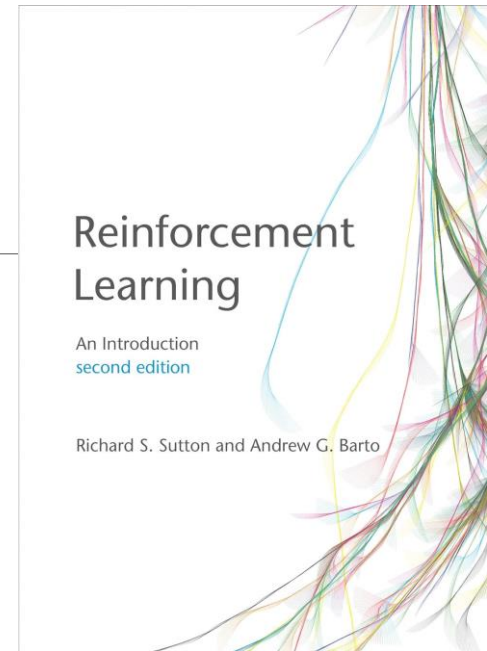


Reinforcement Learning

MIGUEL S. E. MARTINS

11 OCTOBER 2023

Reinforcement learning (RL)

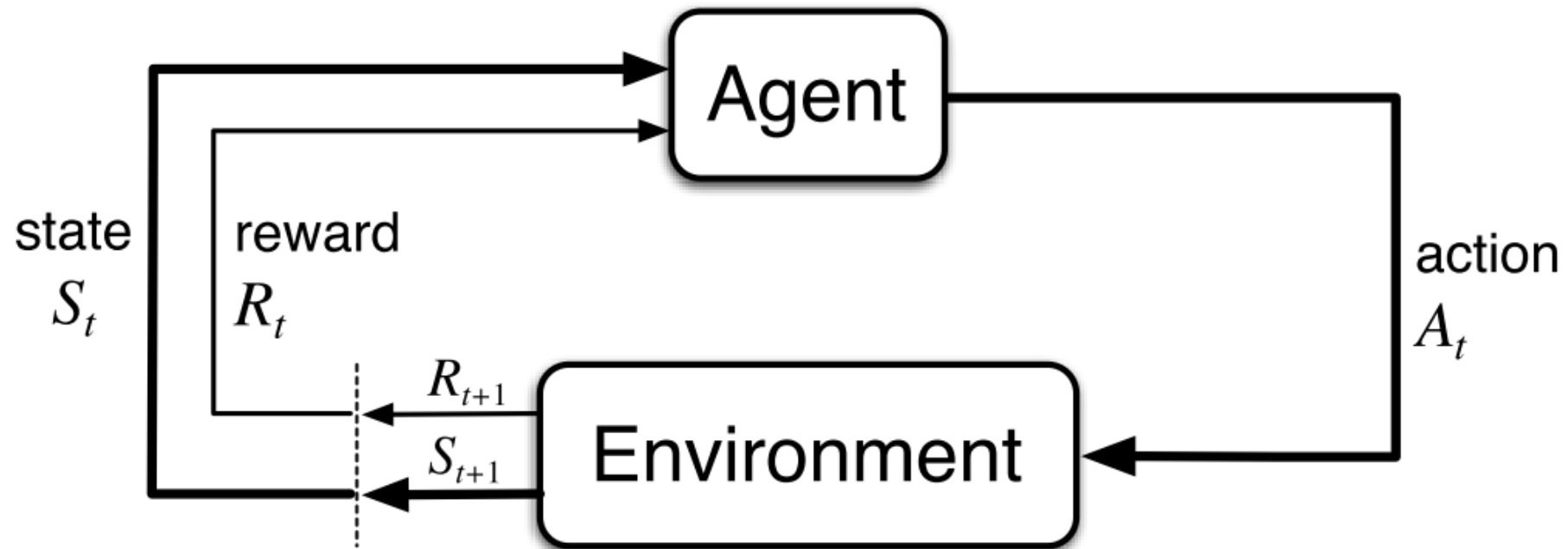


[http://incompleteideas.net/
book/the-book-2nd.html](http://incompleteideas.net/book/the-book-2nd.html)

1.1 Reinforcement Learning

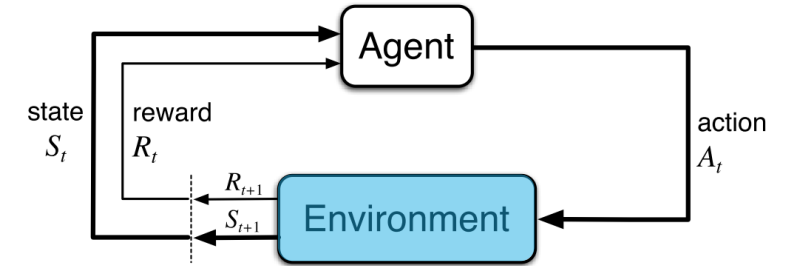
Reinforcement learning is **learning what to do**—how to map situations to actions—so **as to maximize a numerical reward signal**. The learner is not told which actions to take, but instead must discover which actions yield the most reward by **trying them**.


RL loop

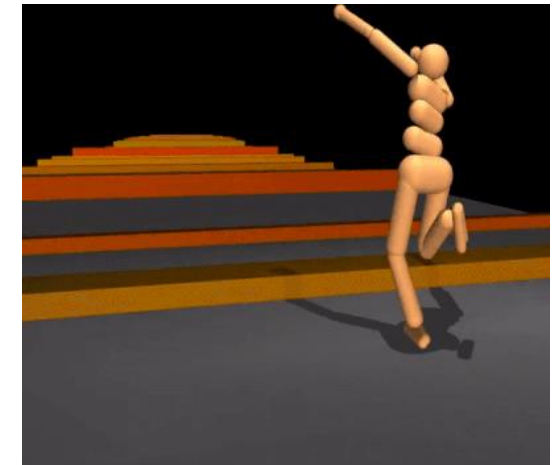
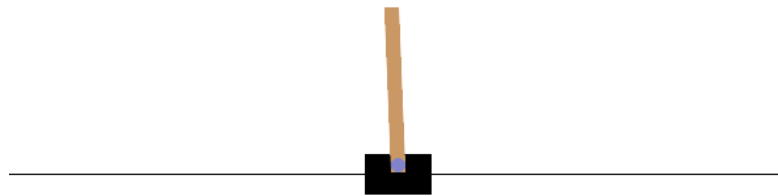
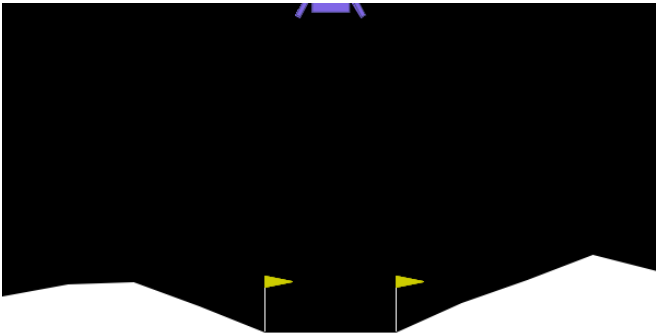


Agent-environment interaction in an MDP system (Sutton, 2018)

Environment



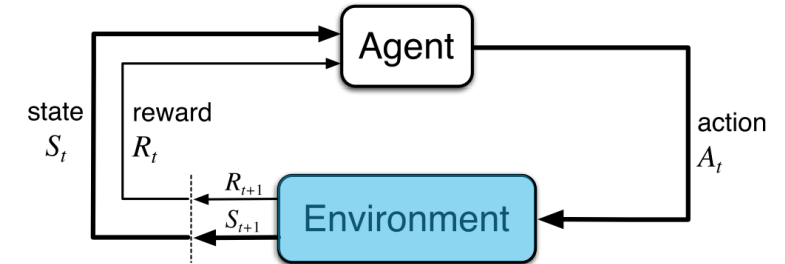
- Markov Decision Process (MDP): mathematical framework to model **sequential** decision-making problems.
- Popular Python library for RL environments:  Gymnasium



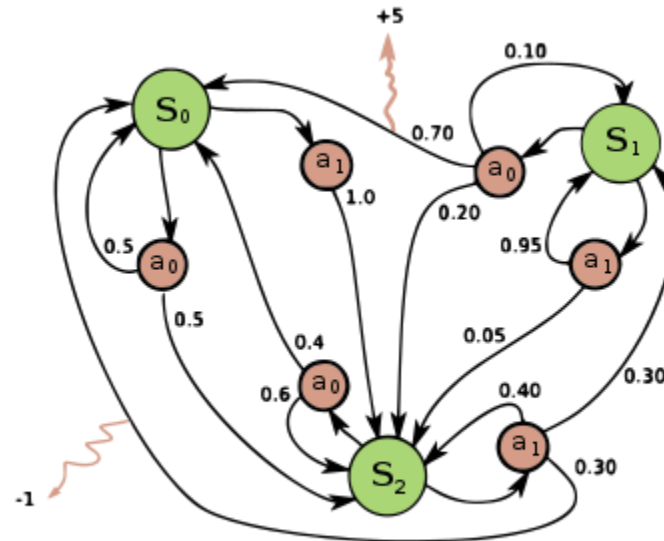
Producing flexible behaviours in simulated environments
(DeepMind, 2017)

Environment

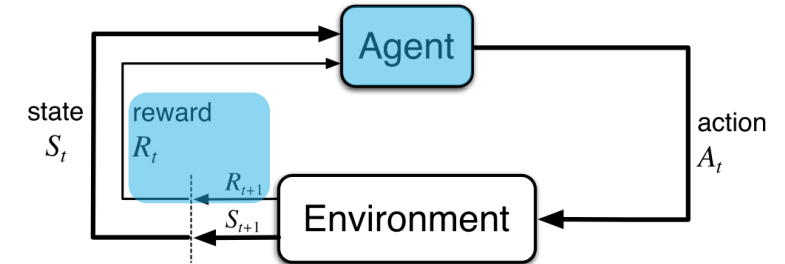
Finite Markov Decision Process



- MDP requires **well-defined transition probabilities**.
 - For each state-action pair, these probabilities define the **likelihood** of reaching a new state s' when action a is taken in state s .

