U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Artificial Intelligence

Lecture 12: Intelligent Agents – Introduction to Reinforcement Learning

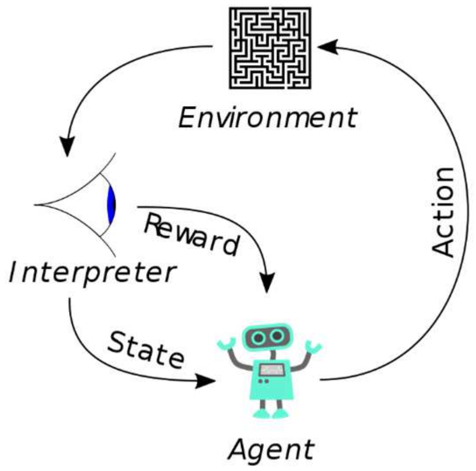
(slides baseados em "Cardoso, H.L., 2018" and Choudhary, 2019)

Luís Paulo Reis, Henrique L. Cardoso

lpreis@fe.up.pt, hlc@fe.up.pt

What is Reinforcement Learning?

- **Reinforcement Learning (RL)** is focused on goal-directed learning from interaction



- RL 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: it must discover which actions yield the most reward by trying them
  - Typically, actions may affect not only immediate reward but also the next situation and subsequent rewards
- The exploration-exploitation tradeoff
  - Agent must prefer actions that it knows to be effective – *exploit*
  - But to discover such actions, it has to try actions not selected before – *explore*

RL vs (Un)Supervised Learning

- **Different** from **supervised learning**
  - In interactive problems it is impractical to obtain examples of desired behavior
  - In uncharted territory, an agent must learn from its own experience
- **Different** from **unsupervised learning**
  - RL is trying to maximize a reward signal, not trying to find hidden structure in collections of unlabeled data
- RL explicitly considers the *whole* problem of a goal-directed agent interacting with an uncertain environment
  - Creating a behavior model while applying it in the environment
- RL is the closest form of ML to the kind of learning humans do

Elements of RL

- **Policy π**
  - How should the agent behave over time?
  - A policy is a (possibly stochastic) mapping from perceived states to actions
- **Reward signal r**
  - Defines the goal of the RL problem
  - On each time step, the environment sends a reward to the RL agent – the agent's goal is to maximize the total reward received over the long run
- **Value function v**
  - Specifies what is good in the long run
  - The value of a state is the total amount of reward an agent can expect to accumulate from that state onwards (it takes into account future rewards)

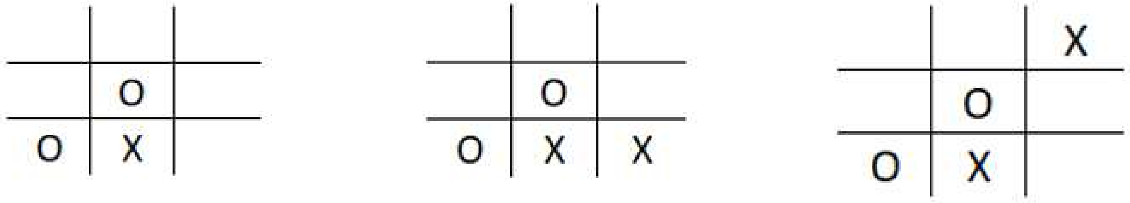
→ We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward over the long run

Agent Environment Interface

- Can we define a rule-based framework to design an efficient bot for playing a game such as tic-tac-toe?
- We sure can, but we will have to hardcode a lot of rules for each of the possible situations that might arise in a game.
- However, an even more interesting question to answer is:
  - Can we train the bot to learn by playing against you several times?
  - And that without being explicitly programmed to play a game efficiently?

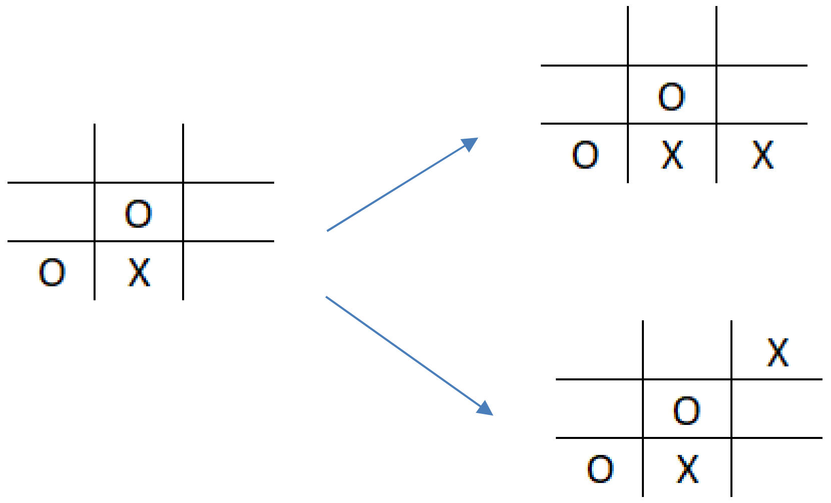
State, Action, Policy, Reward

- Bot needs to understand the situation it is in. A tic-tac-toe has 9 spots to fill with an X or O. Each different possible combination in the game will be a different situation for the bot, based on which it will make the next move. Each scenario is a different **state.**



- Once the state is known, the bot must take an **action** in a way it considers to be optimum to win the game (**policy**)
- This move will result in a new scenario with new combinations of O's and X's which is a **new state** and a numerical **reward** will be given based on the quality of move with the goal of winning the game (**cumulative reward**)

Rewards

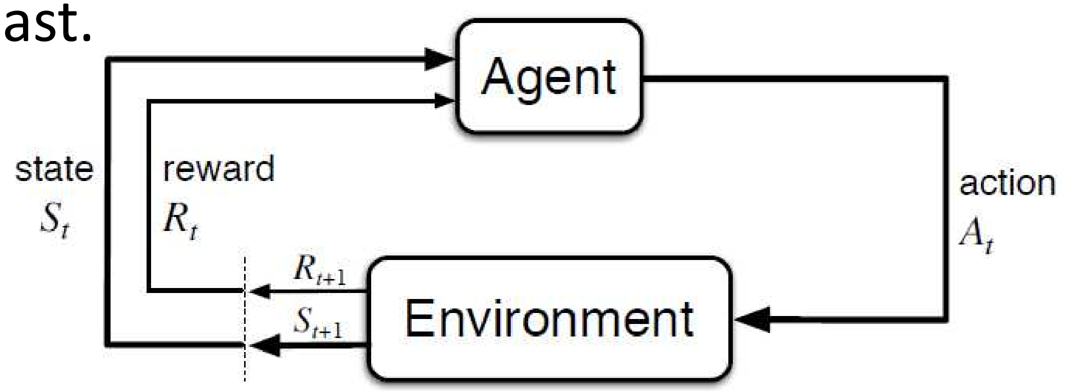


Markov Decision Process

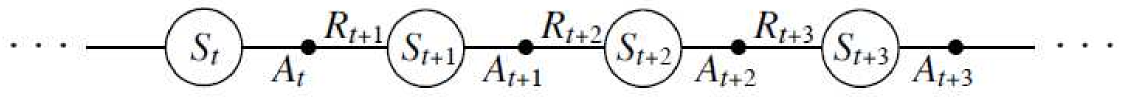
- Markov Decision Process (MDP) model contains:
  - A set of possible world states S
  - A set of possible actions A
  - A real valued reward function R(s,a)
  - A description T of each action’s effects in each state
- “Memoryless’ property”
  – Any random process in which the probability of being in a given state depends only on the previous state, is a markov process.

Markov Decision Process

- In the Markov decision process setup, the environment’s response at time t+1 depends only on the state and action representations at time t, and is independent of whatever happened in the past.



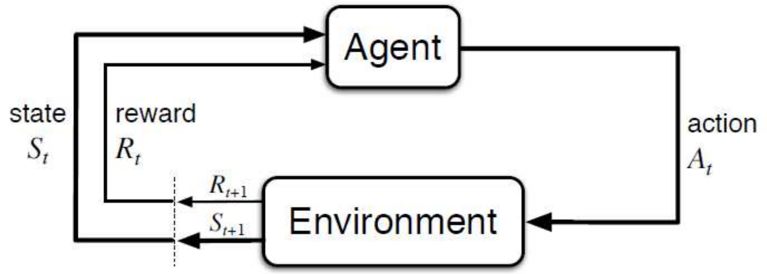
- St: State of the agent at time t
- At: Action taken by agent at time t
- Rt: Reward obtained at time t



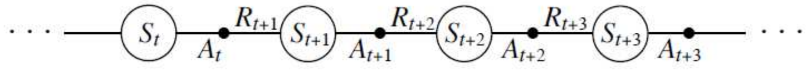
Markov Decision Process

- Iteration at each time step wherein the agent receives a reward R_{t+1} and ends up in state S_{t+1} based on its action A_t at a particular state S_t.
- The overall goal for the agent is to maximise the cumulative reward it receives in the long run. Total reward at any time instant t is given by:

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



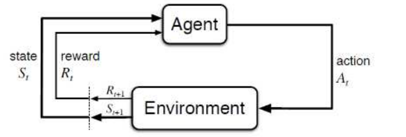
- St: State of the agent at time t
- At: Action taken by agent at time t
- Rt: Reward obtained at time t



Markov Decision Process

- Overall goal for the agent is to maximise the cumulative reward it receives in the long run. Total reward at any time instant t is given by:

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



- Additional concept of discounting. Basically, we define γ as a discounting factor and each reward after the immediate reward is discounted by this factor as follows:

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

- For discount factor < 1, the rewards further in the future are getting diminished. This can be understood as a tuning parameter which can be changed based on how much one wants to consider the long term (γ close to 1) or short term (γ close to 0)

State Value Function

- Can we use the reward function defined at each time step to define how good it is, to be in a given state for a given policy?
- **The value function denoted as v(s) under a policy π represents how good a state is for an agent to be in.**
- What is the average reward that the agent will get starting from the current state under policy π?

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

- E represents the expected reward at each state if the agent follows policy π and *S* represents the set of all possible states.
- Policy is the mapping of probabilities of taking each possible action at each state (π(a/s)). The policy might also be deterministic when it tells you exactly what to do at each state and does not give probabilities.

