

# CES -161 - Modelos Probabilísticos em Grafos

## Reinforcement Learning based on Markov Decision Process

Prof. Paulo André Castro

[pauloac@ita.br](mailto:pauloac@ita.br)

[www.comp.ita.br/~pauloac](http://www.comp.ita.br/~pauloac)

Sala 110, IEC-ITA

# Reinforcement Learning in a nutshell

---

- Imagine playing a new game whose rules you don't know; after a hundred or so moves, your opponent announces, "You lose."
- This is basically reinforcement learning
  - But, the rules does not change
  - and you can see the state that you are in.. (environment is observable!)..
- Reinforcement learning might be considered to encompass all of AI: an agent is placed in an environment and must learn to behave (act) successfully in it

# Markov Decision Process and Reinforcement Learning

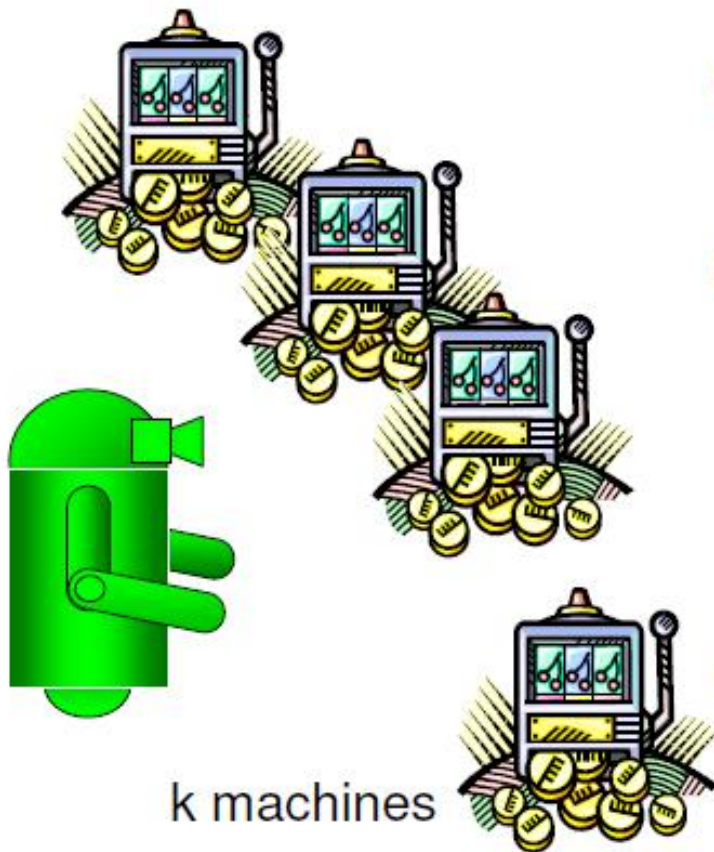
---

- When we talked about MDP,  $\langle S, A, p, r \rangle$ , we assumed that we knew the agent's reinforcement function,  $r$ , and a model of how the world works, expressed as the transition probability function,  $p$ .

In *Reinforcement Learning* (RL), we would like an agent to **learn** to behave well in an MDP world, but **without knowing** anything about  $r$  or  $p$  when it starts out

# New example: Armed bandit (Caça-níqueis)

---



- Every time you pull an arm on a machine, it either pays off a dollar or nothing.
  - Assume that each machine has a **hidden probability of paying off**, and that whenever you pull an arm, the outcome is independent of previous outcomes and is determined by the hidden payoff probability
- *What should you do to make as much money as possible during a given time?*

# O caça níquel de $n$ -alavancas

---

O agente pode, a cada instante de tempo, escolher uma entre  $n$  ações possíveis. Após a escolha, uma recompensa obtida a partir de uma dist. de probabilidade estacionária, que depende só da ação escolhida, é produzida. O objetivo do agente é maximizar a recompensa total obtida sobre um dado período de tempo.

Cada ação tem uma recompensa esperada (média) associada. Esta é a utilidade da ação.

Se conheço a utilidade de cada ação, o problema é trivial. Se não conheço, posso pelo menos manter uma estimativa, e escolher a ação com o maior utilidade estimada. Esta é a ação *greedy* (ou avara).

- ◇ Exploração: escolha da ação *greedy*. Explorar é produzir as ações que são julgadas as melhores com o fim de resolver o problema.
- ◇ Exploração: escolha de ações não-*greedy*. Explorar é produzir ações alternativas, com o intuito de mapear características desconhecidas do problema.



# Opções....

---

- Agir aleatoriamente?
- Mudar de máquina a cada vez que perder.. ?
  - Melhor que aleatório, mas não é ótimo
- Estimar o retorno de cada máquina através de contagem e depois permanece na melhor?
  - Como estimar?

