{a}) = q_\pi(\boldsymbol{s},\boldsymbol{a}) - v_\pi(\boldsymbol{s})$$

Interpretation: The advantage function quantifies how much better an arbitrary action $\boldsymbol{a}$ is compared to the average action (choosing $\boldsymbol{a} \sim \pi$). for a particular state.

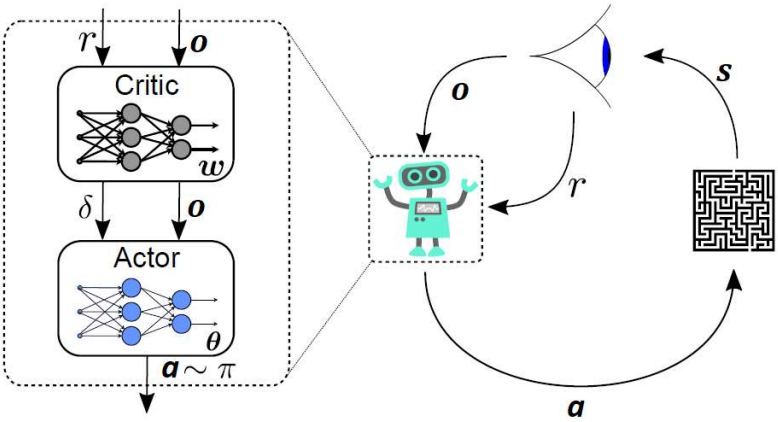
Rewriting results in:

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}\left[\left(a_{\pi}(\boldsymbol{s}, \boldsymbol{a})\right) * \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}|\boldsymbol{s}, \boldsymbol{\theta})\right]$$

For *MC Policy Gradient with Baseline*, this implies learning two parameter sets $\boldsymbol{\theta}$ and $\boldsymbol{w}$, one for the policy and one for the value function $b(\boldsymbol{s}) = v_{\pi}(\boldsymbol{s}, \boldsymbol{w})$.

8.4 Actor-Critic Methods

Methods that learn approximations to *both* policy and value functions are often called *actor-critic methods*, where actor is a reference to the learned policy, and critic refers to the learned value function, usually a state-value function.



Critic (policy evaluation) and actor (policy improvement) can be considered another form of generalized policy iteration (GPI).

This is an online and on-policy algorithm for discrete and continuous action spaces with built-in exploration by stochastic policy functions.

8.4.1 Idea / Derivation

While *MC w/ baseline* learns an unbiased policy it learns slowly due to high variance. An alternative is using an additional function approximator, the so-called critic, to estimate q_{π} (i.e., approximate policy gradient):

$$a_{\pi}(\boldsymbol{s}, \boldsymbol{a}) = q_{\pi}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{w}_q) - v_{\pi}(\boldsymbol{s}, \boldsymbol{w}_v)$$

This would mean we'd have to train another parameter set. Luckily, there is a workaround regarding the value estimation:

The *TD Error* (cf. *5.1.1*) is measuring the difference between the estimated value of s_k and the better estimate $r_{k+1} + \gamma \hat{v}(s_{k+1})$. This quantity, called the *TD error*, arises in various forms throughout reinforcement learning:

$$\delta_{\pi} = r + \gamma v_{\pi}(\boldsymbol{s}') - v_{\pi}(\boldsymbol{s})$$

In expectation, the TD error is equivalent to the advantage function:

$$\begin{aligned}\mathbb{E}_\pi[\delta_\pi|\boldsymbol{s},\boldsymbol{a}] &= \mathbb{E}_\pi[r+\gamma v_\pi(\boldsymbol{s}')|\boldsymbol{s},\boldsymbol{a}] - v_\pi(\boldsymbol{s}) \\ &= q_\pi(\boldsymbol{s},\boldsymbol{a}) - v_\pi(\boldsymbol{s}) \\ &= a_\pi(\boldsymbol{s},\boldsymbol{a})\end{aligned}$$

Hence, the TD error ca be used to calculate the policy gradient:

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_\pi[\delta_\pi * \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})]$$

which results in requiring only one function parameter set:

$$\delta_\pi \approx r + \gamma \hat{v}_\pi(\boldsymbol{s}',\boldsymbol{w}) - \hat{v}_\pi(\boldsymbol{s},\boldsymbol{w})$$

8.4.2 Algorithm: Actor-Critic with TD(0) Targets

input: a differentiable policy function $\pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})$
input: a differentiable state-value function $\hat{v}(\boldsymbol{s},\boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha < 1\}$
init: parameter vectors $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{s}_0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    apply $\boldsymbol{a}_k \sim \pi(\cdot|\boldsymbol{s}_k,\boldsymbol{\theta})$ and observe $\boldsymbol{s}_{k+1}$ and r_{k+1};
    $\delta \leftarrow r_{k+1} + \gamma\hat{v}(\boldsymbol{s}_{k+1},\boldsymbol{w}) - \hat{v}(\boldsymbol{s}_k,\boldsymbol{w})$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_w \delta \nabla_{\boldsymbol{w}} \hat{v}(\boldsymbol{s}_k,\boldsymbol{w})$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \gamma^k \delta \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}_k|\boldsymbol{s}_k,\boldsymbol{\theta})$;

8.5 Deterministic Policy Gradient

So far, we assumed stochastic policies. The resulting on-policy algorithms may not provide best learning performance. The alternative is to apply a deterministic policy with separate exploration or enable off-policy learning with experience replay.