Optimum Policy

- Value function is maximized for each state:

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) \quad \forall s \in \mathbb{S}$$

- State-Action Value Function
  - How good an action is at a particular state?
  - Q-value: value of action *a,* in state *s,* under a policy π:

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

Expected return the agent will get if it takes action At at time t, given state St, and thereafter follows policy π

Bellman Expectation Equation

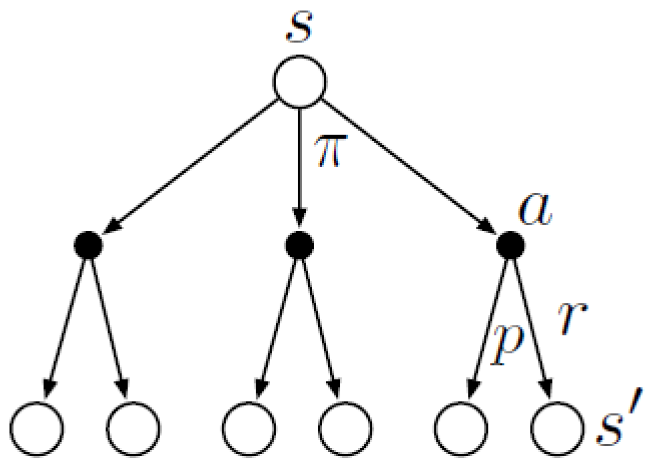
- Value function for a given policy π represented in terms of the value function of the next state.

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

- Choose an action a, with probability π(a/s) at the state s, which leads to state s’ with probability p(s’/s,a). This gives a reward [r + γ*v_π(s)]

Bellman Expectation Equation

- Bellman expectation 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

- We have n (number of states) linear equations with unique solution to solve for each state s

Bellman Optimality Equation

- Find the optimal policy, which when followed by the agent gets the maximum cumulative reward:
  - Find a policy π, such that for no other π can the agent get a better expected return
  - Find a policy which achieves maximum value for each state

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) \quad \forall s \in \mathbb{S}$$

We might not get a unique policy, as under any situation there can be 2 or more paths that have the same return and are still optimal.

$$v_*(s) \doteq \max_{\pi} v_\pi(s) \qquad q_*(s,a) \doteq \max_{\pi} q_\pi(s,a)$$

Bellman Optimality Equation

- Optimal value function can be obtained by finding the action *a* which will lead to the maximum of q*:
  - Bellman optimality equation for v*

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

- Bellman optimality equation says that the value of each state under an optimal policy must be the return the agent gets when it follows the best action as given by the optimal policy.
- For optimal policy π*, the optimal value function is given by:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \end{aligned}$$

Bellman Optimality Equation

- Given a value function q*, we can recover an optimum policy as follows:

$$\begin{aligned}\pi'(s) &\doteq \underset{a}{\arg\max}\, q_\pi(s,a) \\ &= \underset{a}{\arg\max}\, \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a]\end{aligned}$$

- The value function for optimal policy can be solved through a non-linear system of equations.
- We can can solve these efficiently using iterative methods like dynamic programming

Policy Evaluation

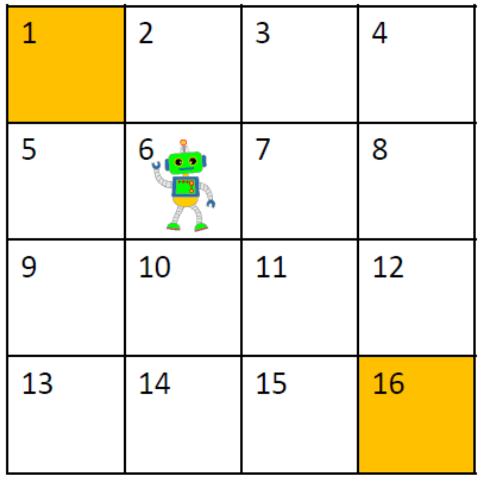
- Policy evaluation answers the question of how good a policy is. Given an MDP and an arbitrary policy π, we compute the state-value function
- Turn bellman expectation equation to an update

$$v_{k+1}(s) \doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s]$$

- To produce each successive approximation vk+1 from vk, iterative policy evaluation applies the same operation to each state s. It replaces 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, until it converges to the true value function of a given policy π

Policy Evaluation

- Grid World



actions

Reward is -1 for all transition

- A bot is required to traverse a grid of 4×4 dimensions to reach its goal (1 or 16). Each step is associated with a reward of -1.
- There are 2 terminal states: 1 and 16 and 14 non-terminal states given by [2,3,….,15].
- Consider a random policy for which, at every state, the probability of every action {up, down, left, right} is equal to 0.25. We start initializing v_0 for the random policy to all 0s.

Policy Evaluation

- Grid World

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

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

$$
\begin{aligned}
v_1(6) &= \sum_{a\in\{u,d,l,r\}} \pi(a|6) \sum_{s',r} p(s',r|6,a)[r+\gamma v_0(s')] \\
&= \sum_{a\in\{u,d,l,r\}} \underbrace{\pi(a|6)}_{=0.25\ \forall a} \sum_{s'} p(s'|6,a)[\underbrace{r}_{=-1} + \underbrace{\gamma v_0(s')}_{=0\ \forall s'}] \\
&= 0.25 * \{-p(2|6,u) - p(10|6,d) - p(5|6,l) - p(7|6,r)\} \\
&= 0.25 * \{-1-1-1-1\} \\
&= -1
\end{aligned}
$$

$$\Rightarrow v_1(6) = -1$$

for all non-terminal states, $v_1(s) = -1$.

For terminal states $p(s'/s,a) = 0$ and hence $v_k(1) = v_k(16) = 0$ for all k.

So v_1 for the random policy is given by:

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

Policy Evaluation

- Grid World

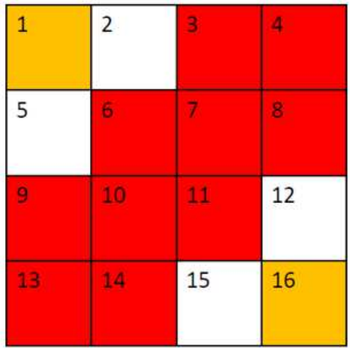
For $v_2(s)$ with γ - discounting factor 1

$$v_2(6) = \sum_{a\in\{u,d,l,r\}} \underbrace{\pi(a|6)}_{= 0.25\ \forall a} \sum_{s'} p(s'|6,a)[\underbrace{r}_{=-1} + \gamma \underbrace{v_1(s')}_{=\begin{cases}-1, s'\in S\\ 0, s'\in S^+\setminus S\end{cases}}]$$

$$= 0.25 * \{p(2|6,u)[-1-\gamma] + p(10|6,d)[-1-\gamma] + p(5|6,l)[-1-\gamma] + p(7|6,r)[-1-\gamma]\}$$

$$\overset{\gamma=1}{=} 0.25 * \{-2-2-2-2\}$$

$$= -2$$



All the red states are identical to 6 for the purpose of calculating the value function. v2(s) = -2.
For the remaining states, i.e., 2, 5, 12 and 15, v2 can be calculated as follows:

$$v_2(2) = \sum_{a\in\{u,d,l,r\}} \underbrace{\pi(a|2)}_{= 0.25\ \forall a} \sum_{s'} p(s'|2,a)[\underbrace{r}_{=-1} + \gamma \underbrace{v_1(s')}_{=\begin{cases}-1, s'\in S\\ 0, s'\in S^+\setminus S\end{cases}}]$$

$$= 0.25 * \{p(2|2,u)[-1-\gamma] + p(6|2,d)[-1-\gamma] + p(1|2,l)[-1-\gamma*0] + p(3|2,r)[-1-\gamma]\}$$

$$\overset{\gamma=1}{=} 0.25 * \{-2-2-1-2\}$$

$$= -1.75$$

$$\Rightarrow v_2(2) = -1.75$$

$\Rightarrow v_2$ for the random policy:

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

Policy Evaluation

- If we repeat this step several times, we get v_{π}:

$k = 0$

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

$\cdots$

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$\cdots$

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

$\leftarrow v_{\pi}$

Policy Improvement

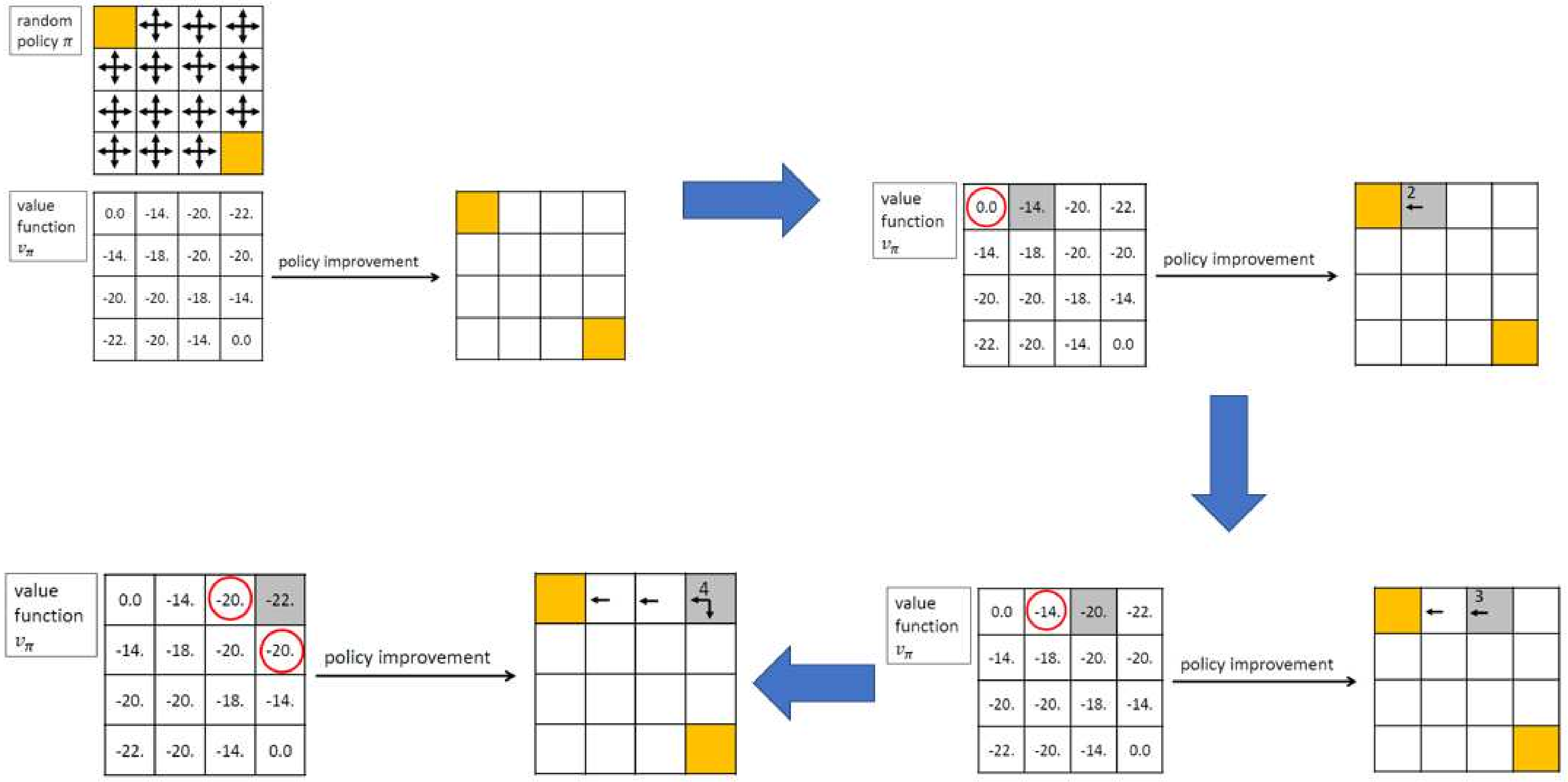
- Using policy evaluation we have determined the value function v for an arbitrary policy π (how good our current policy is)
- Now for some state s, we want to understand what is the impact of taking an action *a* that does not pertain to policy π.
- Let's say we select *a* in *s,* and after that we follow the original policy π.
- The value of this way of behaving is represented as:

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]. \end{aligned}$$

- If this happens to be greater than the value function $v_\pi(s)$, it implies that the new policy π' would be better to take.
- We do this iteratively for all states to find the best policy. Note that in this case, the agent would be following a greedy policy in the sense that it is looking only one step ahead.

Policy Improvement

Using v_{π}, the value function obtained for random policy π, we can improve upon π by following the path of highest value

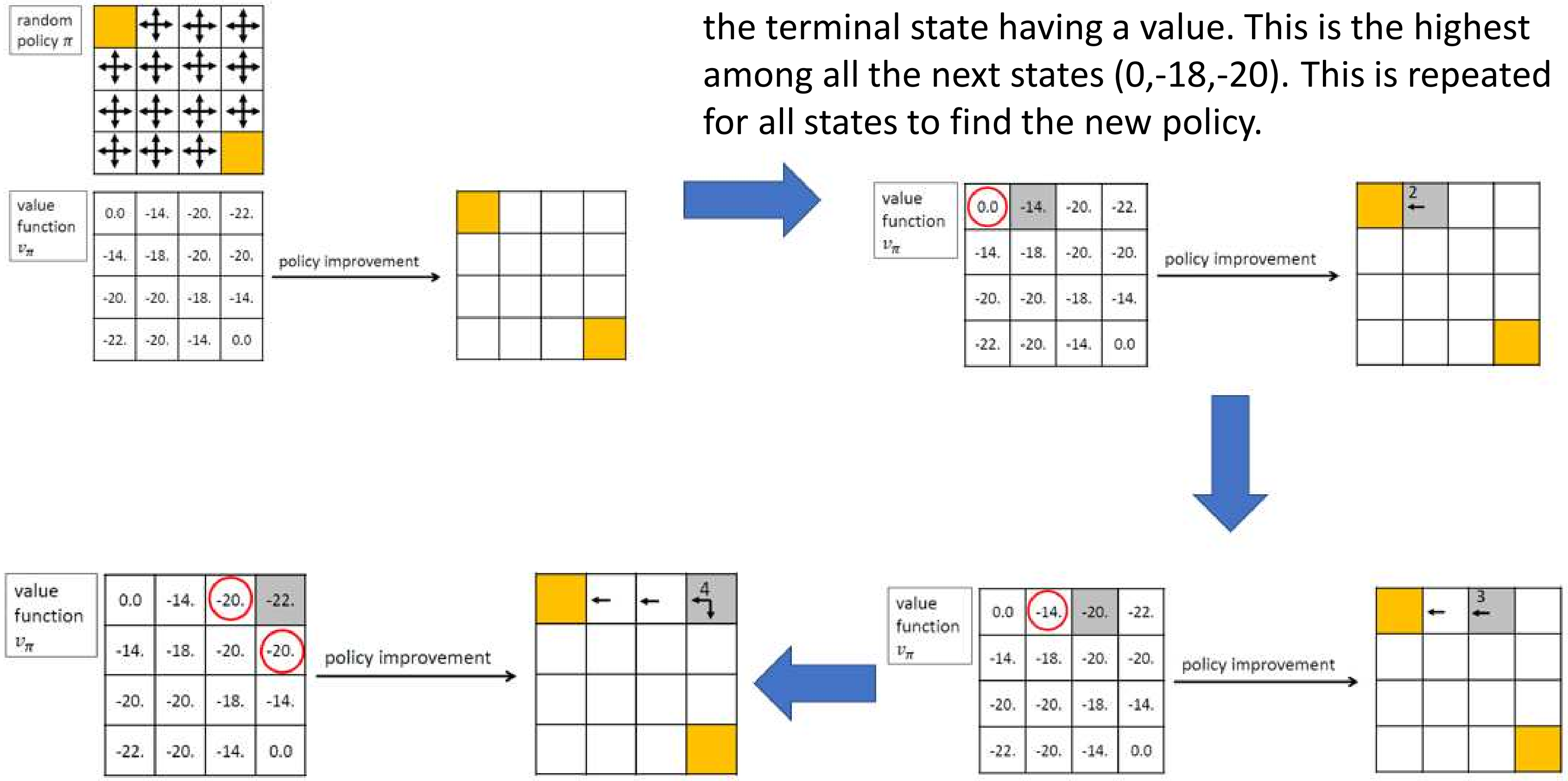


0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Policy Improvement

We start with an arbitrary policy, and for each state one step look-ahead is done to find the action leading to the state with the highest value. This is done successively for each state.

For state 2, the optimal action is left which leads to the terminal state having a value. This is the highest among all the next states (0,-18,-20). This is repeated for all states to find the new policy.

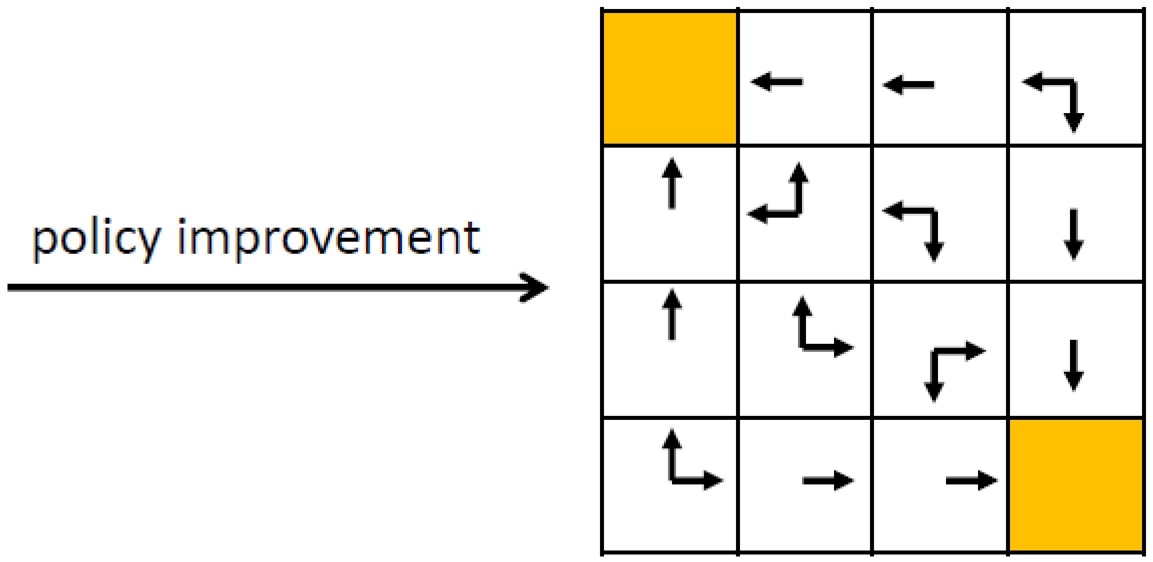


0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Policy Improvement

After the policy improvement step using v_π, we get the new policy π':

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

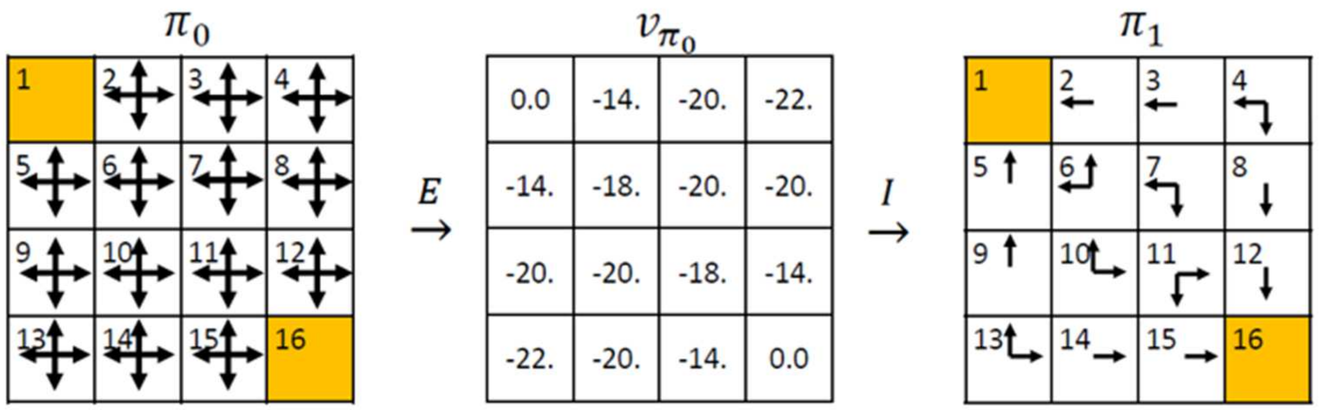


Policy Iteration

Once the policy has been improved using v_π to yield a better policy π’, we can then compute v_π’ to improve it further to π’’.