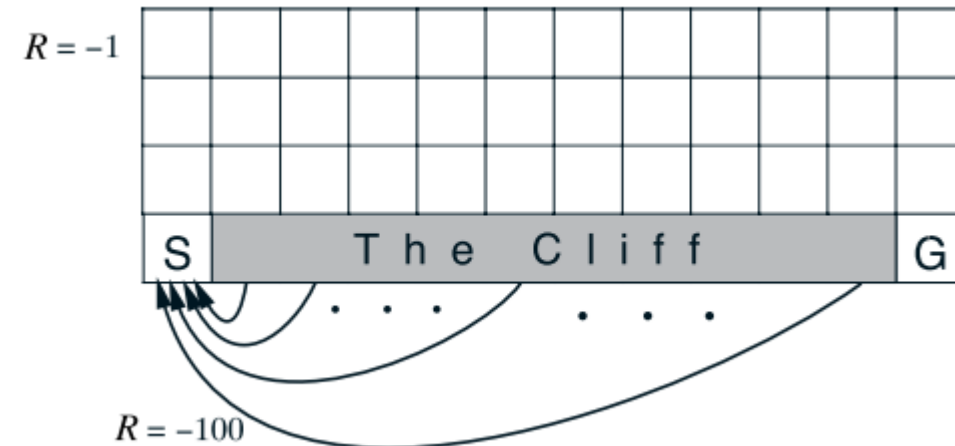
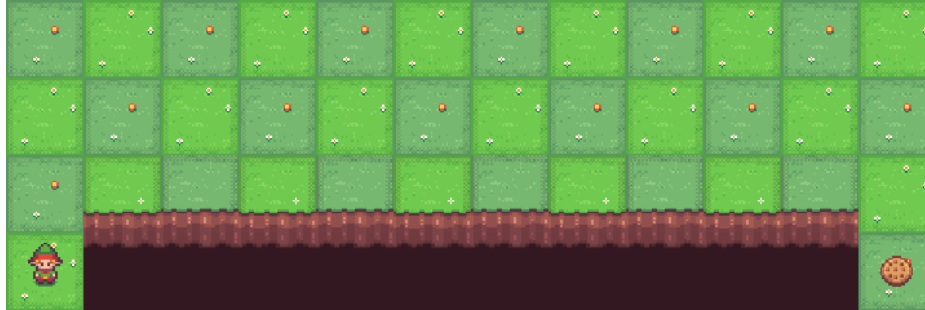


Agent

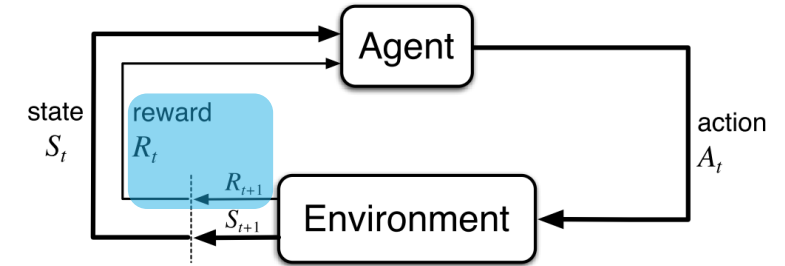


The agent's **objective** is to **maximize the amount of reward it receives over time**.

- Actions influence not just immediate rewards, but future states, and thus future rewards.
- MDPs involve delayed reward and the need to tradeoff immediate and delayed reward



Goals and rewards



In RL, the purpose or goal of the agent is formalized in terms of a special signal, the **reward**.

- At each time step, the reward is a simple number, $R_t \in R$

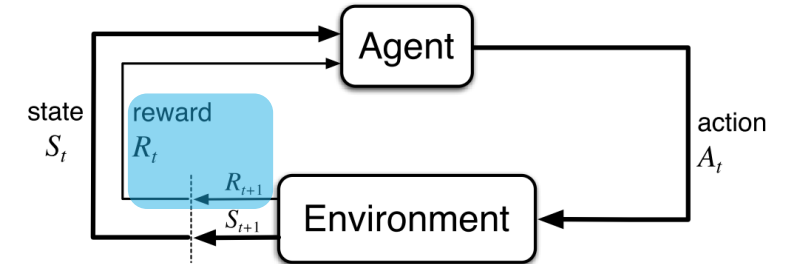
The agent's goal is to maximize the total amount of reward it receives.

- This means maximizing not immediate reward, but cumulative reward in the long run.

Expected return: specific function of the reward sequence, i.e. its sum. T is final time step.

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

Goals and rewards



When agent-environment interaction goes without limits, **discounting** is needed:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*.

Returns at successive time steps are related to each other in a way that is important for the theory and algorithms of reinforcement learning

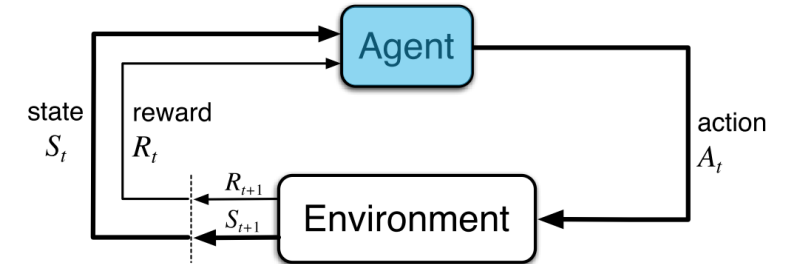
$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

Reward design



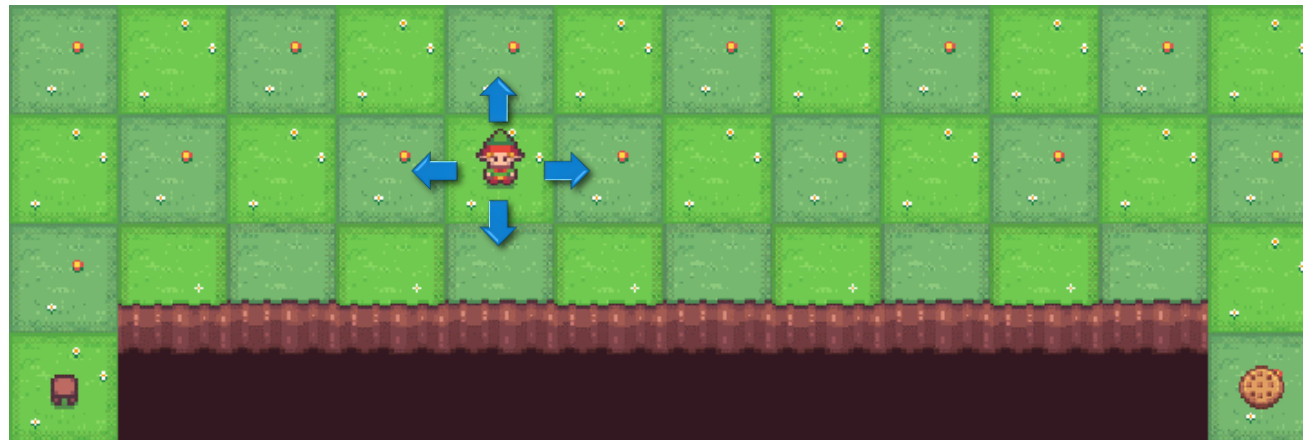
Faulty reward functions in the wild (OpenAI, 2016)

Agent



Policy (π): mapping from **states** to **actions** that guides the agent's decisions.

- Can be deterministic or stochastic.
- An optimal policy (π_*) is a policy that is better than or equal to all alternatives.
- There may be more than one optimal policy.