# Estimation: Frequentism

---

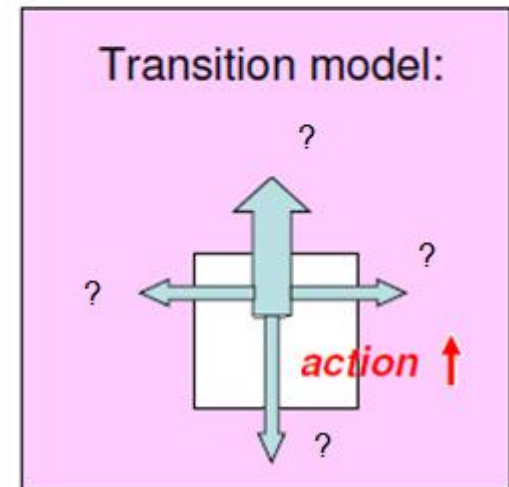
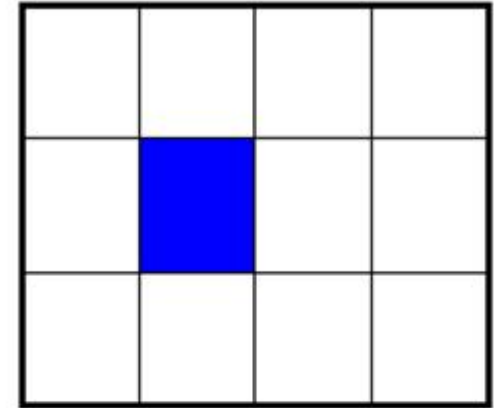
- Average Reward: if I choose an action  $a$ ,  $K_a$  times and receive the rewards  $r_1, r_2, \dots, r_{k_a}$  then I can estimate the reward

$$V_t(a) = \frac{r_1 + r_2 + \dots + r_{k_a}}{k_a}$$

- It is possible to prove that as  $k$  grows it converges to the expected value of reward.
- How to define the actions?

# Example: Revisting the 4x3 World

- Let's think about parameter estimation in the 4x3 world without  $p$  and  $r \Rightarrow$
- But without rewards...
- What is the objective??
- To keep things simple, let's start with a fixed policy  $\pi$ . In state  $s$ , it always executes the action  $\pi(s)$
- The goal is simply to learn how good the policy is— i.e. - the utility function  $U \pi (s)$
- How ???





# Estimating $p$ and $r$ and then solving it

---

**function** PASSIVE-ADP-AGENT(*percept*) **returns** an action

**inputs:** *percept*, a percept indicating the current state  $s'$  and reward signal  $r'$

**persistent:**  $\pi$ , a fixed policy

*mdp*, an MDP with model  $P$ , rewards  $R$ , discount  $\gamma$

$U$ , a table of utilities, initially empty

$N_{sa}$ , a table of frequencies for state–action pairs, initially zero

$N_{s'|sa}$ , a table of outcome frequencies given state–action pairs, initially zero

$s, a$ , the previous state and action, initially null

**if**  $s'$  is new **then**  $U[s'] \leftarrow r'$ ;  $R[s'] \leftarrow r'$

**if**  $s$  is not null **then**

increment  $N_{sa}[s, a]$  and  $N_{s'|sa}[s', s, a]$

**for each**  $t$  such that  $N_{s'|sa}[t, s, a]$  is nonzero **do**

$P(t | s, a) \leftarrow N_{s'|sa}[t, s, a] / N_{sa}[s, a]$

$U \leftarrow$  POLICY-EVALUATION( $\pi, U, mdp$ )

**if**  $s'$ .TERMINAL? **then**  $s, a \leftarrow$  null **else**  $s, a \leftarrow s', \pi[s']$

**return**  $a$

**Algorithm for a passive(fixed policy) reinforcement learning agent**

The POLICY-EVALUATION function calculates  $U$  for a fixed policy for the estimated MDP using Policy iteration algorithm\*\*

# Policy Iteration Algorithm

---

- As we have seen in chapter 5, policy iteration algorithm has two steps:
  - **Policy evaluation:** given a policy  $\pi_i$ , calculate the utility of each state -  $U_i(s)$  - if  $\pi_i$  were to be executed.
  - **Policy improvement:** Calculate a new MEU policy  $\pi_{i+1}$ , using one-step look-ahead based on  $U_i(s)$  as in:

$$\pi^*(s) = \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')$$

- In fact, the first step is basically the value iteration algorithm, but it is possible to simplify it
  - because, it is not necessary to do **exact** policy evaluation.
- Instead, we can perform some number of simplified value iteration steps (simplified because the policy is fixed) to give a reasonably good approximation of the utilities, we will call this **modified policy iteration**

# Modified policy iteration

---

- We are going to iterate the utility vector in versions using this simplified version of Bellman equations:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' | s, \pi_i(s)) U_i(s')$$

- The resulting algorithm (*modified policy iteration*) is often much more efficient than standard policy iteration or even value iteration function
  - We can also stop early if there are no changes in the new policy