Repeated iterations are done to converge approximately to the true value function for a given policy π (policy evaluation).

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



0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

New policy is sure to be an improvement over the previous one and given enough iterations, it will return the optimal policy

Policy Iteration

- Problem:
  - Each iteration in policy iteration itself includes another iteration of policy evaluation that may require multiple sweeps through all the states.
  - Lots of possible solutions such as Value Iteration

- In the gridworld example around k = 10, we were already in a position to find the optimal policy
- So, instead of waiting for the policy evaluation step to converge exactly to the value function v_π, we could stop earlier

Value Iteration

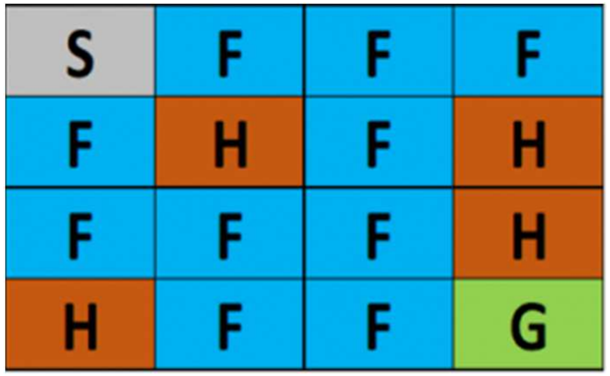
- We can also get the optimal policy with just 1 step of policy evaluation followed by updating the value function repeatedly (but this time with the updates derived from bellman optimality equation):

$$v_{k+1}(s) \doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a]$$

- This is identical to the bellman update in policy evaluation, with the difference of taking the maximum over all actions
- Once the updates are small enough, we can take the value function obtained as final and estimate the optimal policy corresponding to that

Frozen Lake Environment

- The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.
- The surface is described using a grid like the following:



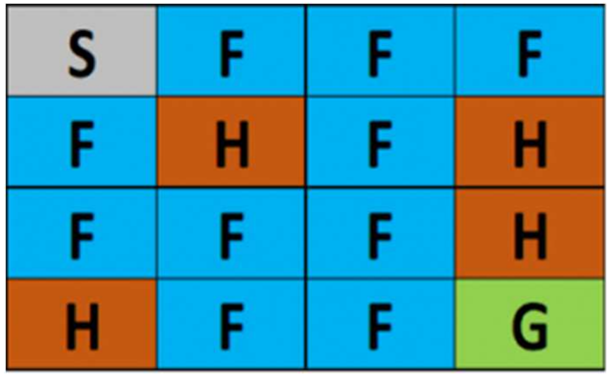
S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

(S: starting point, safe), (F: frozen surface, safe), (H: hole, fall to your doom), (G: goal)

Frozen Lake Environment

- The idea is to reach the goal from the starting point by walking only on frozen surface and avoiding all the holes.
- Once gym library is installed, we may open a jupyter notebook

```
import gym
import numpy as np
env = gym.make('FrozenLake-v0')
```



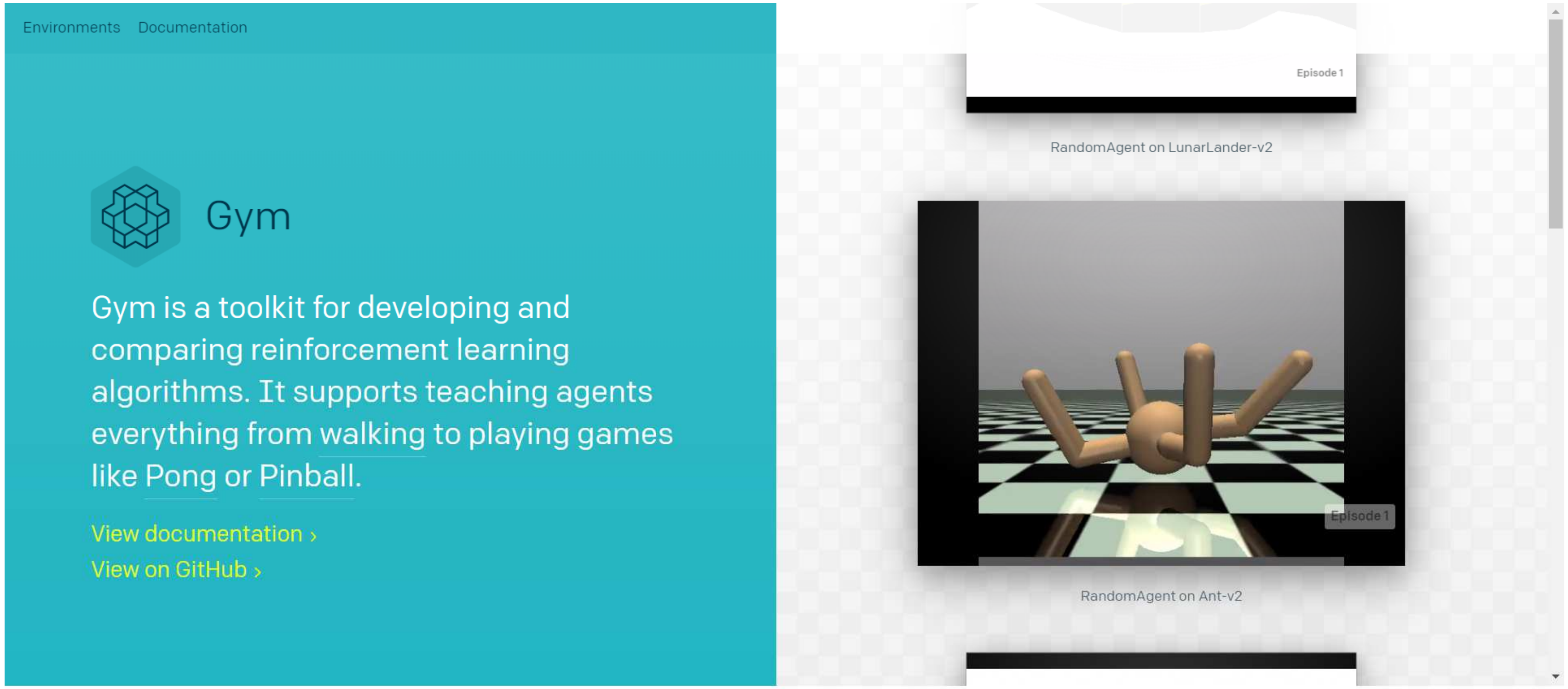
(S: starting point, safe), (F: frozen surface, safe), (H: hole, fall to your doom), (G: goal)

Policy Iteration in python

- **Policy:** 2D array of a size n(S) x n(A), each cell represents a probability of taking action a in state s.
- **Environment:** Initialized OpenAI gym environment object
- **Discount_factor:** MDP discount factor
- **Theta:** A threshold of a value function change. Once the update to value function is below this number
- **Max_iterations**: Maximum number of iterations to avoid letting the program run indefinitely
- This function will return a vector of size nS, which represent a value function for each state.

Open AI Gym

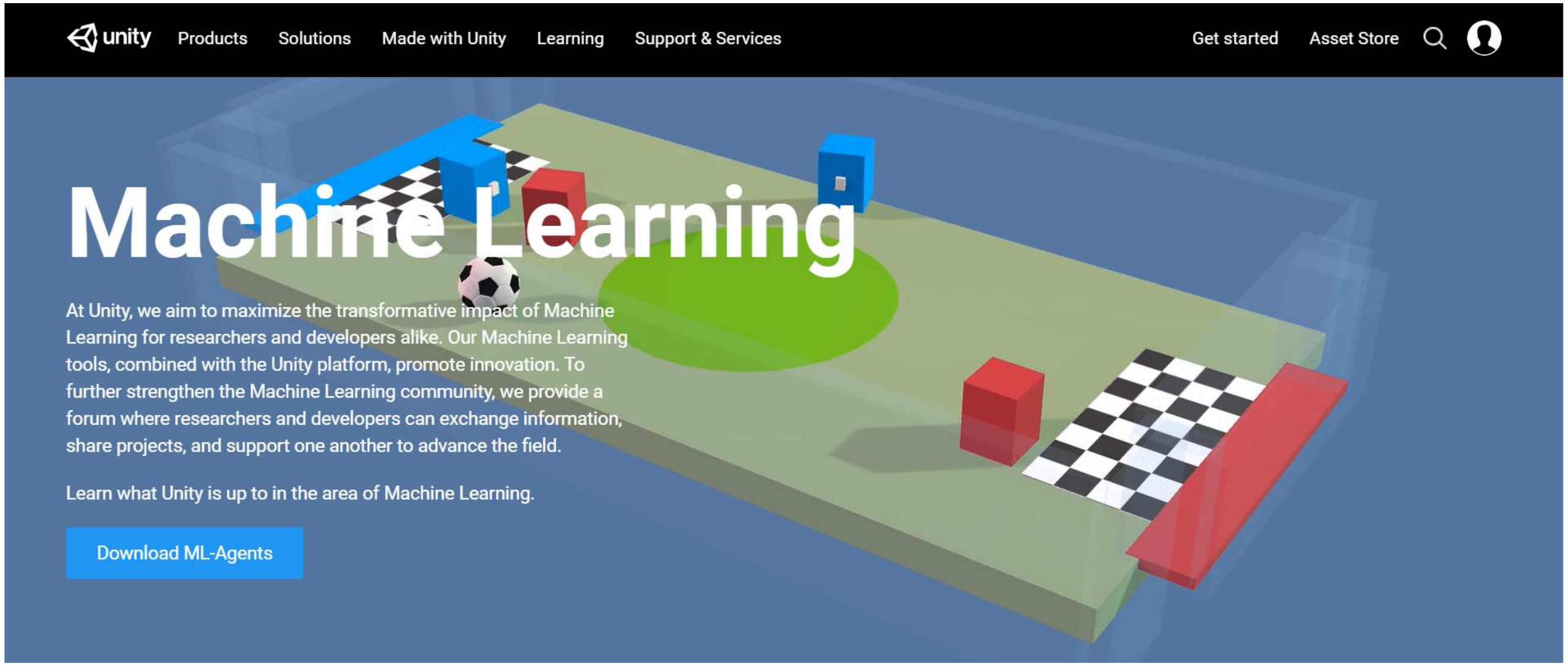
Gym is a toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of the agent, and is compatible with any numerical computation library, such as TensorFlow or Theano.



Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.

Unity ML-Agents

Aimed to maximize the transformative impact of Machine Learning for researchers and developers. Essentially Deep Reinforcement Learning Tools with several application but targeted to games



Machine Learning

At Unity, we aim to maximize the transformative impact of Machine Learning for researchers and developers alike. Our Machine Learning tools, combined with the Unity platform, promote innovation. To further strengthen the Machine Learning community, we provide a forum where researchers and developers can exchange information, share projects, and support one another to advance the field.

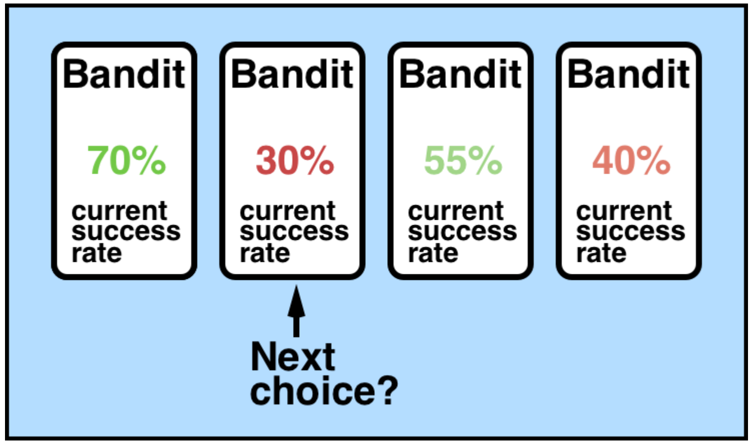
Learn what Unity is up to in the area of Machine Learning.

Bandit Problems

- A simple setting with a single state

- *K*-armed bandit problem
  - There are *k* different actions
  - After each action a numerical reward is received from a stationary probability distribution
  - Each action has a *value* – its expected or mean reward, not known by the agent: $q_*(a) \doteq \mathbb{E}[R_t | A_t = a]$
  - The agent *estimates*, at time step t, the value of an action a: $Q_t(a)$

Bandit Problems



- Selecting *greedy* actions (whose estimated value is greatest): exploiting
- Selecting non-greedy actions: exploring
  - Improve estimates of non-greedy actions' value
- Lower reward in the short run (during *exploration*), but higher in the long run – after discovering the best actions, we can *exploit* them many times

Estimating Action Values

- Sample average:

$$Q_t(a) \doteq \frac{\sum_{i=1}^{t-1} R_i \cdot 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}}$$

- Update rule:

$$Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n]$$

$$\boxed{NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]}$$

  - The target indicates a desirable direction in which to move
  - The *step-size* parameter changes from time step to time step

- Giving more weight to recent rewards – constant *step-size* parameter:

$$Q_{n+1} \doteq Q_n + \alpha[R_n - Q_n]$$

where $\alpha \in (0,1]$

Action Selection

- *Greedy* action selection (always exploits):
  - $A_t \doteq \operatorname*{argmax}_a Q_t(a)$

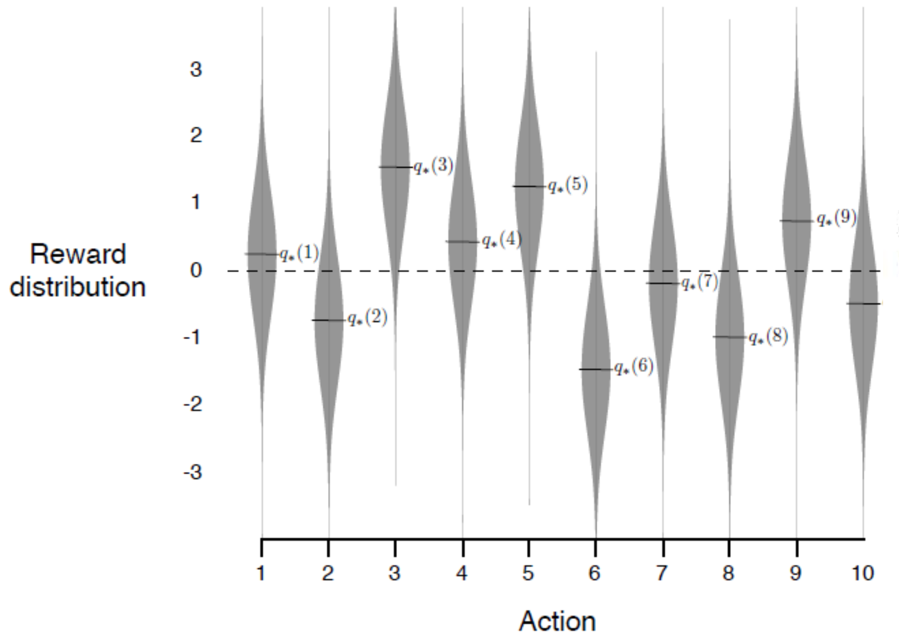
- *ε-greedy* action selection: behave greedily most of the time, but with small probability ε select randomly from among all the actions
  - $Q_t(a)$ will converge to $q_*(a)$ if a is selected sufficiently often

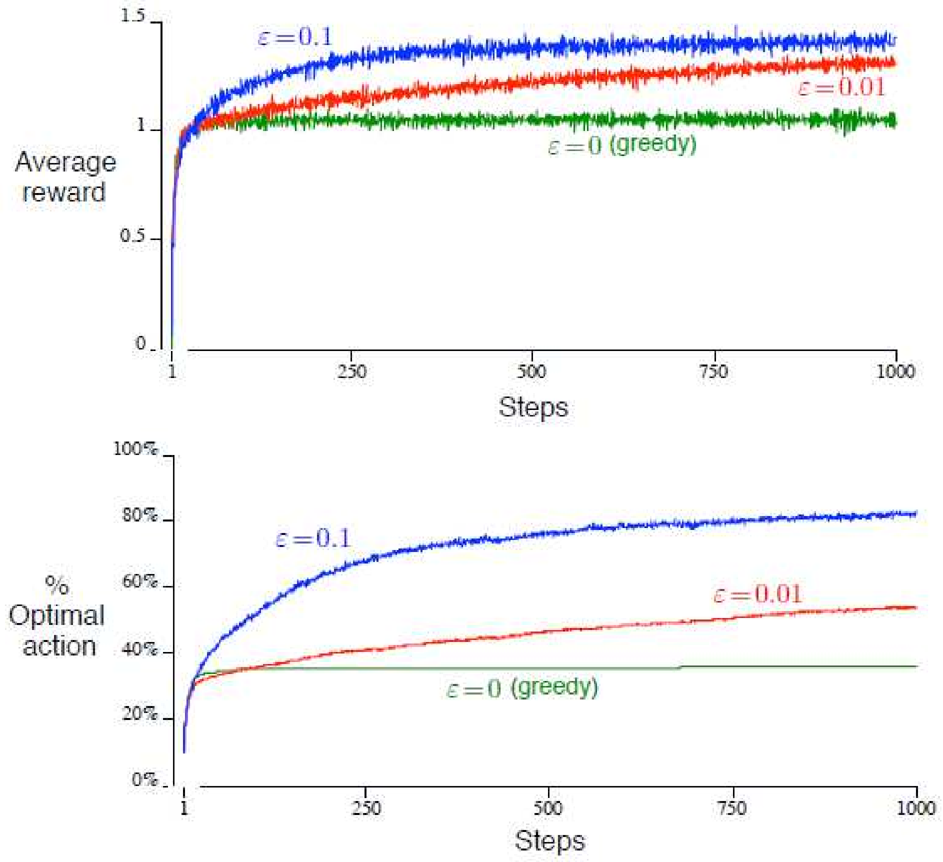
- *Soft-max* action selection (Boltzmann distribution):

$$Pr\{A_t = a\} \doteq \frac{e^{Q_t(a)\tau}}{\sum_{b=1}^{k} e^{Q_t(b)/\tau}}$$

where τ is a temperature parameter: if high, actions will tend to be equiprobable; if low, action values matter more; if $\tau \to 0$, then we have greedy action selection

The 10-armed Testbed

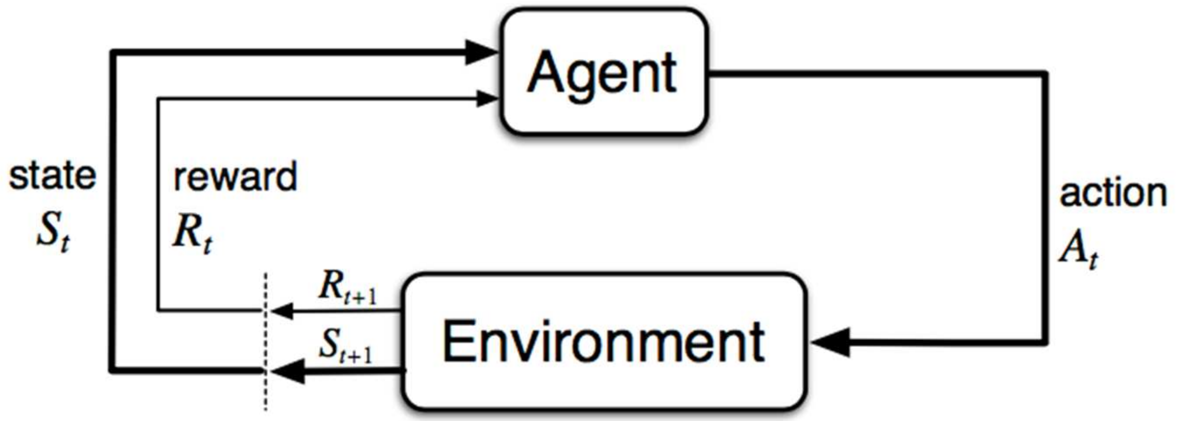




Markov Decision Processes

- As seen before, In the general setting we have many states
- Markov Decision Processes (MDP) are a classical formalization of sequential decision making
  - Actions influence not just immediate rewards, but also subsequent situations (states) and thus future rewards
- In a finite MDP, there is a finite number of states, actions and rewards
- In MDPs we estimate the value $q_*(s, a)$