$\mathbb{E}_\pi[\cdot]$ is the expected value of a random variable given that the agent follows policy π , t is any time step

Policies and value functions

RL algorithms often estimate **value functions**, which are:

- Functions of **states**, $v_\pi(s)$, that estimate **how good a state is**

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \text{ for all } s \in \mathcal{S},$$

- Functions of **state-action pairs**, $q_\pi(s, a)$, that estimate **how good to perform an action on a state**

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right].$$

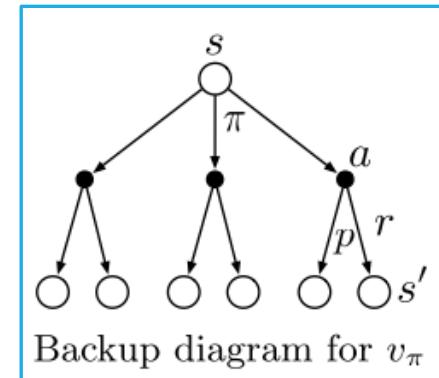
- The optimal versions are, respectively, $v_*(s) \doteq \max_{\pi} v_\pi(s)$, and $q_*(s, a) \doteq \max_{\pi} q_\pi(s, a)$,

Policies and value functions

$E_{\pi}[\cdot]$ is the expected value of a random variable given that the agent follows policy π , t is any time step

A fundamental property of value functions used throughout RL and dynamic programming is that they satisfy recursive relationships, as showed by the **Bellman equation** for v_{π}

$$\begin{aligned} v_{\pi}(s) &\doteq \mathbb{E}_{\pi}[G_t \mid S_t = s] \\ &= \mathbb{E}_{\pi}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) \left[r + \gamma \mathbb{E}_{\pi}[G_{t+1} | S_{t+1} = s'] \right] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) \left[r + \gamma v_{\pi}(s') \right], \quad \text{for all } s \in \mathcal{S}, \end{aligned}$$



Dynamic Programming

Policy Evaluation (Prediction)

Iterative policy evaluation applies the same operation to each state s :

- Replace the old value of s with a new value obtained from the old values of the successor states of s , and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π , the policy to be evaluated

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop:

$\Delta \leftarrow 0$

Loop for each $s \in \mathcal{S}$:

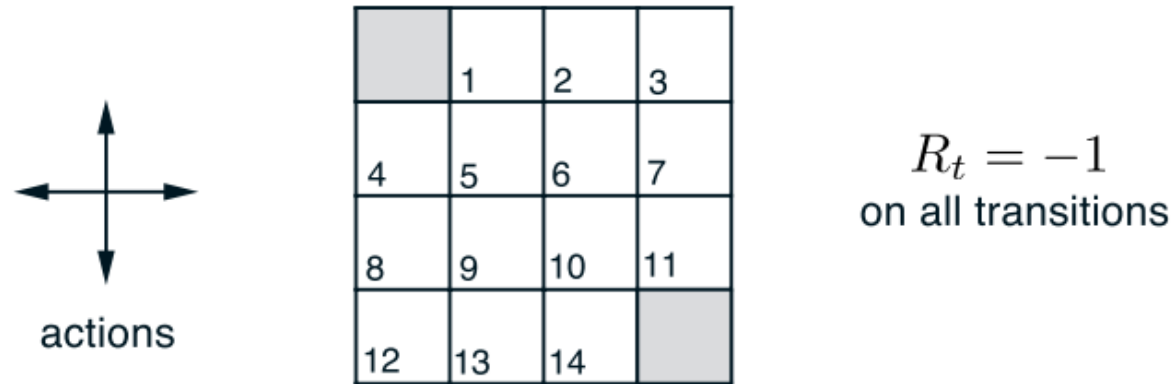
$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$

Example 4.1 Consider the 4×4 gridworld shown below.



The nonterminal states are $\mathcal{S} = \{1, 2, \dots, 14\}$. There are four actions possible in each state, $\mathcal{A} = \{\text{up}, \text{down}, \text{right}, \text{left}\}$, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. Thus, for instance, $p(6, -1 | 5, \text{right}) = 1$, $p(7, -1 | 7, \text{right}) = 1$, and $p(10, r | 5, \text{right}) = 0$ for all $r \in \mathcal{R}$. This is an undiscounted, episodic task. The reward is -1 on all transitions until the terminal state is reached. The terminal state is shaded in the figure (although it is shown in two places, it is formally one state). The expected reward function is thus $r(s, a, s') = -1$ for all states s, s' and actions a . Suppose the agent follows the equiprobable random policy (all actions equally likely). The left side of Figure 4.1 shows the sequence of value functions $\{v_k\}$ computed by iterative policy evaluation. The final estimate is in fact v_π , which in this case gives for each state the negation of the expected number of steps from that state until termination. ■

V_k for the
Random Policy

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

Dynamic Programming

Value Iteration

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop:

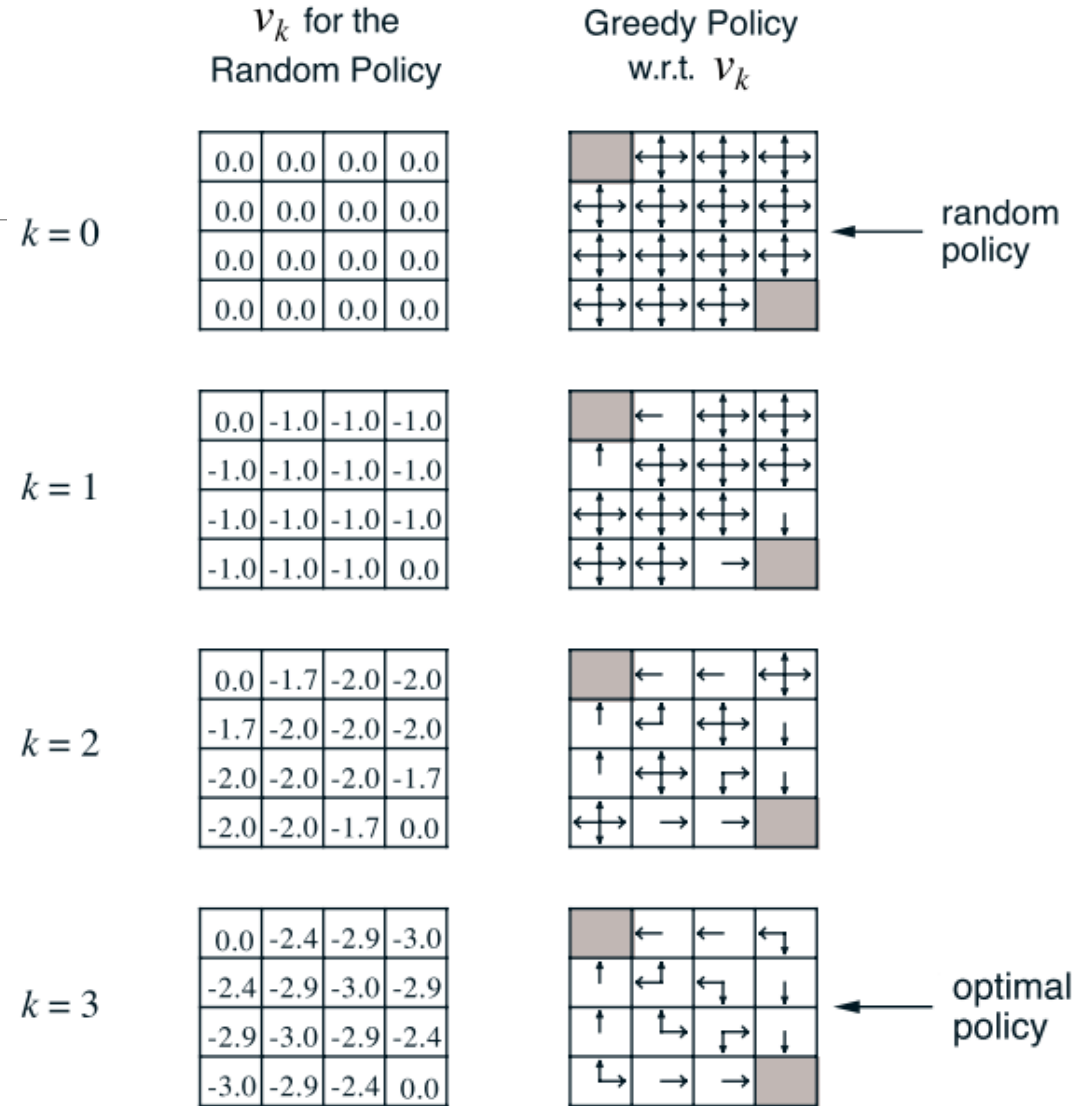
```
|  $\Delta \leftarrow 0$   
| Loop for each  $s \in \mathcal{S}$ :  
|    $v \leftarrow V(s)$   
|    $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$   
|    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ 
```

until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that

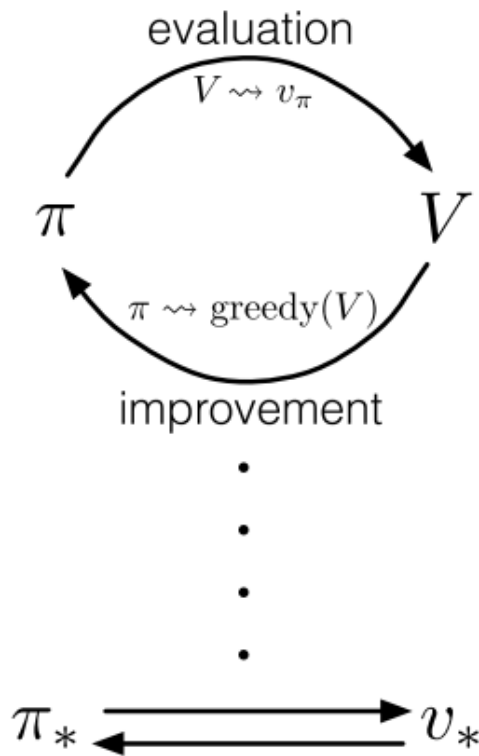
$$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

S



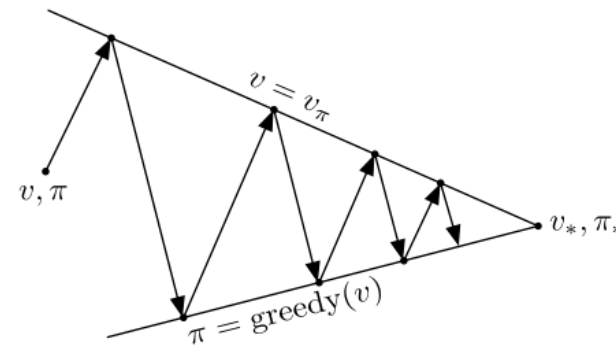
Dynamic Programming

Generalized Policy Iteration



Generalized policy iteration (GPI) to refer to the general idea of letting policy evaluation and policy improvement processes interact.