# Modified policy iteration algorithm

---

**function** POLICY-ITERATION( $mdp$ ) **returns** a policy

**inputs:**  $mdp$ , an MDP with states  $S$ , actions  $A(s)$ , transition model  $P(s' | s, a)$

**local variables:**  $U$ , a vector of utilities for states in  $S$ , initially zero

$\pi$ , a policy vector indexed by state, initially random

**repeat**

$U \leftarrow$  POLICY-EVALUATION( $\pi, U, mdp$ )

$unchanged? \leftarrow$  true

**for each** state  $s$  **in**  $S$  **do**

**if**  $\max_{a \in A(s)} \sum_{s'} P(s' | s, a) U[s'] > \sum_{s'} P(s' | s, \pi[s]) U[s']$  **then do**

$\pi[s] \leftarrow$  argmax  $\sum_{s'} P(s' | s, a) U[s']$

$unchanged? \leftarrow$  false

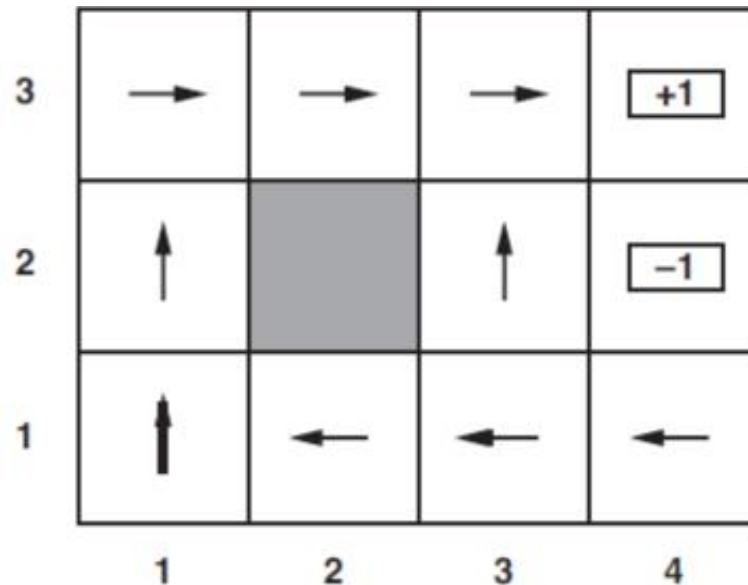
**until**  $unchanged?$

**return**  $\pi$

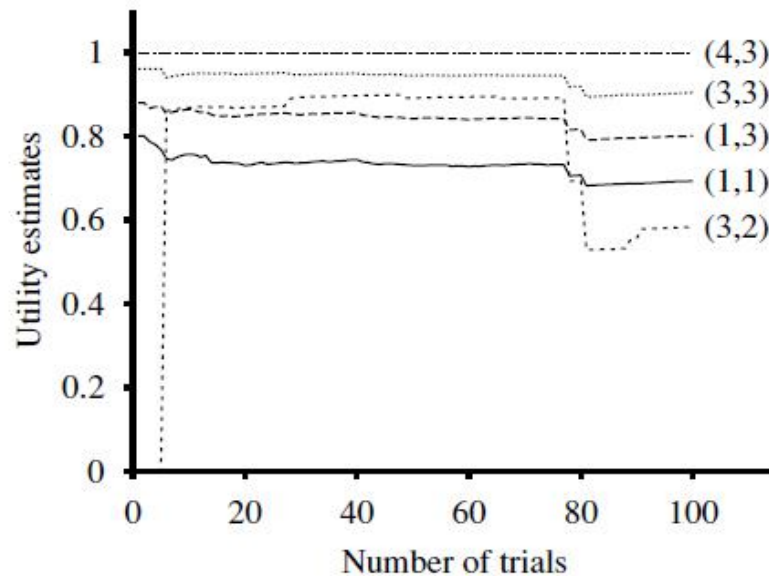
# Let's try this algorithm to 4x3 world

---

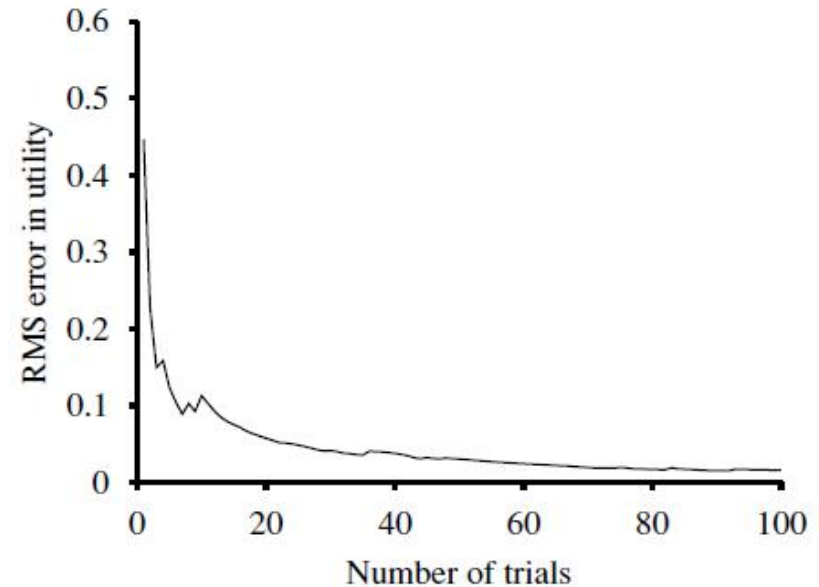
- We need a policy... Let's use that one