Agent-Environment Interface



- Dynamics of the MDP:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

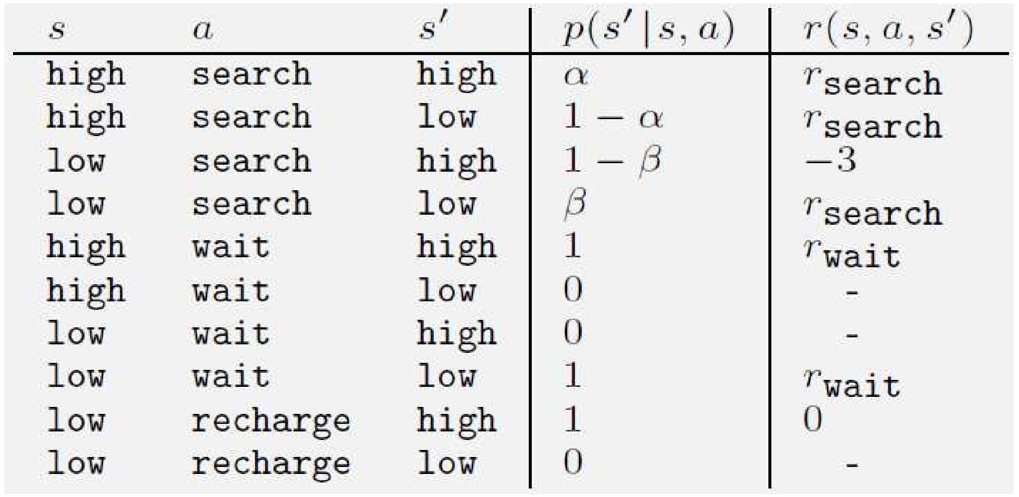
  – The probability of each possible value for s' and r depends only in the immediately preceding state s and action a

  – The state must include all relevant information about the past agent-environment interaction – Markov property

Example: Recycling Robot

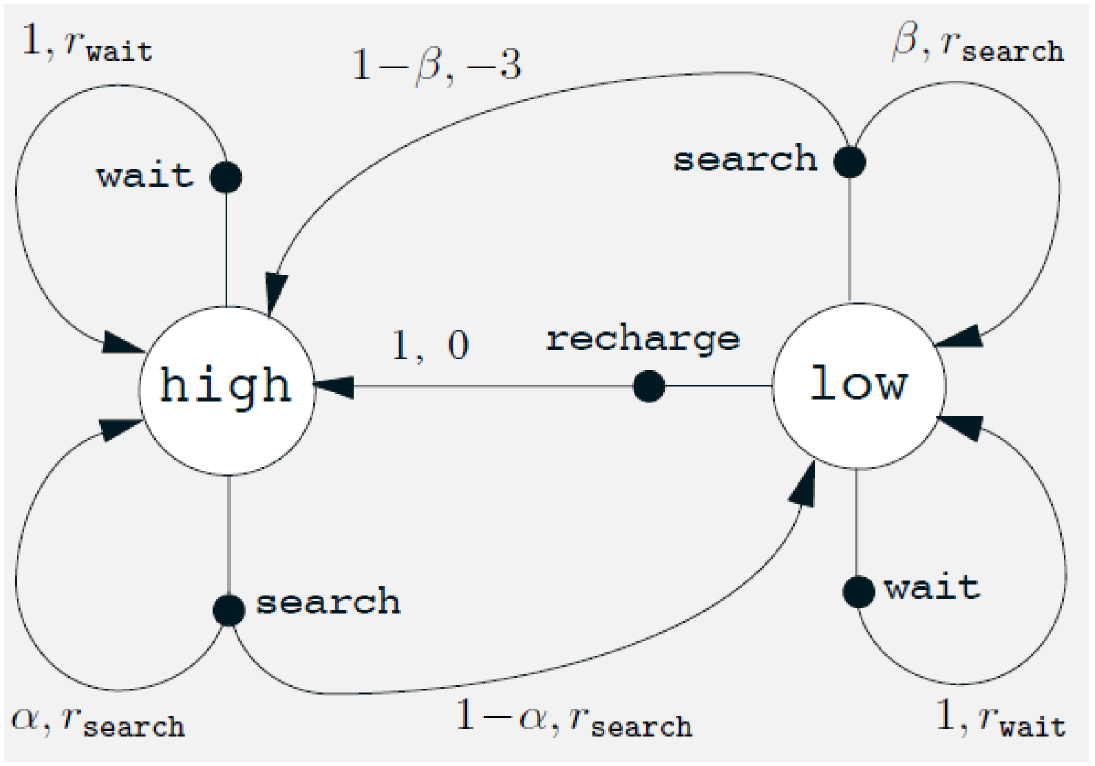
- A robot has to decide whether it should (1) actively search for a can, (2) wait for someone to bring it a can, or (3) go to home base and recharge
- Searching is better (higher probability of getting a can) but runs down battery; if out of battery, the robot has to be rescued
- Decisions made on the basis of current energy level: high, low
- Reward is zero except when getting a can, and negative if out of battery

$$\mathcal{S} = \{high, low\}$$
$$\mathcal{A}(high) = \{search, wait\}$$
$$\mathcal{A}(low) = \{search, wait, recharge\}$$
$$r_{search} > r_{wait}$$



s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	r_{search}
high	search	low	$1-\alpha$	r_{search}
low	search	high	$1-\beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	-

Example: Recycling Robot



Goals and Rewards

- A reward signal is used to define the goal of the agent
  - Learning to walk: reward proportional to the robot’s forward motion
  - Learning to Escape from a maze: reward -1 for any state prior to escape (encourage escaping as quickly as possible)
  - Learning to find empty cans for recycling: reward of 0 most of the time, +1 for each can collected
  - Learning to play checkers or chess: reward +1 for winning, -1 for losing, and 0 for drawing and nonterminal positions

- Provide rewards in such a way that by maximizing them the agent will also achieve our goal

→ The reward signal is a way of communicating to the robot *what* you want it to achieve, not *how*

Returns and Episodes

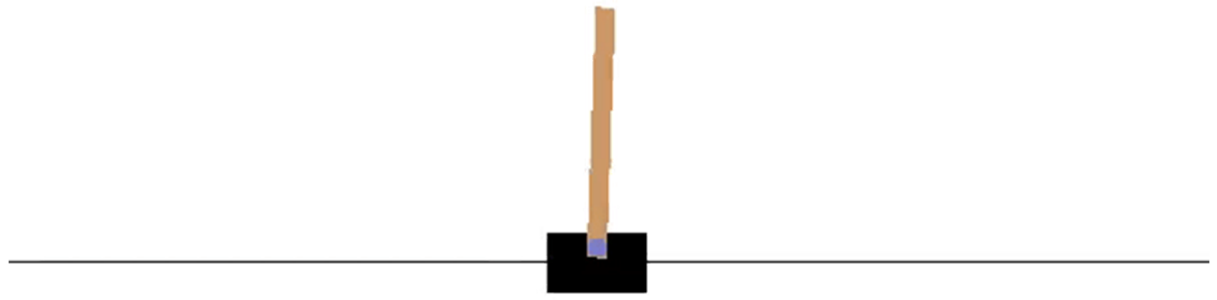
- Episodic tasks: when the agent-environment interaction breaks naturally into subsequences – episodes
  - From a starting state to a terminal state
  - Followed by a reset to another starting state, chosen independently of how the previous episode ended
  - Maximizing expected return: $G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$

- Continuing tasks do not break naturally into identifiable episodes (*e.g.* on-going process-control)

- Discounted return: $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$
  - $0 \le \gamma \le 1$ is the discount rate
    - If $\gamma = 0$ the agent is "myopic" (only immediate reward matters)
    - As γ approaches 1, the agent becomes more farsighted

Example: Pole Balancing

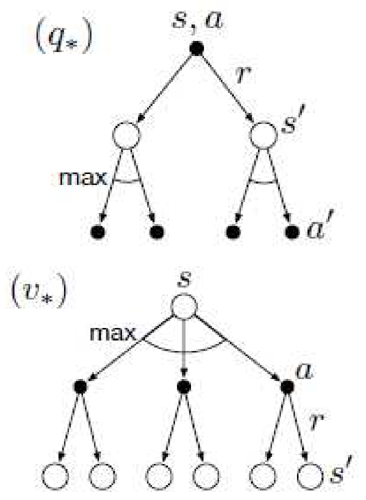
- Move a cart so as to keep a pole from falling over
  - Failure if the pole falls past a given angle or if the cart runs off the track
  - The pole is reset to vertical after each failure



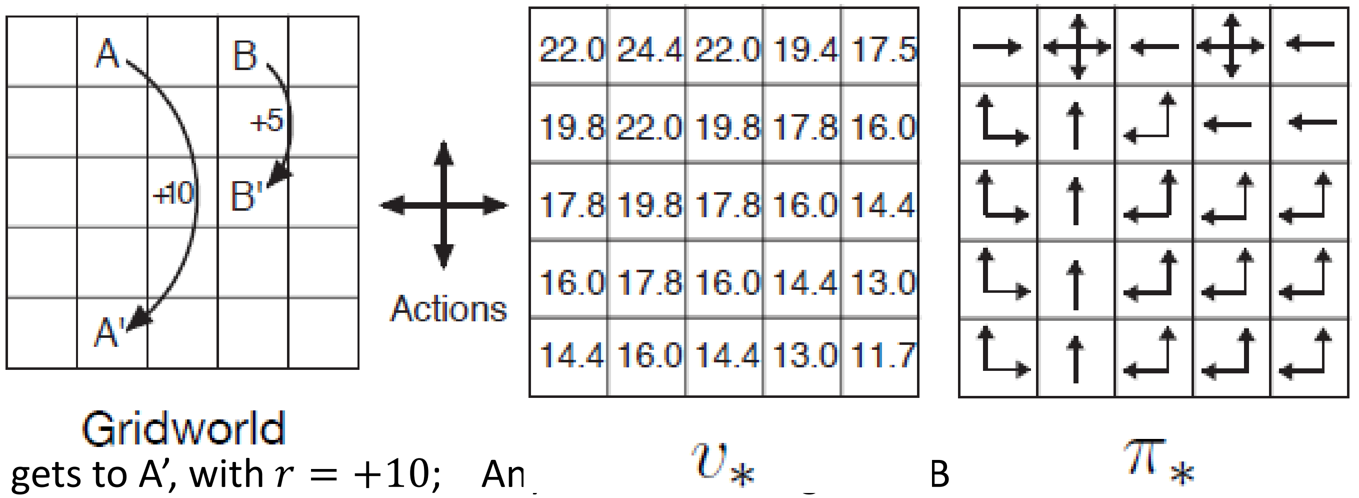
- Episodic task: reward +1 except when failure
  - return is the number of steps until failure
- Continuing task, using discounting: reward -1 on each failure and 0 otherwise
  - return is $-\gamma^k$