The evaluation and improvement processes in GPI can be viewed as both competing and cooperating.



Ultimately, DP may not be practical for large problems, yet it can be quite efficient compared with other MDP methods.

Monte Carlo Methods

Learning methods for estimating value functions and discovering optimal policies.

- Based on **averaging sample returns**.

Difference to DP: no knowledge of the environment

- However, adapt the idea of general policy iteration (GPI).
- Whereas there we **computed** value functions from **knowledge** of the MDP, here we **learn** value functions from **sample returns** with the MDP.

Offline method: agent learns from a fixed dataset of previously collected experiences.



Monte Carlo Methods

Monte Carlo Prediction

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow G + R_{t+1}$

Unless S_t appears in S_0, S_1, \dots, S_{t-1} :

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Monte Carlo Methods

Example: Blackjack

Example 5.1: Blackjack The object of the popular casino card game of *blackjack* is to obtain cards the sum of whose numerical values is as great as possible without exceeding 21. All face cards count as 10, and an ace can count either as 1 or as 11. We consider the version in which each player competes independently against the dealer. The game begins with two cards dealt to both dealer and player. One of the dealer's cards is face up and the other is face down. If the player has 21 immediately (an ace and a 10-card), it is called a *natural*. He then wins unless the dealer also has a natural, in which case the game is a draw. If the player does not have a natural, then he can request additional cards, one by one (*hits*), until he either stops (*sticks*) or exceeds 21 (*goes bust*). If he goes bust, he loses; if he sticks, then it becomes the dealer's turn. The dealer hits or sticks according to a fixed strategy without choice: he sticks on any sum of 17 or greater, and hits otherwise. If the dealer goes bust, then the player wins; otherwise, the outcome—win, lose, or draw—is determined by whose final sum is closer to 21.

Monte Carlo Methods

Example: Blackjack

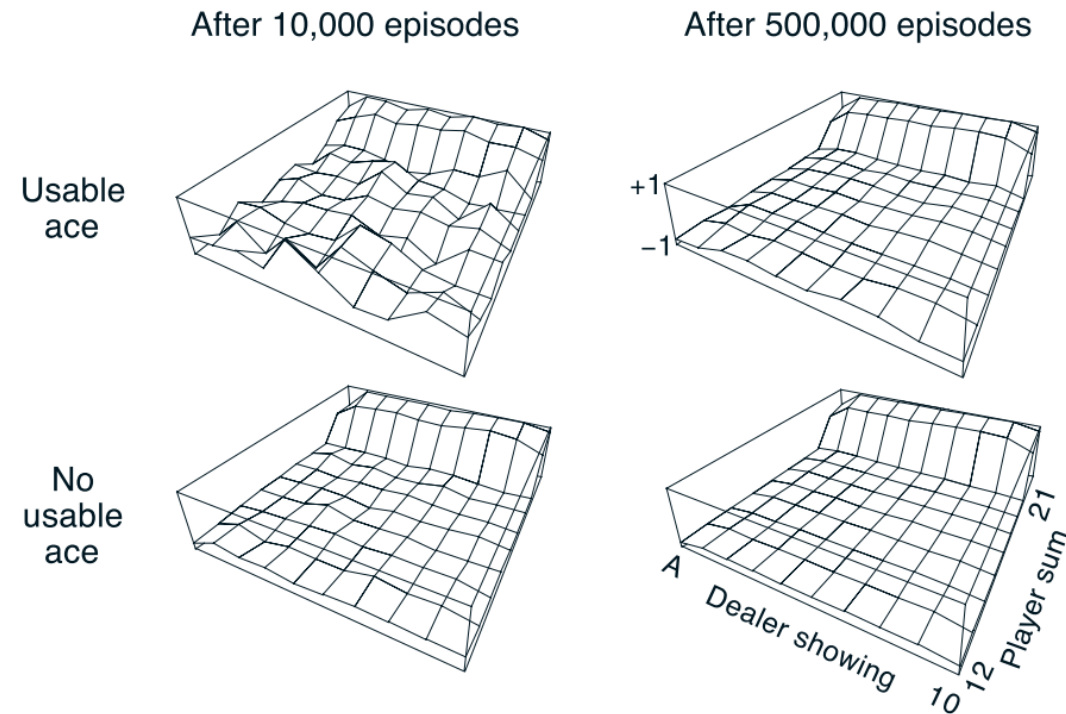


Figure 5.1: Approximate state-value functions for the blackjack policy that sticks only on 20 or 21, computed by Monte Carlo policy evaluation. ■

Monte Carlo with Exploring Starts

If π is a deterministic policy, following π will observe returns only for one of the actions from each state.

- To compare alternatives, we need to estimate the value of all the actions from each state.

Exploring starts assumption:

- Every pair has a nonzero probability of being selected as the start.
- This guarantees that all state–action pairs will be visited an infinite number of times in the limit of an infinite number of episodes.

Possible solution: **ϵ -greedy** strategy

- Using ϵ as a very small number
- With random chance ϵ , take a **random action**
- Else, select the action that will lead to **highest expected value**

Monte Carlo with Exploring Starts

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$

$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ such that all pairs have probability > 0 (exploring starts)

Generate an episode starting from S_0, A_0 , following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$

Monte Carlo Methods

Example: Blackjack

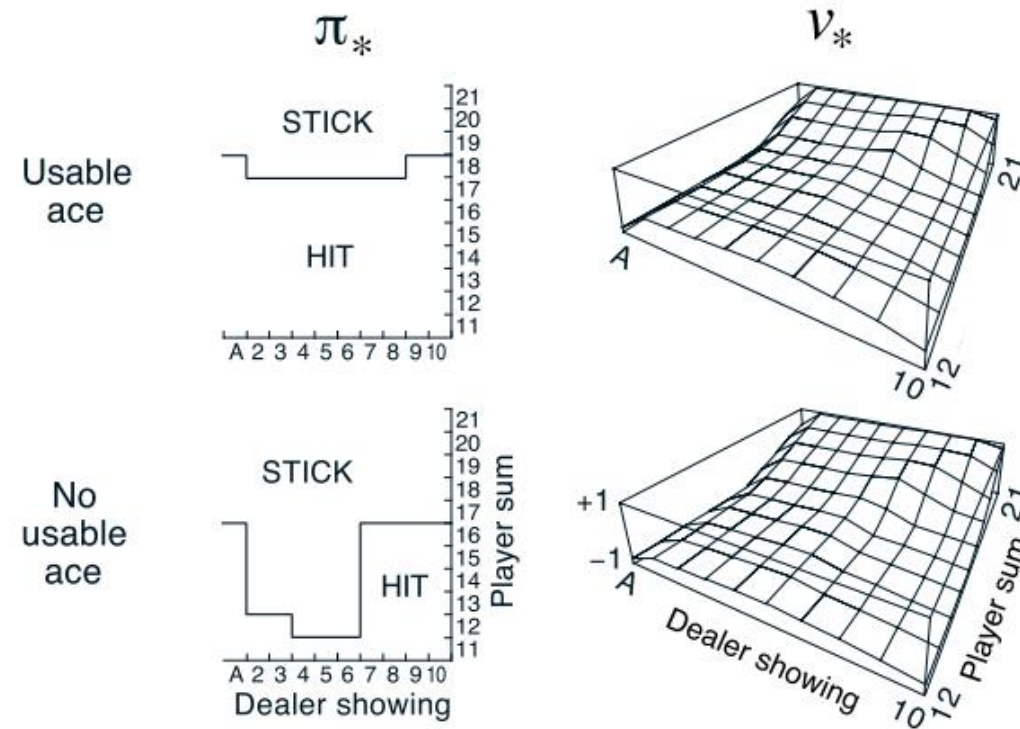


Figure 5.2: The optimal policy and state-value function for blackjack, found by Monte Carlo ES (Figure 5.4). The state-value function shown was computed from the action-value function found by Monte Carlo ES.

Monte Carlo Methods

On-policy vs. off-policy

How to learn action values based on optimal behaviour, but non-optimal behaviour is needed to explore all actions (and find the optimal actions)?

On-policy methods evaluate or improve the policy that is **used to make decisions**

- On-policy methods allow all actions to be selected at the start, then gradually become deterministic
- Example: epsilon-greedy

Off-policy methods evaluate or improve a policy **different** from that used to generate the data.

- Two policies, a learned one that becomes the optimal policy (**target policy**), and one that is more exploratory and is used to generate behaviour (**behaviour policy**).

Temporal-Difference learning

Like Monte Carlo methods, TD methods learn directly from raw experience without a model of the environment's dynamics

- **Model-free**, not **model-based** (build an explicit model of the environment)
- **Online**: learns and updates its policy/value function while interacting with the environment in real-time

Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known),

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)],$$

TD methods need to wait only until the next time step.

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

Temporal-Difference learning

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

$A \leftarrow$ action given by π for S

 Take action A , observe R, S'

$V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$

$S \leftarrow S'$

 until S is terminal

Temporal-Difference learning

SARSA: on-policy TD control

Uses the pattern of generalized policy iteration (GPI).