## Results - The passive ADP learning curves for the 4x3 world, given the optimal policy



(a)



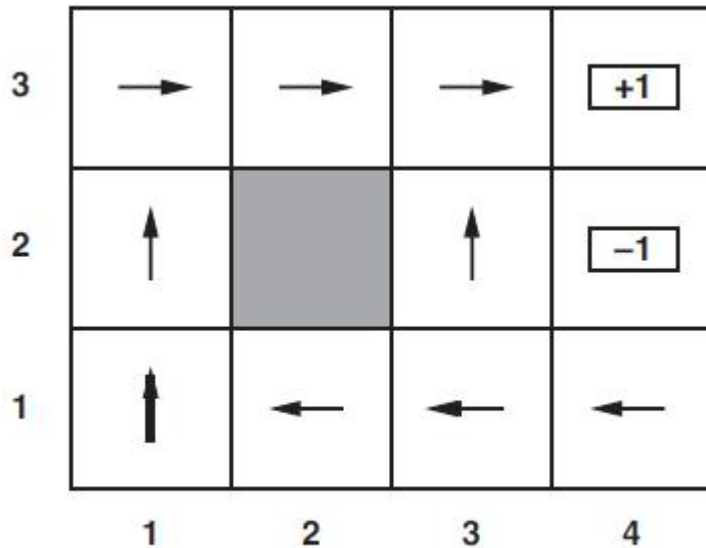
(b)

- (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice the large changes occurring around the 78th trial—this is the first time that the agent falls into the -1 terminal state at (4,2)
- (b) The root-mean-square error in the estimate for  $U(1, 1)$ , averaged over 20 runs of 100 trials each
- Observe the convergence of the Utility Estimates for all values

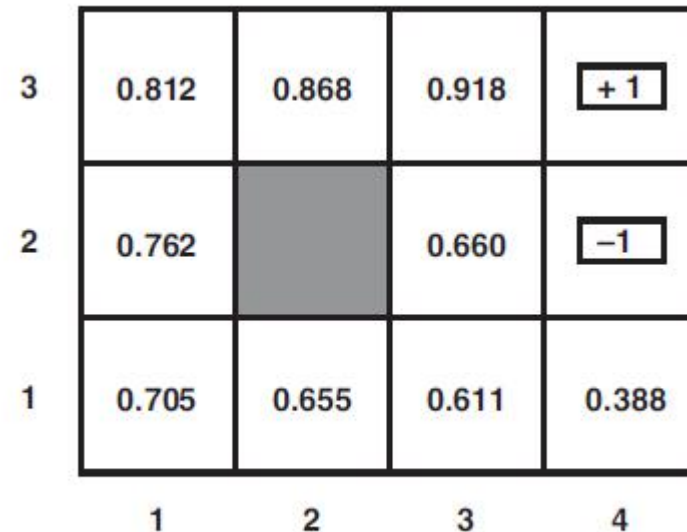


# Results - 2

- The system converges and we find the correct Utility values!!



(a)

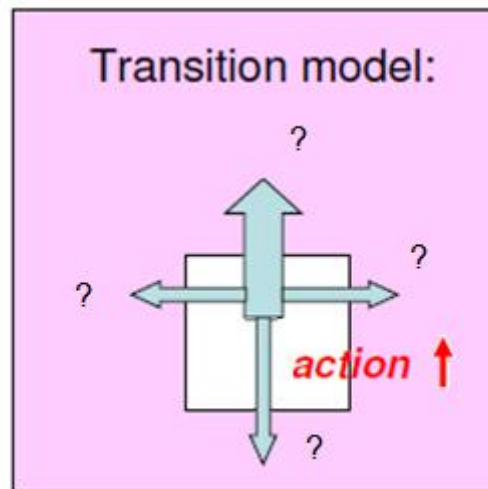
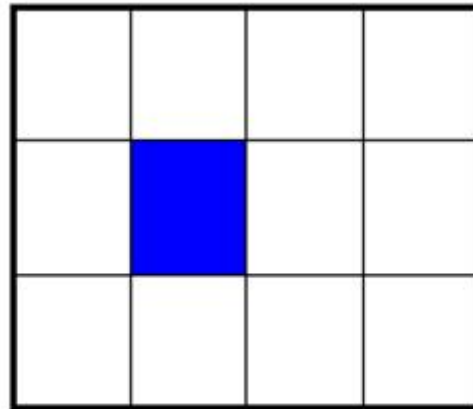


(b)

Next step: Find the optimal policy for the same problem....