In the subsequent section we will focus on a *deterministic policy function* $\mu(\boldsymbol{s},\boldsymbol{\theta})$.

8.5.1 Deterministic Policy Gradient Theorem

Given a metric $J(\boldsymbol{\theta})$ for the undiscounted episodic or continuing tasks (cf. *Policy Objective Function*) and a parameterizable policy $\mu(\boldsymbol{s},\boldsymbol{\theta})$ the deterministic policy gradient is:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\mu\left[\nabla_{\boldsymbol{\theta}}\mu(\boldsymbol{s},\boldsymbol{\theta})\, \nabla_{\boldsymbol{a}} q(\boldsymbol{s},\boldsymbol{a})|_{\boldsymbol{a}=\mu(\boldsymbol{s})}\right]$$

Again, q needs to be approximated using samples, e.g., implementing citic via TD learning. The bias problem applies as in the stochastic case and can be compensated using compatible function approximation with respect to $\hat{q}$.

The DPGT is also (approximately) valid in the off-policy case, i.e., if the sample distribution is obtained from a behavior policy.

Exploration

❗ DPG methods output a single, best estimated action for any given state. When these methods are used in an on-policy setting, the agent only learns from the actions that are suggested by its current policy, without any built-in mechanism for trying out uncertain or potentially suboptimal actions. In this sense, there is no inherent exploration, as the agent doesn't seek out novel states or actions that might have higher long-term value.

On the other hand, stochastic policies, which output a probability distribution over actions, have inherent exploration built in because they naturally induce a randomness in the actions taken. Even in an on-policy setting, the agent will sometimes take actions that are not currently estimated to be optimal, thereby allowing for exploration.

How can we explore then?

- Inherent noise by the environment (random impacts, measurement noise)
- Add noise to the actions → off-policy

Ornstein-Uhlenbeck (OU) process

The OU process can be used to add noise to the actions taken by an agent, which encourages exploration. By adding OU noise to the actions chosen by a deterministic policy, you can induce the agent to try out a wider variety of actions than it would otherwise. The mean-reverting nature of the OU process ensures that the noise doesn't keep increasing or decreasing without bound, but rather, it stays centered around the mean.

Specifically, in continuous action spaces, Gaussian noise can lead to too much exploration or randomness in actions. In contrast, OU noise is temporally correlated, meaning it varies more smoothly over time, making it more suitable for such tasks where the actions need to be consistent and smooth, yet still exploratory.

$$v_{k+1} = \lambda v_k + \sigma \epsilon_k$$

where v_k is the OI noise output, $0 < \lambda < 1$ is a smoothing factor and σ is the variance scaling a standard gaussian sequence ϵ_k (no mean).

8.5.2 Deterministic Actor-Critic

input: a differentiable deterministic policy function $\mu(\boldsymbol{s}, \boldsymbol{\theta})$
input: a differentiable action-value function $\hat{q}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha < 1\}$
init: parameter vectors $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{s}_0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{a}_k \leftarrow$ apply from $\mu(\boldsymbol{s}_k, \boldsymbol{\theta})$ w/wo noise or from behavior policy;
    observe $\boldsymbol{s}_{k+1}$ and r_{k+1};
    choose $\boldsymbol{a}'$ from $\mu(\boldsymbol{s}_{k+1}, \boldsymbol{\theta})$;
    $\delta \leftarrow r_{k+1} + \gamma \hat{q}(\boldsymbol{s}_{k+1}, \boldsymbol{a}', \boldsymbol{w}) - \hat{q}(\boldsymbol{s}_k, \boldsymbol{a}_k, \boldsymbol{w})$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_w \delta \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{s}_k, \boldsymbol{a}_k, \boldsymbol{w})$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \gamma^k \nabla_{\boldsymbol{\theta}} \mu(\boldsymbol{s}_k, \boldsymbol{\theta}) \nabla_{\boldsymbol{a}} \hat{q}(\boldsymbol{s}_k, \boldsymbol{a}_k, \boldsymbol{w})|_{\boldsymbol{a}=\mu(\boldsymbol{s})}$;

Algo. 1.7: Deterministic actor-critic for episodic tasks using Sarsa(0) targets applicable on- and off-policy (output: parameter vector $\boldsymbol{\theta}^*$ for $\mu^*(\boldsymbol{s}, \boldsymbol{\theta}^*)$) and $\boldsymbol{w}^*$ for $\hat{q}^*(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{w}^*)$)

8.6 Deep Deterministic Policy Gradient (DDPG)

DQNs are not directly applicable to (quasi-) continuous action spaces. The DDPG wants to extend the successes of DQN to continuous action spaces. Recall the Q-Learning equation using FA:

$$\boldsymbol{w} = \boldsymbol{w} + \alpha \left[r + \gamma \max_{a} \hat{q}(\boldsymbol{s}', a, \boldsymbol{w}) - \hat{q}(\boldsymbol{s}, a, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{s}, a, \boldsymbol{w})$$

❗ In the *Q-learning target* we're searching for highest possible q-value estimate of the successor state $\boldsymbol{s}'$. For a discrete set of action this maximization is straightforward. Extending this idea to the continuous action space requires an own *optimization routine*, which is computationally expensive for nonlinear function approximation.