The first step is to learn an action-value function, $q_{\pi}(s, a)$, rather than a state-value function

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Choose A from S using policy derived from Q (e.g., ϵ -greedy)

 Loop for each step of episode:

 Take action A , observe R, S'

 Choose A' from S' using policy derived from Q (e.g., ϵ -greedy)

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$

$S \leftarrow S'; A \leftarrow A';$

 until S is terminal

Temporal-Difference learning

Q-learning: off-policy TD control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989), defined by

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

The diagram illustrates the Q-learning update equation with each term in a separate box and its meaning explained below:

- Updated action-value function**: $Q(S_t, A_t)$ (the result of the update)
- Current action-value function**: $Q(S_t, A_t)$ (the term being updated)
- Small step size**: α
- Reward**: R_{t+1}
- Discount factor**: γ
- Maximum Q-value for all possible actions in the next state**: $\max_a Q(S_{t+1}, a)$
- Current action-value function**: $Q(S_t, A_t)$ (the term being subtracted)

Temporal-Difference learning

Q-learning: off-policy TD control

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

The learned action-value function, Q , directly approximates q_* , the optimal action-value function, independent of the policy being followed.

- This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs.

The policy still has an effect in that it determines which state–action pairs are visited and updated.

- All that is required for correct convergence is that all pairs continue to be updated

Temporal-Difference learning

Q-learning: off-policy TD control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

 Choose A from S using policy derived from Q (e.g., ϵ -greedy)

 Take action A , observe R, S'

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

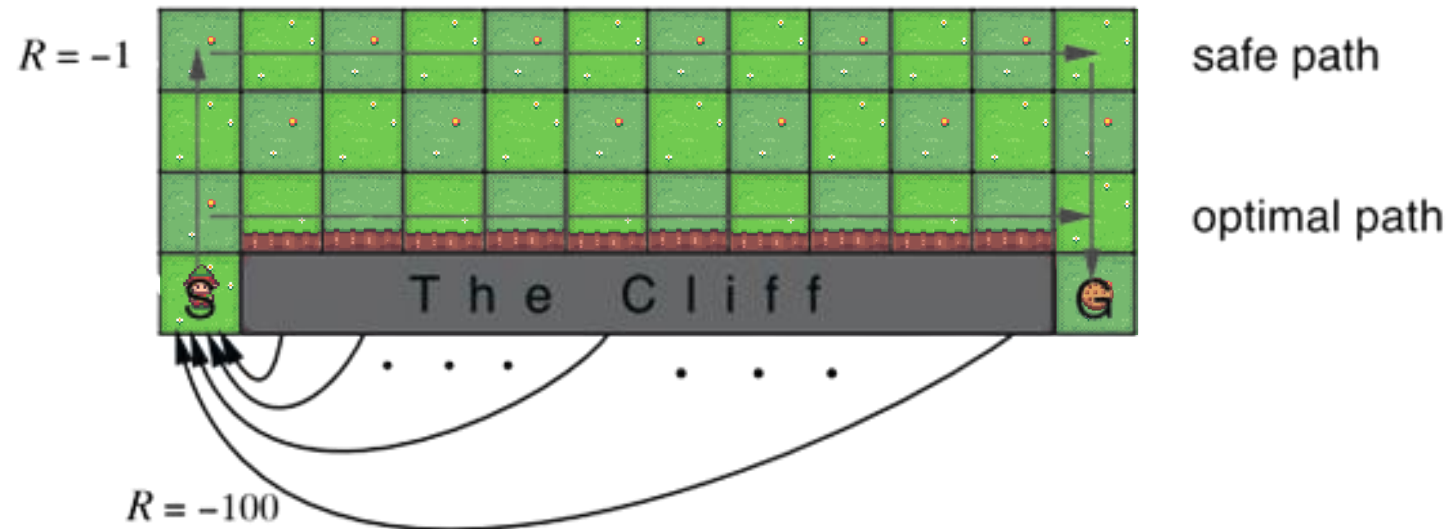
$S \leftarrow S'$

 until S is terminal

Temporal-Difference learning

SARSA vs. Q-learning

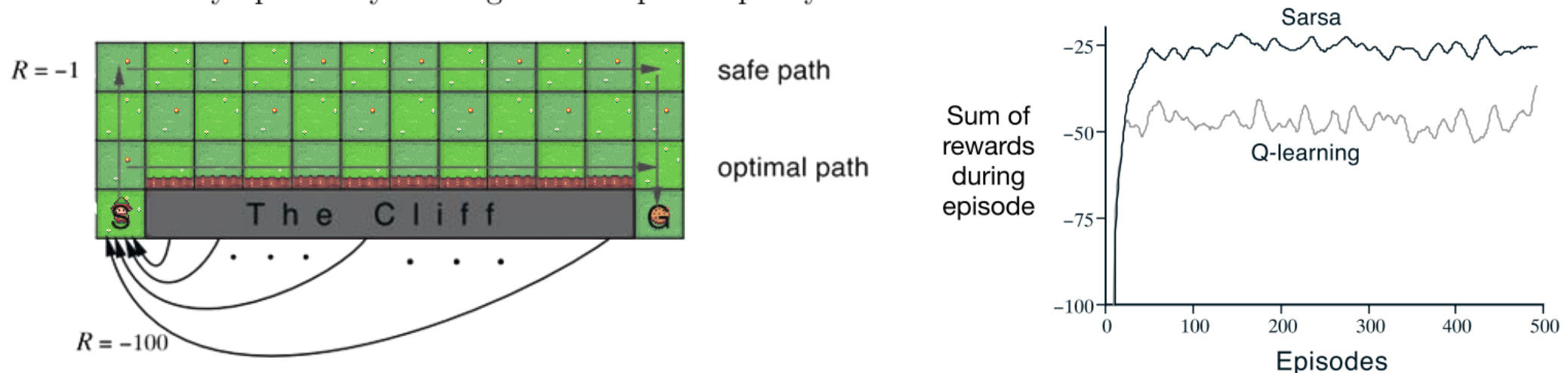
Example 6.6: Cliff Walking This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the gridworld shown in the upper part of Figure 6.4. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the region marked “The Cliff.” Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.



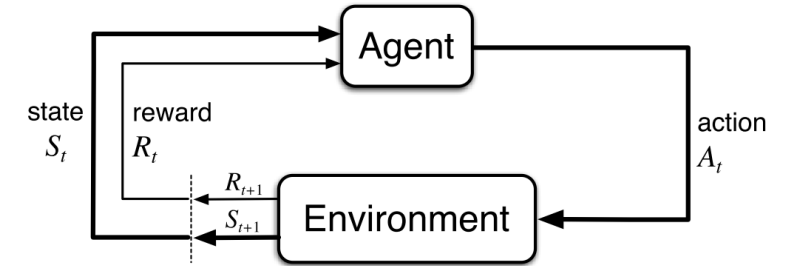
Temporal-Difference learning

SARSA vs. Q-learning

The lower part of Figure 6.4 shows the performance of the Sarsa and Q-learning methods with ε -greedy action selection, $\varepsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε -greedy action selection. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its on-line performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if ε were gradually reduced, then both methods would asymptotically converge to the optimal policy.



Example – tic-tac-toe



Learning a goal-directed behaviour, a **policy** π , requires 3 signals:

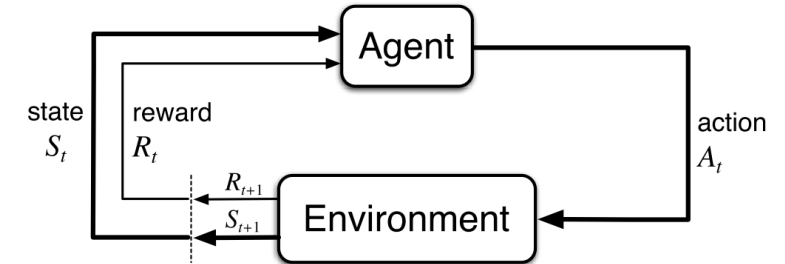
- represent choices of the agent (**action** A),
- represent where the decisions are made (**state** S),
- define the agent's goal (**reward** R).

The game rules and the opponent moves make the **environment**

X	O	O
O	X	X
		X

??