---

How??



# Two options: Learn a model $(p,r)$ or solve directly

---

- What do you do when you don't know how the world works???
- One Option (A): Learn the world
  - Estimate  $r$  (reinforcement function) and  $p$  (transition probability function) from observed data
    - Parameter Estimation
  - Solve the problems using  $\langle S, A, p, r \rangle$  using known techniques
    - Value iteration
    - Policy iteration

# Estimate value function directly

---

- Another Option (B):
  - Estimate a value function directly:
    - This approach is sound and in a lot of cases, it's probably the best thing to do
    - But it's possible to find the optimal value function without ever estimating the state transition (function  $p$ ) ??? We will try an algorithm for that later...
    - Let's check the first approach first: parameter estimation...

# Parameter Estimation

---

1. Parameter estimation: by counting the number of times various events occur and taking ratios.
  - ◆ You can estimate the next-state distribution  $\mathbf{p}(s'|s, \mathbf{a})$  by counting the number of times the agent has taken action  $\mathbf{a}$  in state  $\mathbf{s}$  and looking at the proportion of the time that  $\mathbf{s}'$  has been the next state.
  - ◆ Similarly, you can estimate  $\mathbf{r}(\mathbf{s})$  just by averaging all the reinforcements you've received when you were in state  $\mathbf{s}$ .
2. Solve for optimal policy given estimated  $\mathbf{r}$  and  $\mathbf{p}$

# Problems in Parameter estimation

---

- **How long should you gather data to estimate the model, before it is good enough to use to find a policy?**
  - One Option: Re-estimate the model on every step
    - Each time you have a new piece of evidence, you update the model and run the policy iteration on the updated model!!! Problem?
      - It may be TOO expensive computationally
  - **Another option: Estimate the model after some perceptions and Run value iteration on the updated model, but at a pace that you can afford computationally**
  - **Yet Another option: Temporal difference ( we will talk about that later)**



# Problems in Parameter estimation - 2

---

- Run value iteration on the updated model, but at a pace that you can afford computationally
- We estimate the parameters (or model  $p$  and  $r$ ), we can use the algorithm passive-ADP for that

```
function PASSIVE-ADP-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state  $s'$  and reward signal  $r'$ 
  persistent:  $\pi$ , a fixed policy
              $mdp$ , an MDP with model  $P$ , rewards  $R$ , discount  $\gamma$ 
              $U$ , a table of utilities, initially empty
              $N_{sa}$ , a table of frequencies for state-action pairs, initially zero
              $N_{s'|sa}$ , a table of outcome frequencies given state-action pairs, initially zero
              $s$ ,  $a$ , the previous state and action, initially null

  if  $s'$  is new then  $U[s'] \leftarrow r'$ ;  $R[s'] \leftarrow r'$ 
  if  $s$  is not null then
    increment  $N_{sa}[s, a]$  and  $N_{s'|sa}[s', s, a]$ 
    for each  $t$  such that  $N_{s'|sa}[t, s, a]$  is nonzero do
       $P(t | s, a) \leftarrow N_{s'|sa}[t, s, a] / N_{sa}[s, a]$ 
     $U \leftarrow$  POLICY-EVALUATION( $\pi, U, mdp$ )
  if  $s'.\text{TERMINAL?}$  then  $s, a \leftarrow$  null else  $s, a \leftarrow s', \pi[s']$ 
  return  $a$ 
```

- Execute Value iteration on the learned model
- There is another problem, we always need some specific policy in the learning process. Let's talk about that (passive vs active learning)!