Deterministic Policy Trick

When using a greedy, deterministic policy $\mu(\boldsymbol{s}, \boldsymbol{\theta})$ we can approximate this maximization:

$$\max_{\boldsymbol{a}} \hat{q}(\boldsymbol{s}', \boldsymbol{a}, \boldsymbol{w}) \approx \hat{q}(\boldsymbol{s}', \mu(\boldsymbol{s}', \boldsymbol{\theta}), \boldsymbol{w})$$

Hence, we can obtain explicit Q-learning targets for continuous actions when using a deterministic policy.

For improving the policy, we reuse the *Deterministic Policy Gradient Theorem* (cf. *8.5.1*) in an off-policy fashion:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_b[\nabla_{\boldsymbol{\theta}} \mu(\boldsymbol{S}, \boldsymbol{\theta}) \, \nabla_{\boldsymbol{a}} q(\boldsymbol{S}, \boldsymbol{A}) | \boldsymbol{A} = \mu(\boldsymbol{S}, \boldsymbol{\theta})]$$

given a behavior policy $b(\boldsymbol{a}|\boldsymbol{s})$.

💡 Hence, we can consider the DDPG approach as a combination of DQN and DPG, rendering it an *actor-critic off-policy* approach for *continuous state* and *action spaces*.

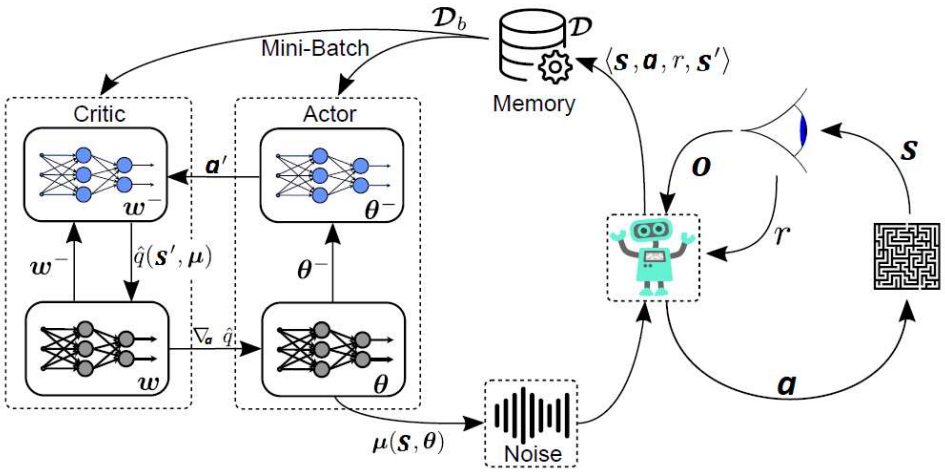
Tweaks

Similar to DQN, several tweaks are introduced to stabilize and improve the learning process.

- Experience Replay Buffer
- (Fixed) Target Networks; with soft updates:
$$\boldsymbol{w}^- = (1 - x)\boldsymbol{w}^- + x\boldsymbol{w}$$
- Minibatch Sampling:
$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma \hat{q}(\boldsymbol{s}', \mu(\boldsymbol{s}', \boldsymbol{\theta}^-), \boldsymbol{w}) \right) - \hat{q}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{w}) \right]^2_{\boldsymbol{D}_b}$$
- Batch Normalization
- Exploration by adding noise to actions (cf. *8.5.1*)

💡 The main idea behind batch normalization is to normalize the outputs of each layer before they go into the activation function. This is done by subtracting the batch mean and dividing by the batch standard deviation, effectively re-centering and re-scaling the outputs. This helps to control the distribution of the inputs to each layer and can prevent the network from experiencing problems like vanishing or exploding gradients during training.

Overview



8.7 Twin Delayed DDPG (TD3)

TD3 is a direct successor of DDPG. In order to reduce both the maximization bias and the learning variance, TD3 introduces mainly three measures on top of the DDPG algorithm.

8.7.1 Prior Problems

Maximization Bias and Double Learning

Recall the problem of *Maximization Bias* and Double Learning of value-based methods (cf. *5.5*):

Due to the inherent randomness in RL environments, the estimation of expected rewards can sometimes be more optimistic than the actual average outcome. This is particularly pronounced in algorithms like Q-learning, where we use a `max`-operation to select the action with the highest estimated Q-value. The high reward could indeed be due to a lucky sample, but because the "max" operation doesn't consider the underlying variability of these estimates, it tends to favor actions that have seen high rewards - even if those high rewards were due to chance and not a truly superior action. This can lead to a bias where the Q-values are consistently overestimated, causing the agent to choose actions based on an overly optimistic view of the expected reward.