Example – tic-tac-toe



State

765 feasible configurations

X	O	O
O	X	X
		X

Action

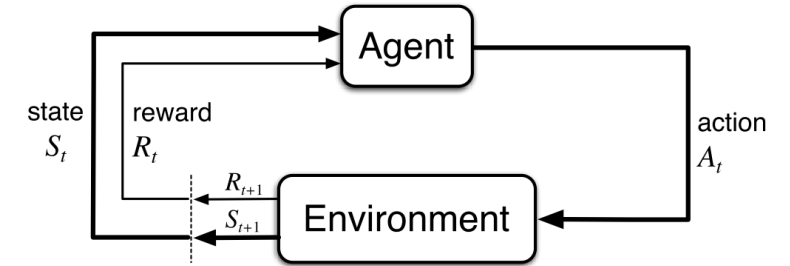
9 possible

1	2	3
4	5	6
7	8	9

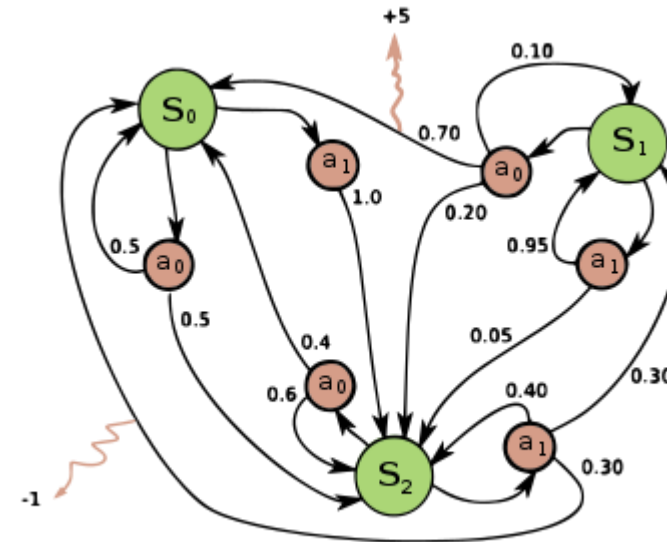
Reward

Win -> reward = 1
Lose -> reward = -1
0 otherwise

State Tabular representation

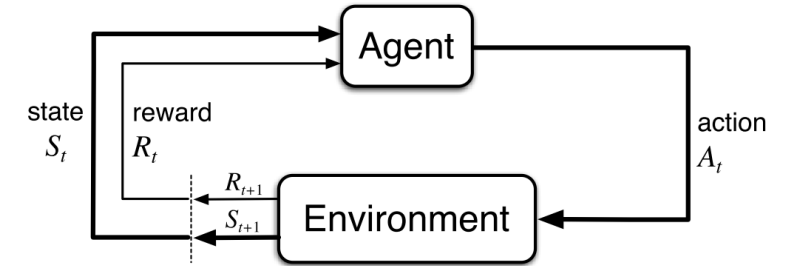


	a_0	a_1
s_0		
s_1		
s_2		



State

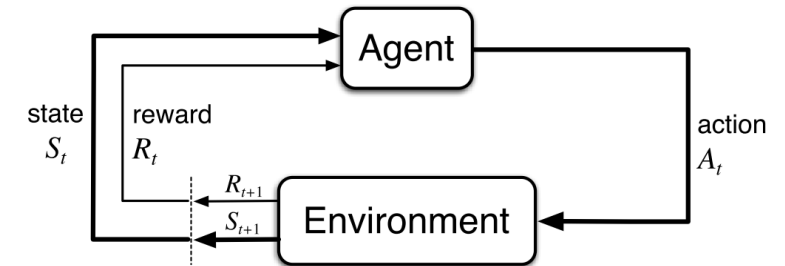
Tabular representation



X	O	O
O	X	X
		X

??

State Tabular representation



s	Q(s)									
<table><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></table>										
<table><tr><td>x</td><td>o</td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></table>	x	o								
x	o									
<table><tr><td></td><td>x</td><td>o</td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></table>		x	o							
	x	o								
...										
<table><tr><td>x</td><td>o</td><td>o</td></tr><tr><td>o</td><td>x</td><td>x</td></tr><tr><td></td><td></td><td></td></tr></table>	x	o	o	o	x	x				
x	o	o								
o	x	x								

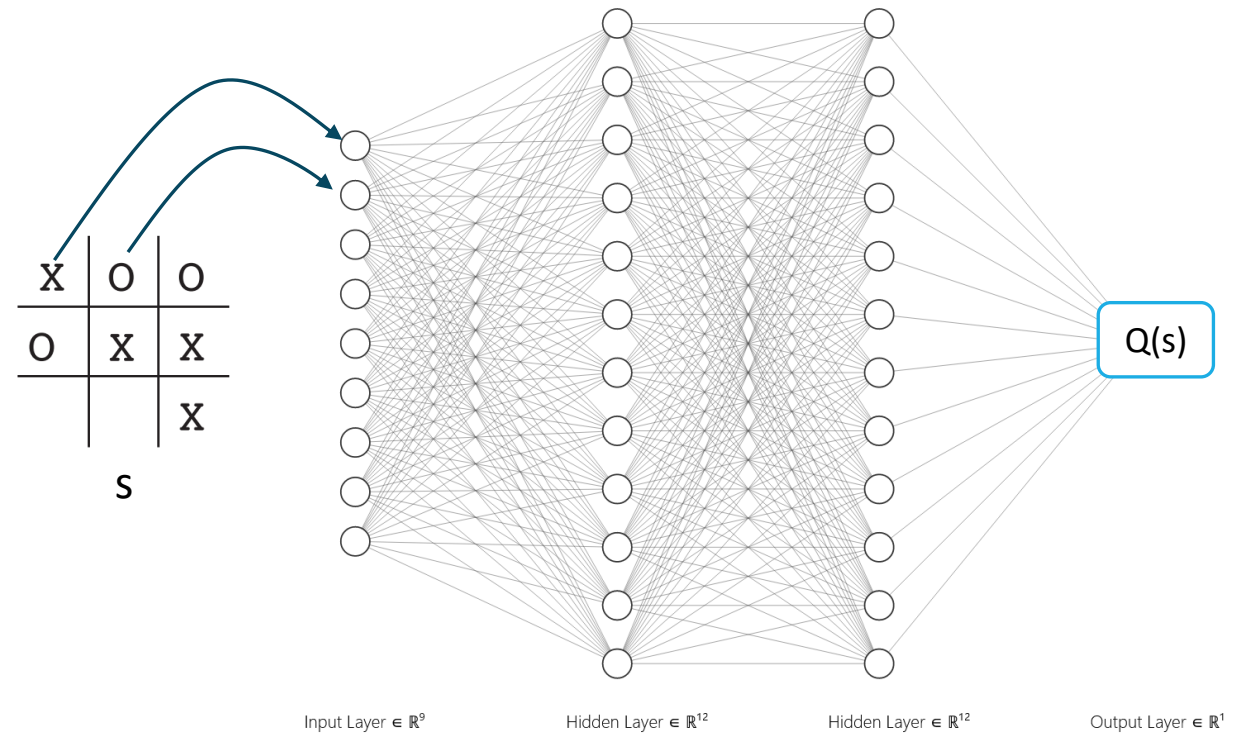
Q(s,a)	<div><div>1</div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div></div>	<div><div>2</div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div></div>	<div><div>3</div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div></div>	<div><div>4</div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div></div>	<div><div>5</div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div></div>	<div><div>6</div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div></div>	<div><div>7</div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div></div>	<div><div>8</div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div></div>	<div><div>9</div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div></div>
<div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div>									
<div><div><div>x</div><div>o</div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div>									
<div><div><div></div><div>x</div><div>o</div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div>									
...									
<div><div><div>x</div><div>o</div><div>o</div></div><div><div>o</div><div>x</div><div>x</div></div><div><div></div><div></div><div></div></div></div>									

Tabular methods

vs.

Approximate solutions

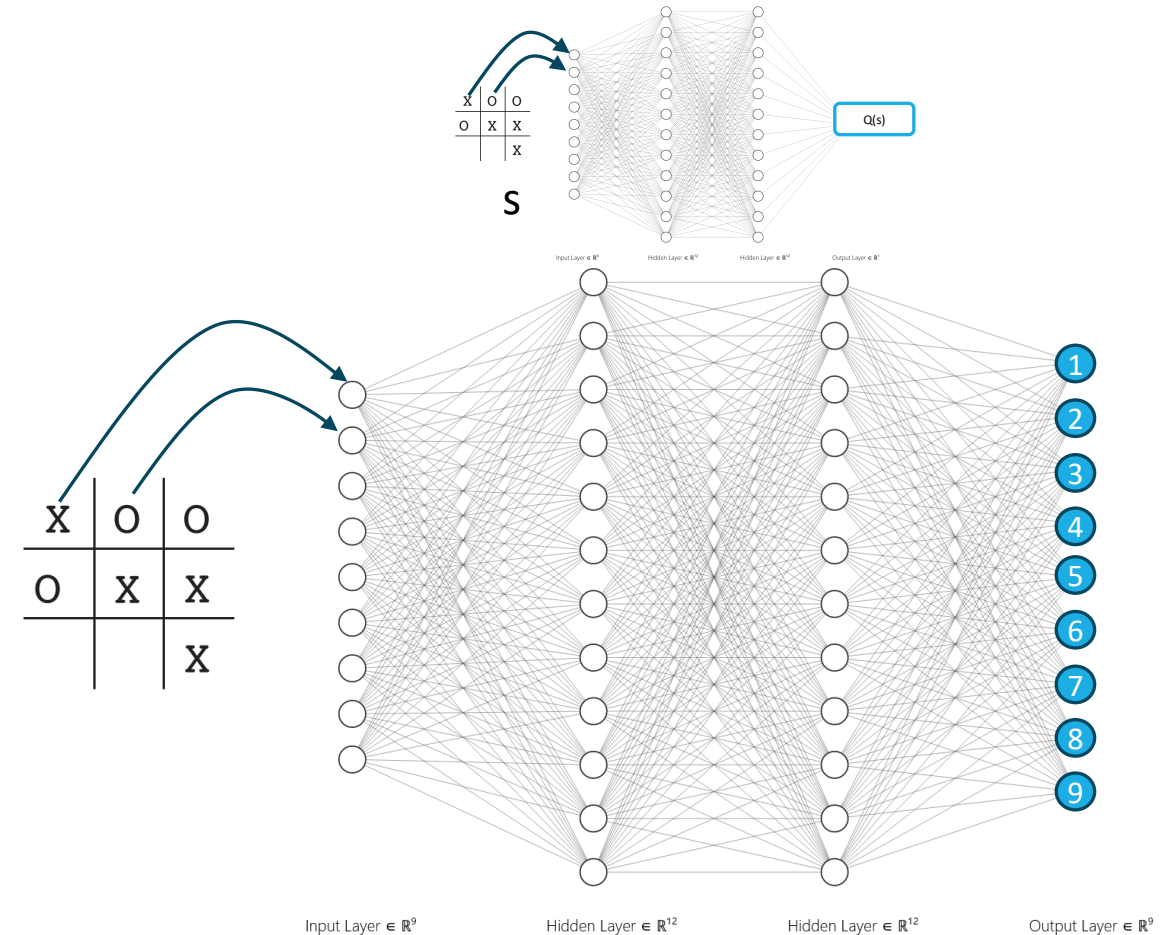
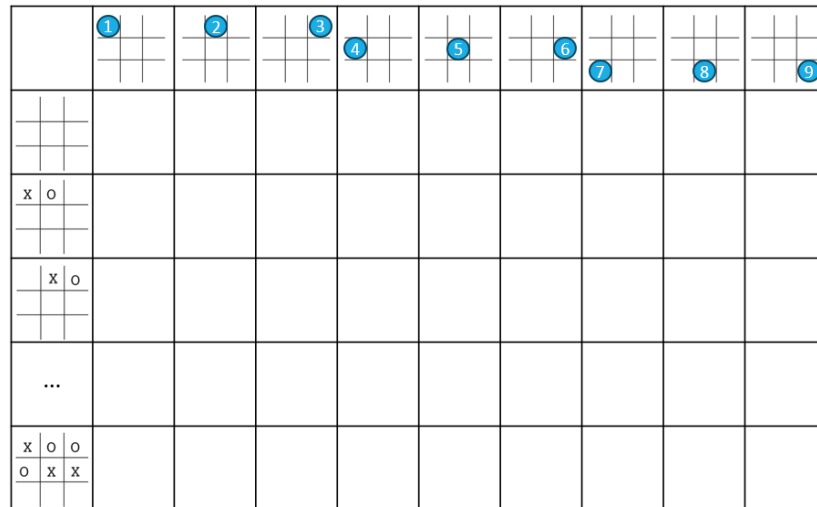
		1		2		3	4	5		6	7		8		9
x	o														
	x	o													
...															
x	o	o													
o	x	x													