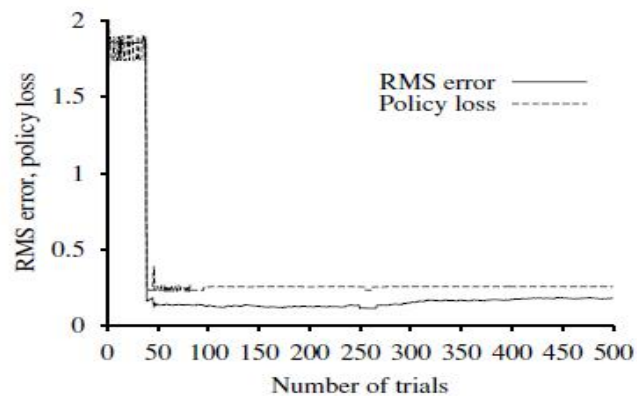
# Passive and active learning

---

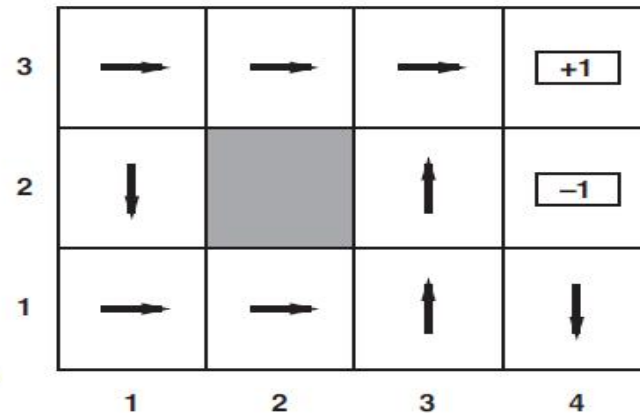
- Passive learning agent: it follows a given policy, while learning
- Active agent: it decides what to do during the learning process
- **We need a complete policy while learning, The initial policy may change what is learned?**
  - For instance, If we follow the recommendation of the optimal policy for the learned model at each step, are we always going to find the **true optimal policy**?

# Active learning

- The figure below shows the results of one sequence of trials for an greedy active agent that follows the recommendation of the optimal policy for the learned model at each step.



(a)

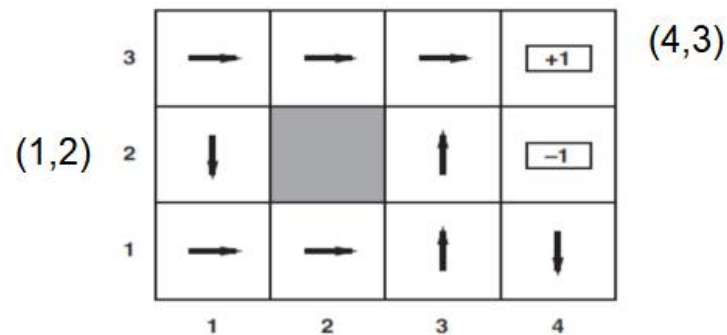


(b)

- (a) Root mean square (RMS) error on the mean utilities of the non-terminal states
- (b) Suboptimal Policy that the agent converges for
- The agent **does not** learn the true utilities or the true optimal policy!
- In fact, repeated experiments show that the greedy agent **very seldom** converges to the optimal policy

# Problem!!

- The agent does not learn the true utilities or the true optimal policy!
- What happens instead is that, in the 39th trial, it finds a policy that reaches the +1 reward along the lower route via (2,1), (3,1), (3,2), and (3,3)



- After experimenting with minor variations, from the 276th trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2), (1,3), and (2,3)
- How can it be that choosing the optimal action leads to suboptimal results?

# Problem - 2

---

- How can it be that choosing the optimal action leads to suboptimal results?
- The answer is that the learned model is not the same as the true environment; what is optimal in the learned model can therefore be suboptimal in the true environment
- Unfortunately, the agent does not know what the true environment (*Reinforcement learning problem!*) is , so it cannot compute the optimal action for the true environment

# Problem - 3

---

- How can we solve the problem of learning a wrong model??
- The point is that actions do more than provide rewards according to the current learned model; they also contribute to learning the true model by affecting the percepts that are received.
- By improving the model, the agent will receive greater rewards in the future.
  - In other words, it is good a idea to really know the world, even when we think we know something about it



# Exploitation (Exploração) vs Exploration (Exploração)

---

- ◆ There are two, possibly opposing reasons for the agent to choose an action:
  1. because it thinks the action will have a good result in the world (*exploitation*), or
  2. because it thinks the action will give it more information about how the world works (*exploration*).

# Exploitation vs Exploration

---

- An agent therefore must make a **tradeoff** between exploitation — maximize its reward, as reflected in its current utility estimates — and exploration to know better the world and maximize its long-term well-being
- **Pure exploitation** risks getting stuck in a known path, but with suboptimal utility
- **Pure exploration** to improve knowledge is of **no use** if you never puts that knowledge into practice

# How to pick actions in active learning?

---

- Always Greedy (Exploitation) = Maximize expected utility
  - The problem is that you may despise good actions, because they seem bad in short term, but that can be very good in the long run
- Random (exploration): select actions randomly
- Trade-off solution: Greedy policy but with a probability  $\epsilon$  of choosing a random action. It is usually called  $\epsilon$ -greedy