Theres an additional problem when applying function approximation: the estimator itself introduces additional variance during the learning process which represents another source of the maximization bias problem. This problem was addressed in the value-based context by using double learning, but so far not for actor-critic-based methods.

Learning Variance

Using function approximation, the Bellman equation (*target for critic loss*) is never exactly satisfied, leaving room for some amount of *residual TD-error*. Although this error might be considered small per update step, it may accumulate over future steps if biased. It was observed that variance of $\hat{q}$ will be proportional to the variance of *future reward* and *residual TD-errors*.

If γ is large, the estimation variance might increase significantly. Mini-batch sampling will contribute to this variance issue.

8.7.2 Clipped Double Q-Learning for Actor-Critic

Following Maximization Bias and Double Learning, a second pair of critics $\{\hat{q}_{w1}, \hat{q}_{w2}\}$ is introduced. In contrast, the clipped target y (with target critic networks $\{\boldsymbol{w}_1^-, \boldsymbol{w}_2^-\}$) is:

$$y = r + \gamma \min_{i=1,2}\left[\hat{q}_{\boldsymbol{w}_i^-}(\boldsymbol{s}', \boldsymbol{a}')\right]$$

which provides an upper-bound on the estimated action value. May introduce some underestimation; but that is considered less critical than overestimation, since the value of underestimated actions will not be explicitly propagated through the policy update.

The min operator will also (indirectly) favor actions leading to values with estimation errors of lower variance.

8.7.3 Target Policy Smoothing Regularization

In DDPG, the actor network outputs a single action for a given state, and this action is used to calculate the Q-value from the critic network. In practice, this means the actor learns to find a very specific action that maximizes the critic's Q-value estimate. But what if the critic's Q-value estimate is a little off, maybe due to some noise or error? The actor could get really good at picking an action that doesn't work as well in reality as the critic's estimate suggests.

The idea is to add a bit of noise to the actions that the actor suggests, making the action space around the chosen action a bit „fuzzier“. This means the critic now learns to estimate Q-values over a range of actions close to the one that the actor suggests, not just the single action. The hope is that this will make the actor's policy more robust to small changes and errors in the Q-value estimates.

$$y = r + \gamma \hat{q}_{\boldsymbol{w}^-}(\boldsymbol{s}', \boldsymbol{\mu}_{\boldsymbol{\theta}^-}(\boldsymbol{s}') + \epsilon)$$

Here, the *clipped noise* $\epsilon \sim clip(\mathcal{N}(0, \Sigma), -\boldsymbol{c}, \boldsymbol{c})$ is a mean-free, Gaussian noise with covariance Σ, which is clipped at $\pm\boldsymbol{c}$ while $\boldsymbol{\theta}^-$ are the policy target network parameters.

However, this added noise could potentially cause the suggested actions to violate the constraints of the action space.

To ensure that the "noisy" actions remain within the valid action range, the noise needs to be clipped to a certain range, typically a small range around the action suggested by the actor. Then, after adding the clipped noise, the action needs to be clipped again to ensure it remains within the overall valid action space:

$$\boldsymbol{a}' = clip(\boldsymbol{\mu}_{\boldsymbol{\theta}^-}(\boldsymbol{s}') + \epsilon, \underline{\boldsymbol{a}}, \overline{\boldsymbol{a}})$$

This modified action is then used for the target calculation.

8.7.4 Delayed Policy Updates

Similar to DDPG, TD3 uses policy target networks $\boldsymbol{\theta}^-$ and (two) critic target networks $\{\boldsymbol{w}_1^-, \boldsymbol{w}_2^-\}$ in order to provide (rather) fixed Q-learning targets trying to stabilize the learning of $\hat{q}$.

The target networks are also continuously updated using:

$$\boldsymbol{w}_i^- = (1-\tau)\boldsymbol{w}_i^- + \tau \boldsymbol{w}_i$$
$$\boldsymbol{\theta}^- = (1-\tau)\boldsymbol{\theta}^- + \tau \boldsymbol{\theta}$$

However, each policy update will inherently change the (true) Q-learning target directly adding variance to the learning process. Therefore, it is argued that a policy update should not follow after each Q-learning update such that the critic can adapt properly to the previous policy update. We can consider this update rate a hyperparameter.