Tabular methods

VS.

Approximate solutions



Tabular methods

vs.

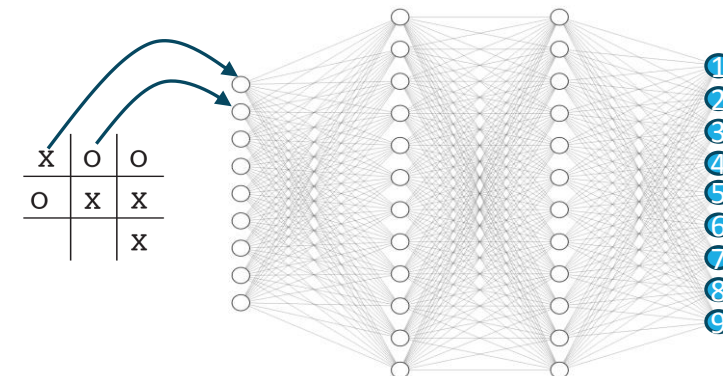
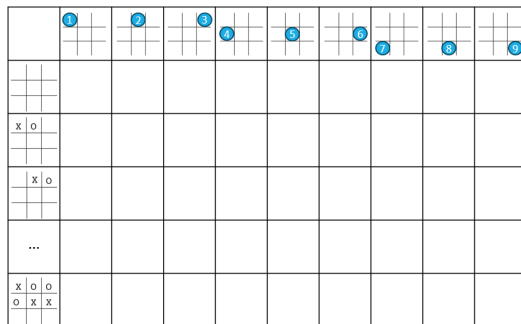
Approximate solutions

- Precise storage and retrieval, more accurate policies
- Easy to interpret
- Guarantee convergence

- Impractical for large and continuous state problems
- Faces exploration challenges with sparse rewards and complex state spaces

- Appropriate for large and continuous state spaces
- Able to generalize past experiences into never seen scenarios

- Introduce approximation errors, which can lead to suboptimal policies
- Training must reach optimal model configuration to avoid suboptimal policies



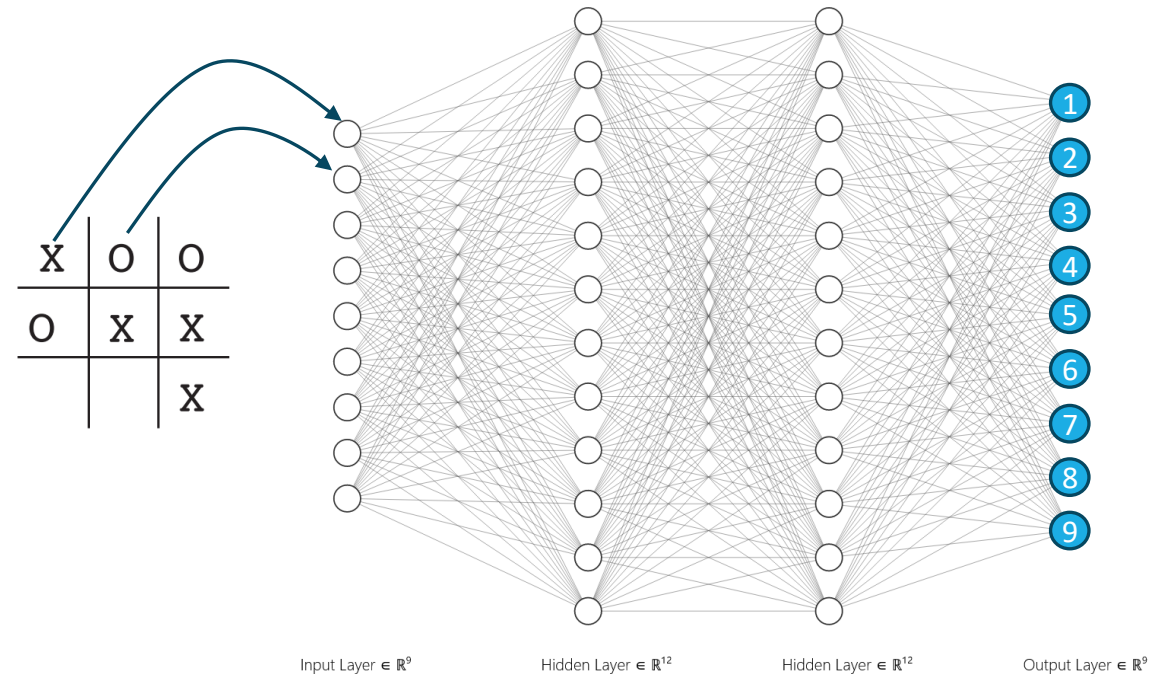
Modern examples

Deep Q-learning (DQN)

Model-free, online off-policy value-based method.

Same as Q-learning, but instead of a table a deep neural network is used.

Often uses a **memory replay** to store past experiences and use during training more than once.



Approximate solution methods

Policy gradient

$\Pr\{X=x\}$ -> probability
that a random variable X
takes the value of x

So far, all the methods have **learned the values of actions** and then **selected actions** based on their estimated action values.

Policy gradient methods instead **learn a parameterized policy** that can **select actions without consulting a value function**.

- Methods output the probability distribution over actions for a given state

Thus, we write

$$\pi(a|s, \boldsymbol{\theta}) = \Pr\{A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\}$$

for the probability that action a is taken at time t given that the environment is in state s at time t with parameter $\boldsymbol{\theta}$, where $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ is the policy's parameter vector.

If a method uses a learned value function as well, then the value function's weight vector is denoted $\mathbf{w} \in \mathbb{R}^d$

Modern examples

Advantage Actor Critic (A2C)

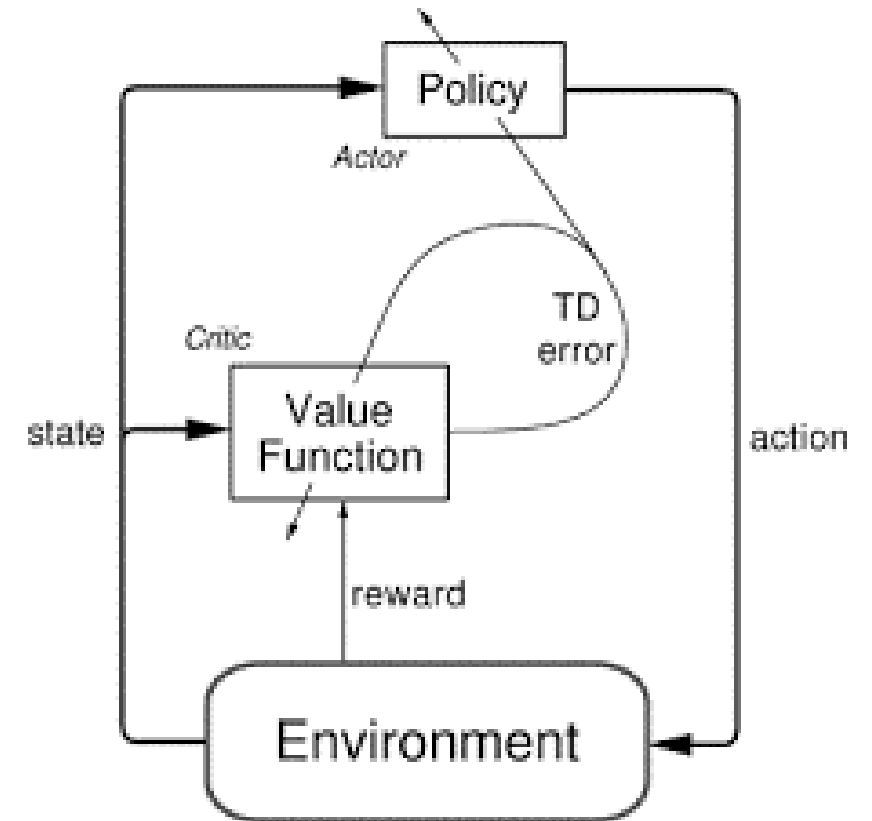
Combines policy gradient methods with value-based methods.