Policies and Value Functions

- Optimal policy π_*: expected return is greater than any other policy
  - Optimal state-value function: $v_*(s) \doteq \max_\pi v_\pi(s)$
  - Optimal action-value function: $q_*(s,a) \doteq \max_\pi q_\pi(s,a)$
- Once we know v_* or q_*, the optimal policy is greedy



- Example:



22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Any action from A gets to A’, with $r = +10$; An B

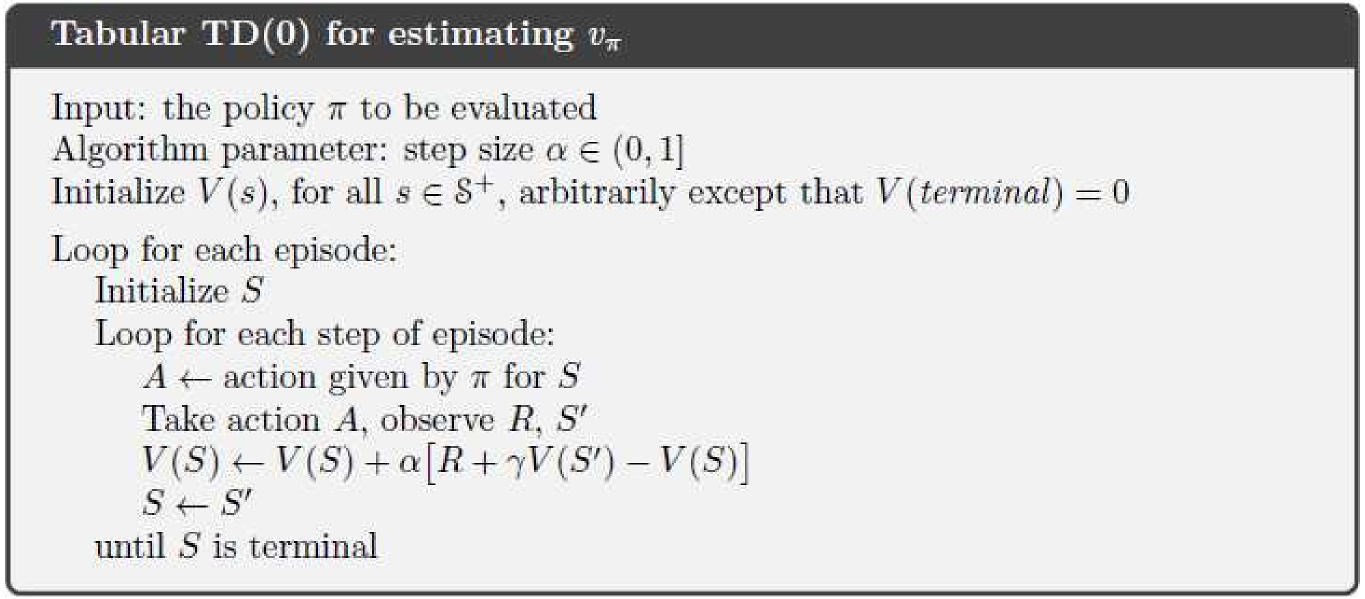
Approximation

- Optimal policies are computationally costly to find – we can only approximate
  - In tasks with small, finite state sets: tabular methods
  - Otherwise: function approximation using a more compact parameterized function representation (*e.g.* using neural networks)

→ The online nature of RL allows us to *put more effort into learning to make decisions for frequently encountered states*

Temporal-Difference Learning

- TD methods update estimates based on immediately observed reward and state
- Update rule: $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$
- Because it bases its update in part on an existing estimate, it is a *bootstrapping* method



Tabular TD(0) for estimating v_π

```
Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0,1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        $A \leftarrow$ action given by $\pi$ 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
```

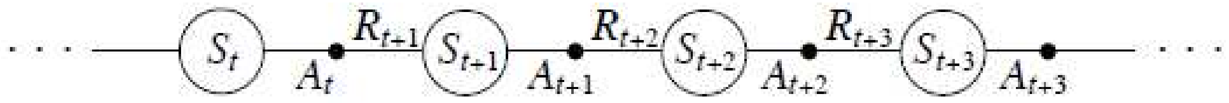
Temporal-Difference Learning

- TD vs Dynamic Programming methods
  - TD methods do not require a model of the environment's dynamics (rewards and next-state probability distributions)

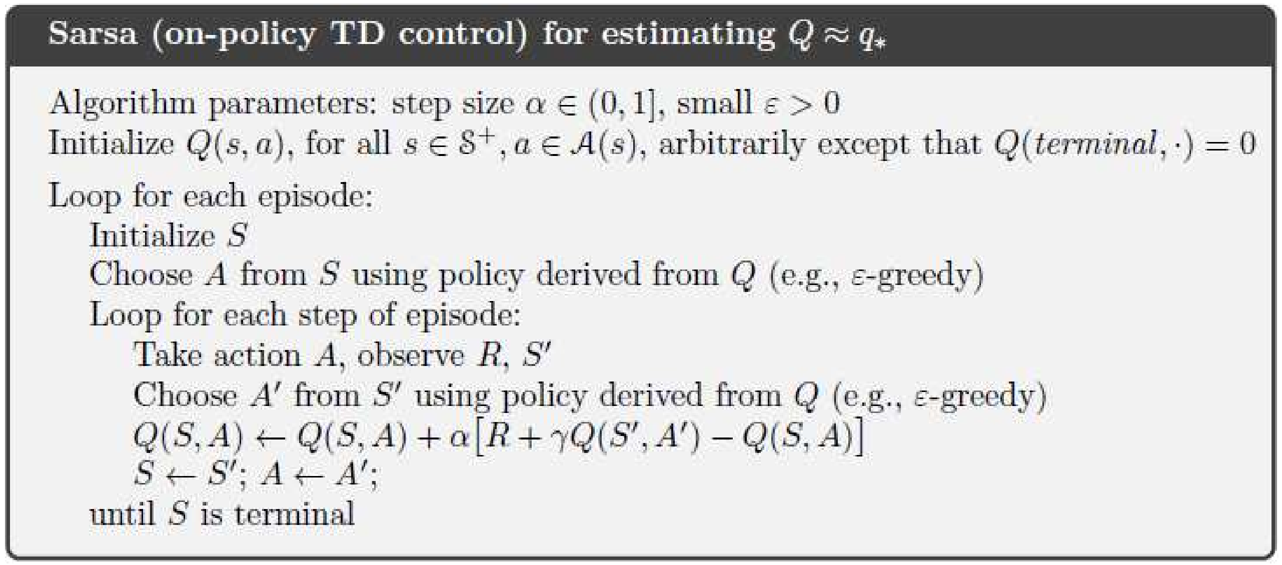
- TD vs Monte Carlo methods
  - TD methods are naturally implemented in an online, fully incremental fashion, while MC methods must wait until the end of an episode
    - Useful if episodes are very long, or in continuing tasks (that have no episodes at all)

- Usually, TD methods converge faster than MC methods on stochastic tasks

Sarsa: On-policy TD Control

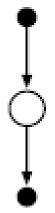


- Update rule: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
  - This rule uses every element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$



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 $Q(terminal, \cdot) = 0$

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

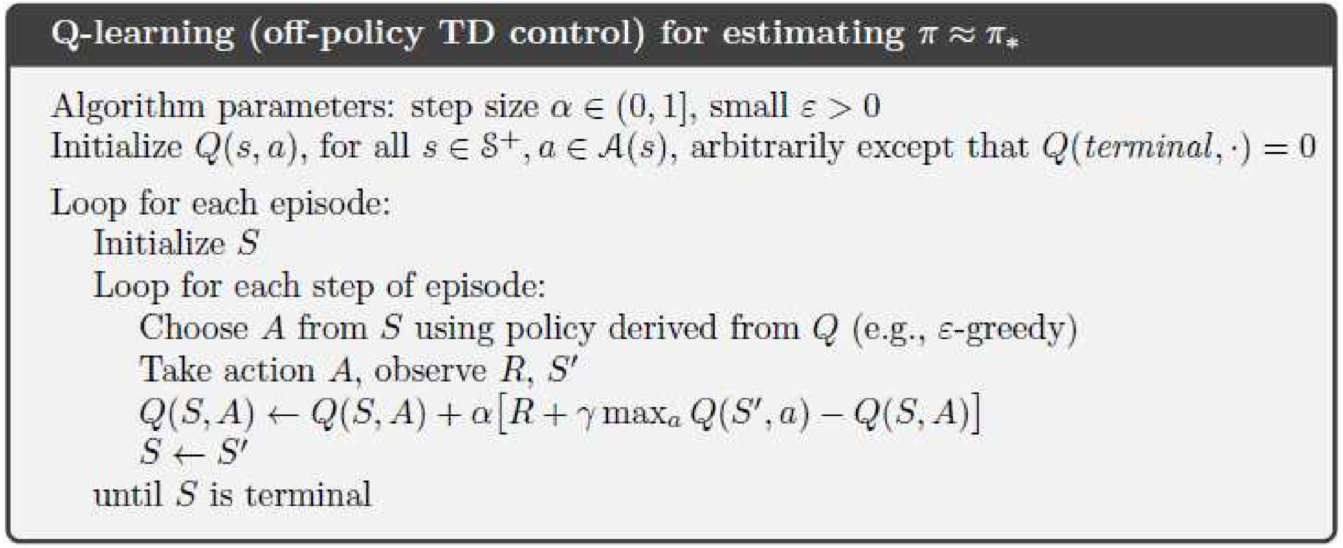

Sarsa

- Converges to optimal policy and action-value function if all state-action pairs are visited infinitely and policy converges to greedy (*e.g.* using ε-greedy with $\varepsilon = 1/t$)