8.8 Trust Region Policy Optimization (TRPO)

Above methods have all been focusing on deterministic policies. This will change now as we look into stochastic policies $\pi(\boldsymbol{a}|\boldsymbol{s})$ in the following sections. First, we rewrite the performance metric for the episodic task $v_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{s}_0)$ (cf. *8.2 Policy Gradient Theorem*) to obtain:

$$J_\pi = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_k \right]$$

We can represent an updated policy $\pi \to \tilde{\pi}$ by its advantage function $a_\pi(\boldsymbol{s}, \boldsymbol{a}) = q_\pi(\boldsymbol{s}, \boldsymbol{a}) - v_\pi(\boldsymbol{s})$ (cf. *8.3.2*):

$$J_{\tilde{\pi}} = J_\pi + \int_{\mathcal{S}} p^{\tilde{\pi}}(\boldsymbol{s}) \int_{\mathcal{A}} \tilde{\pi}(\boldsymbol{a}|\boldsymbol{s}) * a_\pi(\boldsymbol{s}, \boldsymbol{a})$$

While for finite MDPs, the policy improvement theorem guaranteed $J_{\tilde{\pi}} \geq J_\pi$ for each policy update, there might be *some* state-action combinations where $\int_{\mathcal{A}} \tilde{\pi}(\boldsymbol{a}|\boldsymbol{s}) * a_\pi(\boldsymbol{s}, \boldsymbol{a}) < 0$ (the update is worse) for continuous MDPs using function approximation.

📝 For easier calculation, we introduce a local approximation:

$$\mathcal{L}(\tilde{\pi}) = J_\pi + \int_{\mathcal{S}} p^{\pi}(\boldsymbol{s}) \int_{\mathcal{A}} \tilde{\pi}(\boldsymbol{a}|\boldsymbol{s}) * a_\pi(\boldsymbol{s}, \boldsymbol{a})$$

where $p^\pi(\boldsymbol{s})$ is used instead of $p^{\tilde{\pi}}(s)$, i.e., neglecting the state distribution change due to a policy update. We assume therefore that the policy update does not significantly change the state distribution.

- $\mathcal{L}(\tilde{\pi})$ is the objective function of the new policy $\tilde{\pi}$, where $\tilde{\pi}$ represents an updated or adjusted policy. The goal in policy optimization is to find the policy that maximizes this function.
- J_π is the value function for policy π, computed as the expectation of the sum of future rewards when following policy π from each state, weighted by the *stationary state distribution* under policy π.
- $p^\pi(\boldsymbol{s})$ is the *stationary state distribution* following policy π. It gives the likelihood of being in each state in the long run when following policy π

💡 For any parameterizable and differentiable policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}, \boldsymbol{\theta})$ it can be shown that, *given a small enough policy update step*, the local approximation is the same as the true value $\mathcal{L}(\tilde{\pi}) = J_{\tilde{\pi}}$. Also, their gradients are the same. Hence, for a small enough step size, improving $\mathcal{L}(\tilde{\pi})$ will also improve $J_{\tilde{\pi}}$.

❗ Therefore, we need a metric to describe how much a policy has changed in the action space when updating the policy in the parameter space.

8.8.1 Kullback-Leibler (KL) Divergence

Against this background, we make use of the KL divergence:

$$D_{KL}(P||Q) = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

The KL divergence of two probability distributions P and Q provides a measure of the difference between these two distributions. It quantifies the "amount of information lost" when Q is used to approximate P.

Formally, it's defined as the sum over all events in the probability space, of the probability of the event according to P, times the log of the ratio of the probability of the event according to P and the probability of the event according to Q.

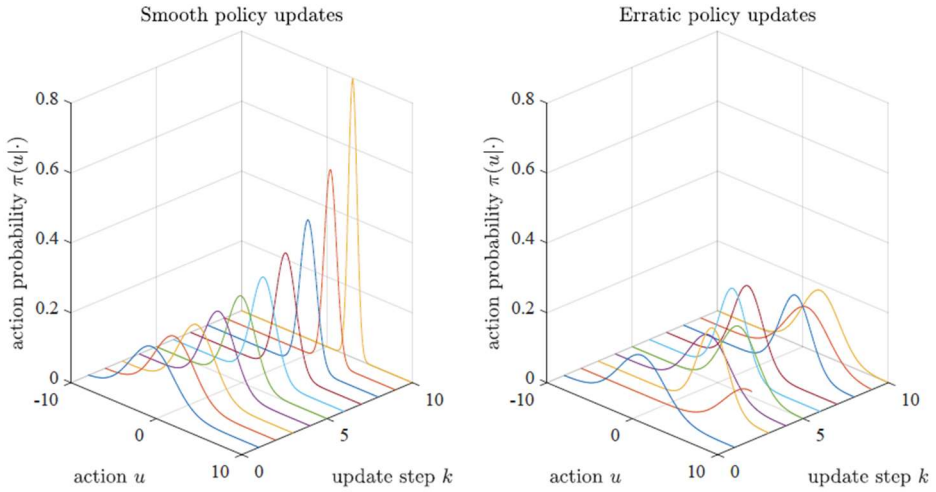
It's important to note that KL divergence is not symmetric; that is, $D_{KL}(P||Q)$ is not generally equal to $D_{KL}(Q||P)$.

The TRPO updates the policy parameters while constraining the KL divergence between the new and the old policy distribution:

$$\max_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}),$$

$$s.t. \bar{D}_{KL}(\boldsymbol{\theta}, \boldsymbol{\theta}_k) \leq \kappa$$

Hence, we want to limit the average KL divergence w.r.t. the states visited by the old policy. The constraint κ (kappa) is a TRPO hyperparameter (typically $\kappa << 1$). We can use this as a tool to prevent erratic policy updates.