Actor Network: responsible for learning and representing the policy.

- Current state as input and outputs probabilities for each action.
- Uses policy gradients to encourage actions that lead to higher rewards

Critic Network: estimates the value function.

- Estimates the expected return (or value) of being in a state.
- Value function is used to compute advantages



Modern examples

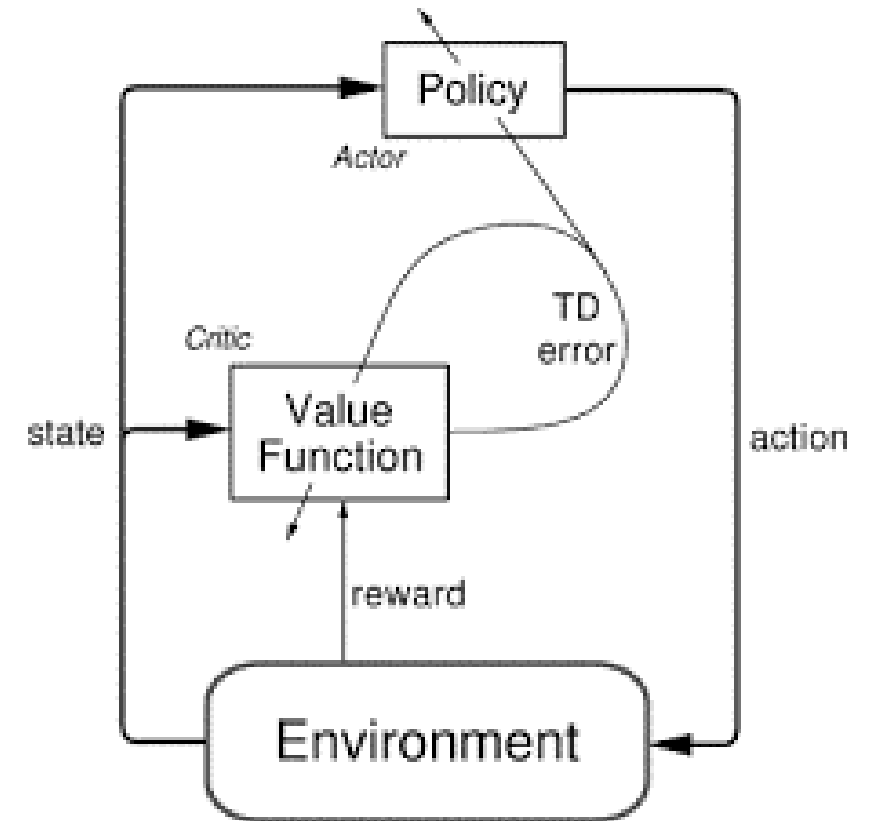
Advantage Actor Critic (A2C)

Advantage Estimation: The difference between observed and expected reward in each state.

- Indicates how much better or worse an action is than the average action taken in that state.
- Used to update the policy and value function, helping the agent focus on actions that lead to better-than-average rewards.

Entropy Regularization: measure of the policy's uncertainty.

- Adding an entropy term to the loss function encourages exploration by discouraging the policy from becoming too deterministic.
- It helps balance exploration and exploitation.



Modern examples

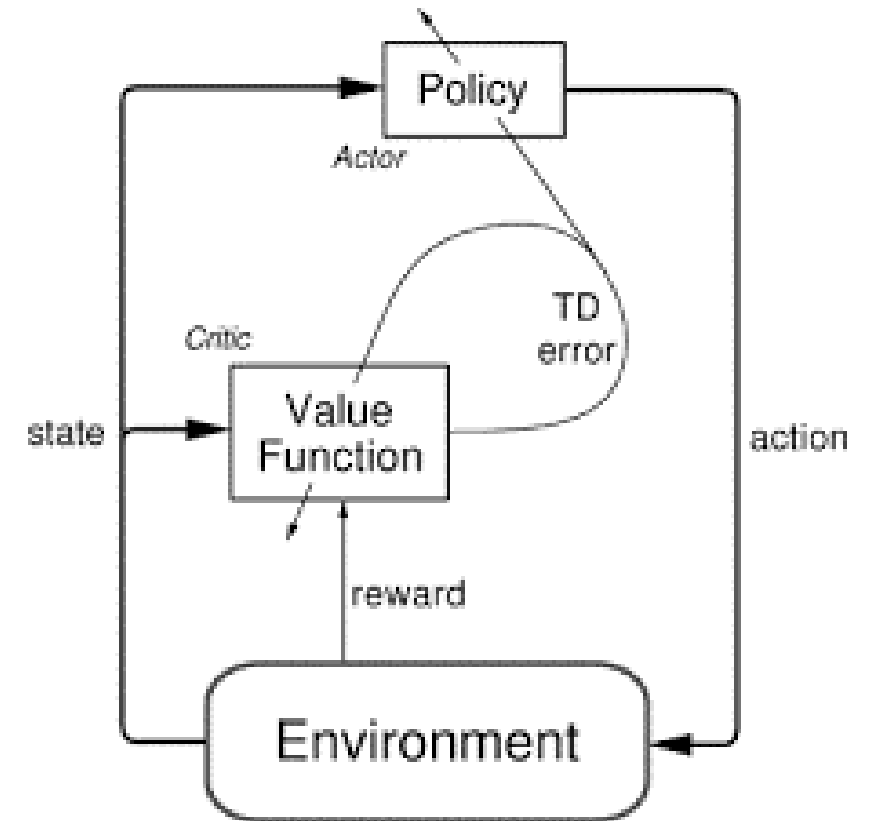
Advantage Actor Critic (A2C)

Synchronous Training and Parallelization:

- A2C can be parallelized to collect experiences from multiple agents simultaneously.
- Updates occur at the same time using the combined experiences from all agents.

Shared Parameters:

- The actor and critic networks often share some of their parameters.
- Allows both networks to learn useful features from the environment, which can enhance learning efficiency.



Modern examples

Proximal Policy Optimization

Trust region:

- PPO employs a **clipped** surrogate objective function, which **penalizes policy updates that deviate too much** from the previous policy.
- By clipping the objective function, the policy does not change too drastically, maintaining **stability in learning**.

Importance Sampling:

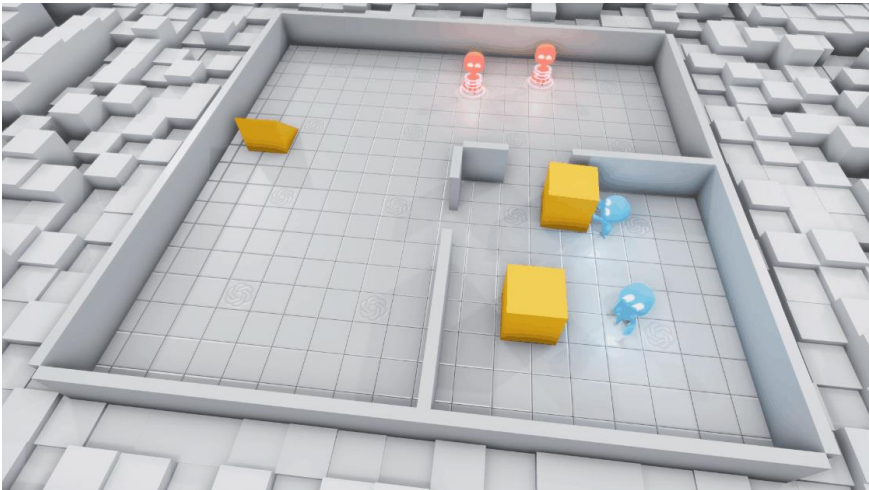
- Estimate an expectation under one probability distribution using samples generated from another probability distribution
- Utilizes old data more effectively by weighing the updates based on the probability of actions under the old and new policies.
- This helps in achieving a balance between exploration and exploitation.

Multiple Epochs:

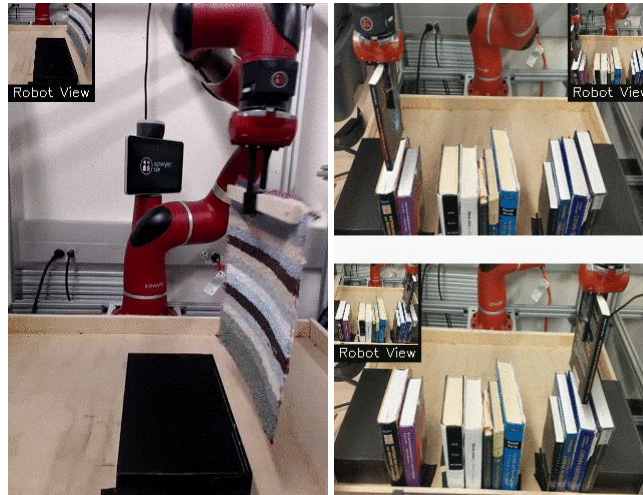
- Multiple optimization epochs on the collected data. In each epoch, the algorithm uses the data multiple times to perform policy updates. This process helps in leveraging the available data more effectively.

Real-world examples

Emergent tool use from multi-agent interaction (OpenAI, 2019)



End-to-End Deep Reinforcement Learning without Reward Engineering (BAIR, 2019)



From motor control to team play in simulated humanoid football (DeepMind, 2022)