# Strategies

---

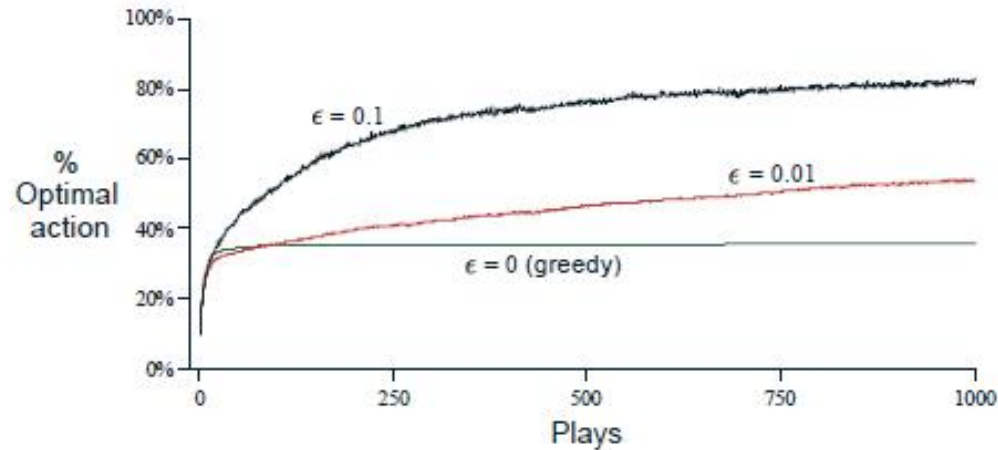
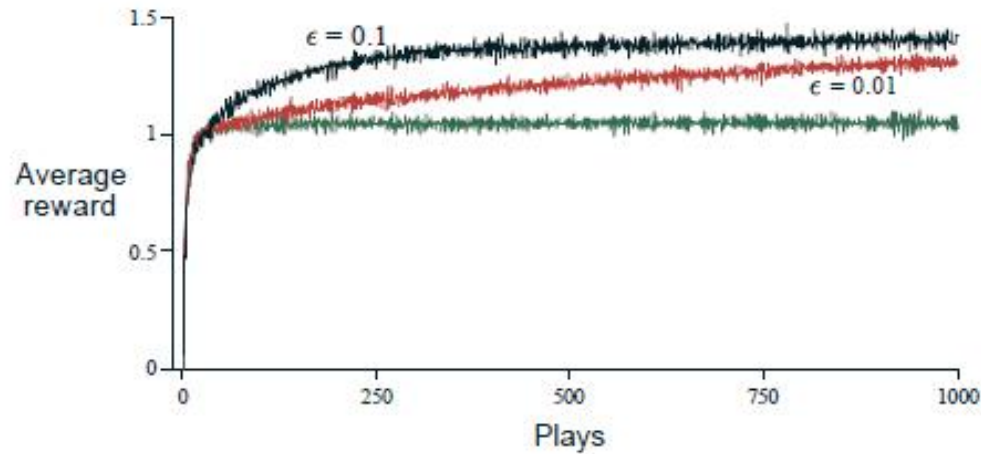
- Ultimately, the best strategies spend
  - ◆ some time **exploring**: trying all the arms to see what their probabilities are like, and
  - ◆ some time **exploiting**: doing the apparently best action to try to get reward.

In general, the longer you expect to live, the more time you should devote to exploration.

→  $\epsilon$ -greedy strategy  $\left\{ \begin{array}{l} \epsilon: \text{Exploration} \\ 1 - \epsilon: \text{Exploitation} \end{array} \right.$

# Alguns Resultados com $\epsilon$ -greedy

---



# $\epsilon$ -greedy

---

- It is possible to realize (and prove) that a  $\epsilon$  -greedy produces convergence to the optimal policy
  - It happens because  $\epsilon$  -greedy avoids to be stuck in a rut (routine) by picks another action time to time
- However, it can also bring other problem..What problem?
  - If you keep choosing with  $\epsilon$  -greedy, you will never act according to the optimal policy
  - But, if you stop after finding the optimal policy, how can you be sure that you learned the true model and its optimal policy and not other model..

# Greedy in the limit of infinite exploration

---

- A learning policy must try each action in each state an unbounded number of times to avoid having a probability that an optimal action is missed because of an unusually bad series of outcomes.
  - An passive agent using such a policy will eventually learn the true environment model
  - In other words, the policy should be greedy in the limit of infinite exploration (GLIE)
- So the agent's actions become optimal with respect to the learned (and hence the true) model.
- There are several GLIE schemes; one of the simplest is to have the agent choose a random action a fraction  $1/t$  of the time and to follow the greedy policy otherwise
  - This does eventually converge to an optimal policy, but it can be **extremely slow**. How could we do better??

# Trying something better...

---

- A more sensible approach would **give some weight to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility.** We can describe that as a function  $f$  to choose between actions
- Let us use  $U^+(s)$  to denote the optimistic estimate of the utility of the state  $s$ , and let  $N(s, a)$  be the number of times action  $a$  has been tried in state  $s$ . We may define a new maximization

$$U^+(s) \leftarrow R(s) + \gamma \max_a f \left( \sum_{s'} P(s' | s, a) U^+(s'), N(s, a) \right)$$

- Function  $f(u, n)$  determines how greed (preference for high values of  $u$ ) is traded off against curiosity (preference for actions that have not been tried often and have low  $n$ ). It is called a **exploration function**



# Exploration functions

---

- Function  $f(u, n)$  determines how greed (preference for high values of  $u$ ) is traded off against curiosity (preference for actions that have not been tried often and have low  $n$ ). Many functions could be defined, for instance:

$$f(u, n) = \begin{cases} R^+ & \text{if } n < N \\ u & \text{otherwise} \end{cases}$$