Simplified representation of the policy evolution for a scalar action given some fixed state.

Left: TRPO-style updates finding the optimal action with increasing probability.

Right: Unmonitored policy distributions not converging towards an optimal policy ('policy chattering').

8.8.2 Sample-Based Objective

To solve the maximization $\max_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}_k}(\boldsymbol{\theta})$ we will make use of samplings from MC rollouts. Expanding the objective yields:

$$\max_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}) = \max_{\boldsymbol{\theta}} J_{\pi_k} + \int_{\mathcal{S}} p^{\pi_k}(\boldsymbol{s}) \int_{\mathcal{A}} \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta}) * a_{\pi_k}(\boldsymbol{s},\boldsymbol{a})$$

- J_{π_k} can be dropped since it's a constant and therefore not relevant to the maximization
- Since we cannot sample the entire state space $\int_{\mathcal{S}}$ we need to approximate it by:

  $$\int_{\mathcal{S}} p^{\pi_k}(\boldsymbol{s}) \approx \frac{1}{1-\gamma} \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}}[\boldsymbol{S}]$$

  This represents sampling states according to the policy $\pi_{\boldsymbol{\theta}_k}$. We are interested in states that are likely under the current policy because those are the states where the policy will be used. So, Monte Carlo rollouts under the current policy are used to generate samples. The $\frac{1}{1-\gamma}$ factor comes from summing up an infinite geometric series in the discounted case, where γ is the discount factor.

- Moreover, applying importance sampling based on data from the old policy we can approximate:

  $$\int_{\mathcal{A}} \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta}) * a_{\pi_k}(\boldsymbol{s},\boldsymbol{a}) \approx \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{A}|\boldsymbol{S})} a_{\pi_k}(\boldsymbol{S},\boldsymbol{A})$$

  This is an application of the importance sampling method, which is commonly used in policy gradient methods to reduce the variance of the gradient estimates. Here, we are estimating the expectation of the advantage function $a_{\pi_k}(\boldsymbol{S},\boldsymbol{A})$ under the new policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})$ based on samples from the old policy $\pi_{\boldsymbol{\theta}_k}(\boldsymbol{A}|\boldsymbol{S})$. The ratio $\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{A}|\boldsymbol{S})}$ is known as the importance sampling ratio, which adjusts for the difference in probability of the actions between the new and old policies.

8.8.3 Main Optimization Problem

Applying the previous sample-based estimation we obtain:

$$\boldsymbol{\theta}_{k+1} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{A}|\boldsymbol{S})} a_{\pi_k}(\boldsymbol{S},\boldsymbol{A}) \right],$$
$$s.t. \quad \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[D_{KL}\left(\pi_{\boldsymbol{\theta}_k}(\cdot\,|\boldsymbol{S}) \,||\, \pi_{\theta}(\cdot\,|\boldsymbol{S}) \right) \leq \kappa \right]$$

With $\mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{A}|\boldsymbol{S})} a_{\pi_k}(\boldsymbol{S},\boldsymbol{A}) \right]$ being the surrogate objective function that TRPO maximizes. It's an estimate of the expected return under the new policy parameters $\boldsymbol{\theta}$, based on samples generated from the old policy $\pi_{\boldsymbol{\theta}_k}$. It uses the importance sampling ratio $\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{A}|\boldsymbol{S})}$ to adjust for the difference between the new and old policies, and it multiplies this ratio by the advantage function $a_{\pi_k}(\boldsymbol{S},\boldsymbol{A})$, which estimates how much better it is to take action $\boldsymbol{A}$ in state $\boldsymbol{S}$ under the new policy compared to the average action under the old policy.

This is the main optimization problem that TRPO solves at each iteration. It maximizes a surrogate objective function that provides an estimate of the expected improvement of policy under the new parameters θ over the old parameters $\boldsymbol{\theta}_k$. At the same time, it ensures that the new policy does not deviate too far from the old policy by the KL-divergence constraint.

Hence, we have a three-step procedure for each TRPO update:

1. Use Monte-Carlo simulations based on the old policy to obtain data.
2. Use the data to construct ***Error! Reference source not found.***.
3. Solve the constrained optimization problem to update the policy parameter vector.

8.8.4 Generalized Advantage Estimation (GAE)

💡 Accurately calculating the advantage function often requires knowledge of the true state-value and action-value functions, which are typically not known in practice. Therefore, we need methods to estimate the advantage function from data.

Having data $\langle \boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}' \rangle$ in $\boldsymbol{\mathcal{D}}$ from a Monte Carlo rollout available, an important problem is to estimate $a_{\pi_k}(\boldsymbol{S}, \boldsymbol{A})$ in ***Error! Reference source not found.***. A particular suggestion in the TRPO context is to use a GAE:

$$a_k^{(\gamma,\lambda)} = \sum_{i=0}^{\infty} (\gamma\lambda)^i \delta_{k+i}$$

Here, $\delta_k = r_k + \gamma v(s_{k+1}) - v(s_k)$ is a single advantage sample. Hence, the GAE is the exponentially weighted average of the discounted advantage samples with an additional weighting λ.