Q-learning: Off-policy TD Control

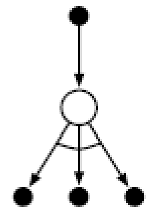
- Update rule:

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



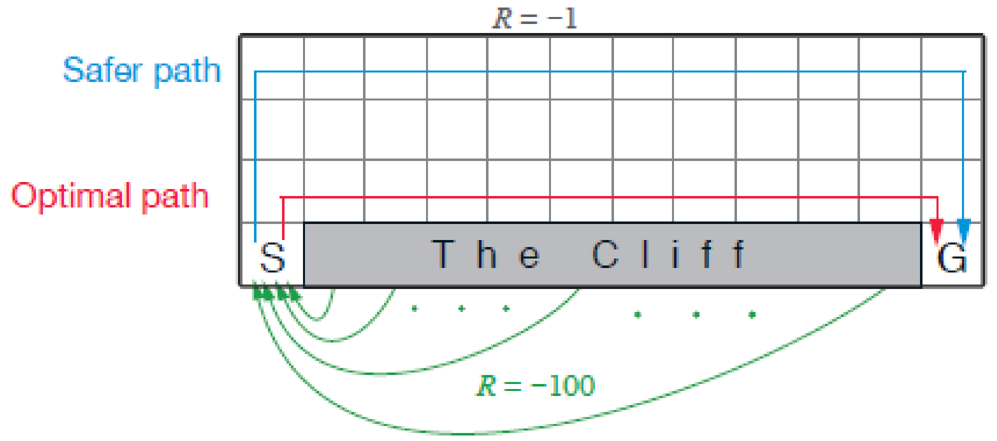
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

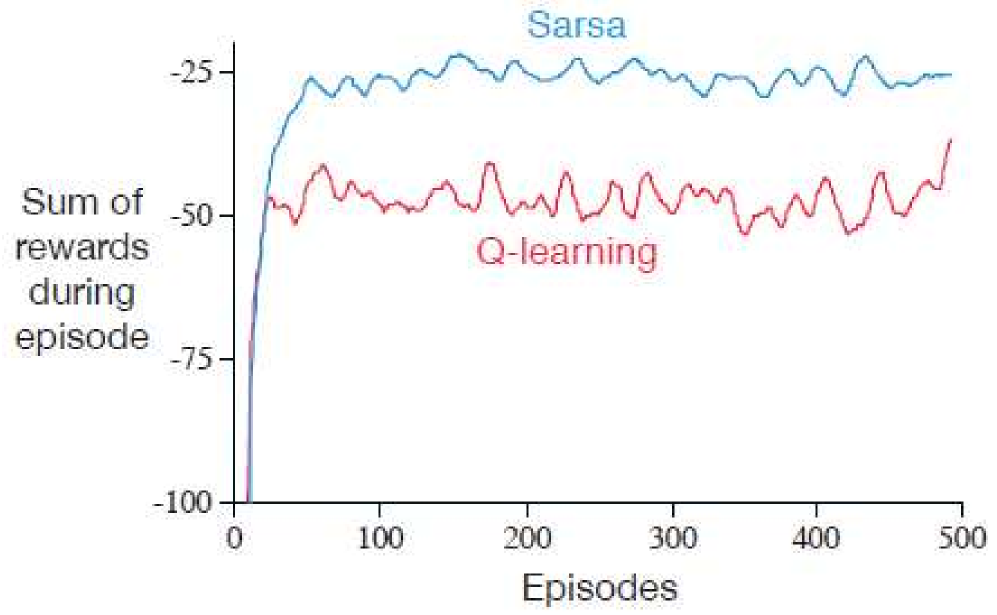
```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-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
```


Q-learning

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

Example: Cliff Walking



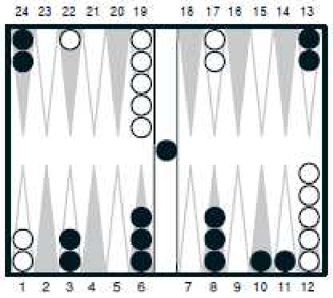


Sarsa and Q-learning with ε-greedy action selection ($\varepsilon = 0.1$)

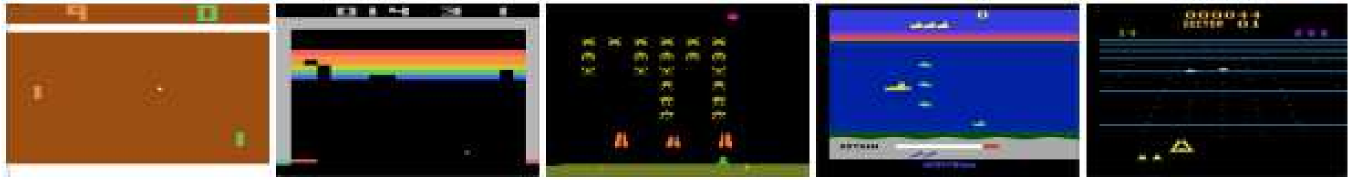
- Q-learning learns values for the optimal policy
- Sarsa takes action selection into account and learns the longer but safer path
- Given exploration, Q-learning occasionally falls off the cliff, hence the lower online performance

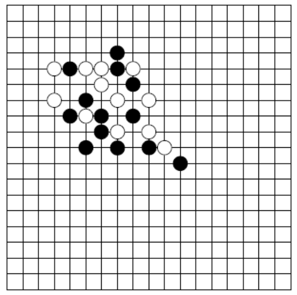
If ε is gradually reduced, both methods converge to the optimal policy

RL in Games

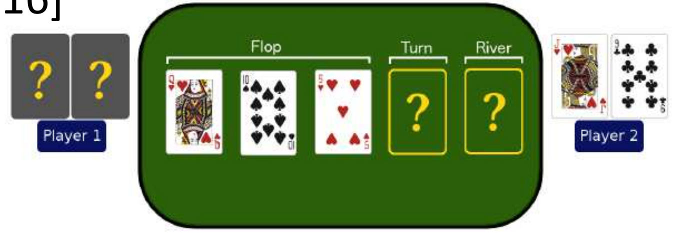


- TD-Gammon [Tesauro, 1995]
  - Neural Network trained with self-play reinforcement learning
- Atari 2600 Games [DeepMind, 2013]
  - Learn control policies directly from high-dimensional sensory input using reinforcement learning
  - Input is raw pixels and output is a value function estimating future rewards



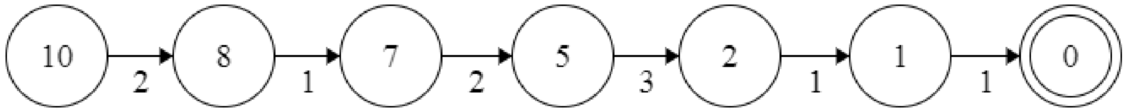


- AlphaGo [Google DeepMind, 2016]
  - Convolutional Neural Networks trained with human expert data
  - Deep Reinforcement Learning with fictitious self-play
- Poker: Heads-Up Limit Texas Hold'em – NFSP [UCL, 2016]
  - Deep Reinforcement Learning with fictitious self-play
  - No prior knowledge



The Toothpick Game

- 10 toothpicks
- 2 players take 1, 2 or 3 in turn
- Avoid last toothpick
  - $r(0) = +1 \qquad r(1) = -1 \qquad r(n) = 0, n > 1$
- Using Q-learning with $\alpha = 0.5$ and $\gamma = 0.9$, which values of $Q(s, a)$ change in each of the following episodes? (Consider only self-actions, shown in **bold**.)



$$\begin{tikzcd} 10 \arrow[r, "2"'] & 8 \arrow[r, "1"'] & 7 \arrow[r, "2"'] & 5 \arrow[r, "3"'] & 2 \arrow[r, "1"'] & 1 \arrow[r, "1"'] & 0 \end{tikzcd}$$

  - Actions: **2**-1-**2**-3-**1**-1 (States: **10**-8-**7**-5-**2**-1-**0**)
  - Actions: **2**-2-**1**-3-**1**-1 (States: **10**-8-**6**-5-**2**-1-**0**)
  - Actions: **1**-3-**1**-3-**1**-1 (States: **10**-9-**6**-5-**2**-1-**0**)
- After such updates, does the agent win when starting in state 10 and following a greedy policy, assuming the opponent always takes as much toothpicks as it can?

RL Algorithms

Algorithm	Description	Model	Policy	Action Space	State Space	Operator
Monte Carlo	Every visit to Monte Carlo	Model-Free	Off-policy	Discrete	Discrete	Sample-means
Q-learning	State–action–reward–state	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA	State–action–reward–state–action	Model-Free	On-policy	Discrete	Discrete	Q-value
Q-learning - Lambda	State–action–reward–state with eligibility traces	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA - Lambda	State–action–reward–state–action with eligibility traces	Model-Free	On-policy	Discrete	Discrete	Q-value
DQN	Deep Q Network	Model-Free	Off-policy	Discrete	Continuous	Q-value
DDPG	Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
A3C	Asynchronous Advantage Actor-Critic Algorithm	Model-Free	On-policy	Continuous	Continuous	Advantage
NAF	Q-Learning with Normalized Advantage Functions	Model-Free	Off-policy	Continuous	Continuous	Advantage
TRPO	Trust Region Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
PPO	Proximal Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
TD3	Twin Delayed Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
SAC	Soft Actor-Critic	Model-Free	Off-policy	Continuous	Continuous	Advantage

Further Reading

- Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning – An Introduction*, 2nd ed., The MIT Press: Chap. 1-3, 6
- Simple Tutorial Videos for Deep RL:

  Introduction on Reinforcement Learning and Deep RL:

  Https://www.youtube.com/watch?v=JgvyzIkgxF0

  PPO – Proximal Policy Optimization (PPO):

  https://www.youtube.com/watch?v=5P7I-xPq8u8

Conclusions

- RL enables to learn intelligent behavior in complex environments
- Large number of algorithms and approaches
- Amazing results in vintage Atari Games
- Stunning results of AlphaGo and AlphaZero
- Very promising results in Robotics
- Very fast evolution in the last few years

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Artificial Intelligence

Lecture 12: Intelligent Agents – Reinforcement Learning

(slides baseados em "Cardoso, H.L., 2018" and Choudhary, 2019)

Luís Paulo Reis, Henrique L. Cardoso

lpreis@fe.up.pt, hlc@fe.up.pt