- Similar formulation compared to TD(λ) but instead of the state value the estimator's target is the advantage.
- The choice of $(\gamma\lambda)$ trade-offs the bias and variance of the estimator.

In practice, the infinite sum over all future time steps is typically truncated to a finite horizon for computational feasibility, which corresponds to the number of steps in the Monte Carlo rollout from which the data $\langle \boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}' \rangle$ in $\boldsymbol{\mathcal{D}}$ is collected.

8.8.5 Key Facts

- The TRPO constrains policy distribution changes when updating the policy parameters (for stochastic policies and on-policy learning)
- The objective/motivation is to enable a monotonically improving learning process
- Using trust regions, erratic policy updates should be prevented

8.8.6 Main Hurdles

- Constructing the objective function and constraint requires Monte Carlo rollouts (time consuming, data inefficient); rollout buffer is different, only valid for one update step
- When the sampled optimization problem is set up, a nonlinear and constrained optimization step is required (no simple policy gradient)
- For speedy implementations, only approximate solutions of TRPO are possible

8.9 Proximal Policy Optimization (PPO)

PPO aims to retain the benefits of TRPO's approach while simplifying the computational requirements. It does this by converting the original constrained problem into an equivalent unconstrained problem. The objective will be reformulated to incorporate mechanisms preventing excessively large variations of the policy distribution during a parameter update (leading to an updated policy with sufficient proximity to the old one). This is achieved by modifying the objective function to include a penalty term, which encourages updates to stay close to the original policy.

Therefore, PPO incorporates two variants which we will discuss:

8.9.1 Clipped Surrogate Objective

Recall:

- $\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})$ represents the probability of taking an action under the policy we are currently optimizing and want to update .
- $\pi_{\boldsymbol{\theta}_k}(\boldsymbol{A}|\boldsymbol{S})$ represents the probability of taking an action under the "old" policy, the policy that we followed to collect the experiences we are currently learning from.

The first approach is based on the following objective:

$$E_{\pi_{\boldsymbol{\theta}_k}}\left[\min\left\{\begin{array}{c}\dfrac{\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{A}|\boldsymbol{S})}a_{\pi_k}(\boldsymbol{S},\boldsymbol{A}) \\ clip\left(\dfrac{\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{A}|\boldsymbol{S})}, 1-\epsilon, 1+\epsilon\right)a_{\pi_k}(\boldsymbol{S},\boldsymbol{A})\end{array}\right\}\right]$$

- The clipped version of the above probability ratio, which is constrained to lie within $[1-\epsilon, 1+\epsilon]$. If the ratio is below this range, it's *clipped* to 1 - ϵ, and if it's above, it's clipped to 1 + ϵ. This *clip operation* prevents the probability ratio from becoming too large or too small, thus ensuring the updated policy $\pi_{\boldsymbol{\theta}}$ does not deviate too far from the old policy $\pi_{\boldsymbol{\theta}_k}$. This term is also multiplied by the advantage function $a_{\pi_k}(S, A)$.
- The minimum of these two quantities is taken to form the final objective. The expectation of this minimum value is what PPO seeks to maximize. By taking the minimum, PPO ensures that the updated policy does not get too much credit for actions that are much more likely under the new policy compared to the old policy.

Consider a single sample $(\boldsymbol{s}, \boldsymbol{a})$ with a po*sitive advantage* $a_{\pi_k}(\boldsymbol{s}, \boldsymbol{a})$. Because the advantage is positive, after performing a policy gradient step, the likelihood of $\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})$ increases.

The entire objective is therefore limited to $(1+\epsilon)\,\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})$. Interpretation: the new policy does not benefit from going further away from the old policy.

Consider a single sample $(\boldsymbol{s}, \boldsymbol{a})$ with a negative *advantage* $a_{\pi_k}(\boldsymbol{s}, \boldsymbol{a})$. Because the advantage is negative, after performing a policy gradient step, the likelihood of $\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})$ decreases.

Similarly, the entire objective is therefore limited to $(1-\epsilon)\,\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})$. Interpretation: the new policy does not benefit from going further away from the old policy.

8.9.2 Adaptive KL Penalty

The second PPO variant makes use of the following KL-penalized objective:

$$E_{\pi_{\boldsymbol{\theta}_k}}\left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{A}|\boldsymbol{S})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{A}|\boldsymbol{S})} a_{\pi_k}(\boldsymbol{S},\boldsymbol{A}) - \beta D_{KL}\left(\pi_{\boldsymbol{\theta}_k}(\cdot\,|\boldsymbol{S}) \,||\, \pi_{\boldsymbol{\theta}}(\cdot\,|\boldsymbol{S})\right)\right]$$

The second term $\beta D_{KL}(\cdot)$ measures the divergence between the old policy $\pi_{\boldsymbol{\theta}_k}$ and the new policy $\pi_{\boldsymbol{\theta}}$ for state $\boldsymbol{S}$. This term is subtracted from the objective to penalize changes in the policy that lead to a high KL divergence.

The β parameter is a coefficient that determines the strength of the KL penalty. Higher β values make the penalty stronger, thus more strongly discouraging large policy changes.

This transfers the KL-based constraint into a penalty for large policy distribution changes. The parameter β weights the penalty against the policy improvement. The original PPO implementation suggests an adaptive tuning of β w.r.t. the sampled average KL divergence $D_{KL}(\boldsymbol{\theta}_k, \boldsymbol{\theta})$ estimated from previous experience.

This formulation seeks to strike a balance between policy improvement (as guided by the advantage function) and policy stability (as enforced by the KL penalty). It allows the policy to change and improve, but not so much that it becomes significantly different from the old policy, which could destabilize the learning process. The balance between improvement and stability is controlled by the β parameter.

8.9.3 Algorithmic Implementation

input: diff. stochastic policy fct. $\pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta})$ and value fct. $\hat{v}(\boldsymbol{s},\boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha\}$
init: weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily, memory $\mathcal{D}$
for $j = 1, 2, \ldots,$ *iterations* **do**
  initialize $\boldsymbol{s}_0$ (if new episode);
  collect a set of tuples $\langle \boldsymbol{s}_k, \boldsymbol{a}_k, r_{k+1}, \boldsymbol{s}_{k+1}\rangle$ by running $\pi(\boldsymbol{a}|\boldsymbol{s},\boldsymbol{\theta}_j)$;
  store them in $\mathcal{D}$;
  estimate the advantage $\hat{a}_{\pi_j}(\boldsymbol{s},\boldsymbol{a})$ based on $\hat{v}(\boldsymbol{s},\boldsymbol{w}_j)$ and $\mathcal{D}$ (e.g., GAE);
  $\boldsymbol{\theta}_{j+1} \leftarrow$ policy gradient update on *8.9.1* or *8.9.2;*
  $\boldsymbol{w}_{j+1} \leftarrow$ minimizing the mean-squared TD errors using $\mathcal{D}$;
  delete entries in $\mathcal{D}$;

8.10 Soft Actor Critic (SAC)

Soft Actor-Critic (SAC) is a reinforcement learning algorithm that aims to maximize expected return while also maintaining a balance of exploration and exploitation. It achieves this through a form of entropy regularization, where the policy is encouraged to be more stochastic, thus improving exploration.

The *maximum entropy principle* refers to a strategy that encourages the agent to explore a wide range of actions, as opposed to solely the most promising ones. This principle is embodied by maximizing an entropy-augmented reward function, which encourages not just maximizing the expected cumulative reward, but also ensuring the agent's policy remains as random as possible. This added randomness fosters more robust exploration.

9 Other Stuff

9.1 Experience Replay Buffer vs. Rollout Buffer

Experience Replay Buffer and Rollout Buffer are both forms of memory storage, but they have different uses and characteristics:

- Experience Replay Buffer (cf. *7.4.1*): This is mainly used in off-policy algorithms like DQN or DDQN. In these methods, the agent learns from past experiences which are stored in the Experience Replay Buffer. This buffer stores a number of the most recent experiences $\langle \boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}', done \rangle$. The agent samples a batch of experiences from this buffer to learn, allowing it to reuse past experiences and learn more efficiently. This breaks the correlation between consecutive experiences, which stabilizes the learning process.

- Rollout Buffer: This is used in on-policy algorithms like PPO. On-policy methods update the policy based on the experiences $\langle \boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}', done \rangle$ of the current policy only. The Rollout Buffer stores the experiences generated by the policy during each rollout (or episode). Once the rollout is complete, the experiences are used to update the policy. After the policy is updated, the Rollout Buffer is typically discarded or reset, because these experiences do not apply to the updated policy anymore.

So the key difference is that an Experience Replay Buffer is used in off-policy learning for storing and reusing experiences over multiple updates, while a Rollout Buffer is used in on-policy learning and only stores the experiences of the current policy for one update.

	Experience Replay Buffer	**Rollout Buffer**
Value-Based	DQN, Rainbow DQN	
Policy-Based		MC Policy Gradient (REINFORCE)
Actor-Critic	DDPG, TD3, SAC	PPO

9.2 Bias

In RL the term "bias" usually refers to the systematic error that an estimator introduces, that is, the difference between the expected value of the estimator and the true value it's trying to estimate.

For instance, if the learning algorithm tends to consistently underestimate or overestimate the expected rewards from actions, we would say it's a "biased" estimator. Bootstrapping methods like TD introduce bias through their estimates (since they base the estimate of a state's value on the estimated value of future states), but they have lower variance than Monte Carlo methods, which don't bootstrap but have higher variance because they rely on the actual, potentially noisy, rewards from full trajectories.

In function approximation methods, using a simpler function approximator (like a linear function) might introduce more bias (because it can't perfectly fit the true value or policy function), but it also reduces variance and overfitting compared to a complex function approximator (like a deep neural network).