- $R^+$  is a very high value compared to the actual possible returns and  $N$  is a fixed number.
  - This will have the effect of making the agent try each action-state pair at least  $N$  times

# Learn the world

---

- Using that we have a implementable  **$\epsilon$ -greedy** that we can use with the algorithms that we have seen before(option A: Learn the world (p and r) and solve it!
- The problem is that it can be extremely slow!!!!
  - We are going to learn world (slow!!)
  - Learn the optimal policy (slow for high number of states)

## Other option (B): Estimate value function directly

---

- Let's try the alternative of estimating a value function directly without explicitly estimating  $p$  and  $r$ 
  - This approach is sound, and in a lot of cases, it's probably the right thing to do
  - But it's possible to find the optimal value function without ever estimating the state transition probabilities directly ?

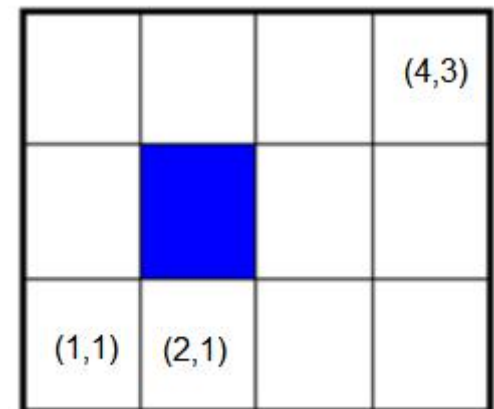
# Temporal difference

- Solving the underlying MDP as seen before is not the only way to use the Bellman equations to bear on the learning problem.
- Another way is to use the observed transitions to adjust the utilities of the observed states so that they agree with the constraint equations.
- Suppose that, as a result of the first trial, the utility estimates are  $U^\pi(1, 3)=0.84$  and  $U^\pi(2, 3)=0.92$ . Now, if this transition occurred all the time, we would expect the utilities to obey the equation:

$$U^\pi(1,3) = -0.04 + U^\pi(2,3)$$

- More generally, when a transition occurs from
- state  $s$  to state  $s'$ , we may apply the
- following update to  $U^\pi(s)$ :

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s))$$



- where,  $\alpha$  is a learning rate parameter

# Temporal difference - 2

---

- Because this update rule uses the difference in utilities between successive states ( $s$  and  $s'$ ), it is often called the temporal-difference (TD) equation.
- This update equation causes the agent to reach the same equilibrium given by :

$$U^\pi(s) = R(s) + \gamma \sum_{s'} P(s' | s, \pi(s)) U^\pi(s')$$

- But...notice that the TD update involves only the observed successor  $s'$  while the actual equilibrium conditions involve all possible next states.

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha (R(s) + \gamma U^\pi(s') - U^\pi(s))$$

- Could it cause an improperly large change in  $U^\pi(s)$  when a very rare transition occurs??

# Temporal difference - 3

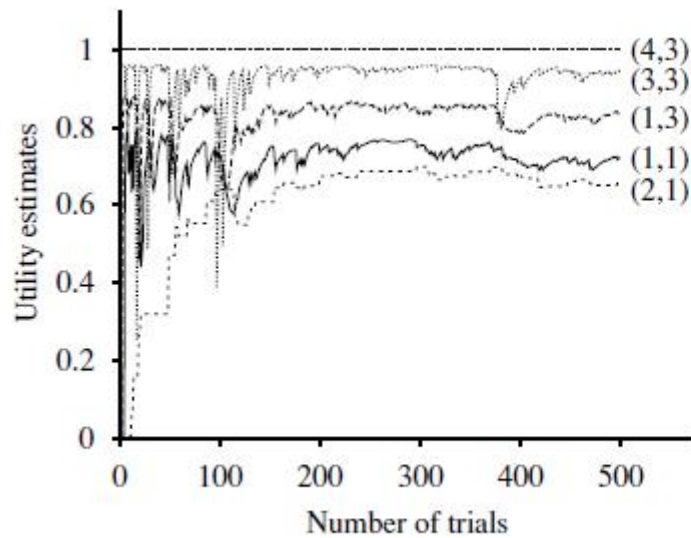
---

- You might think that: Yes! this causes an improperly large change in  $U^\pi(s)$  when a very rare transition occurs;
  - but, in fact, because rare transitions occur only rarely, the average value of  $U^\pi(s)$  will converge to the correct value.
- Furthermore, if we change  $\alpha$  from a fixed parameter to a function that decreases as the number of times a state has been visited increases, then  $U^\pi(s)$  itself will converge to the correct value, given certain conditions:
  - If  $\alpha$  decays as  $O(1/t)$  where  $t$  is the iteration number, then the rule can be shown to converge to the correct value. In fact, it is required that:  $O(1/t)$  satisfies these conditions.

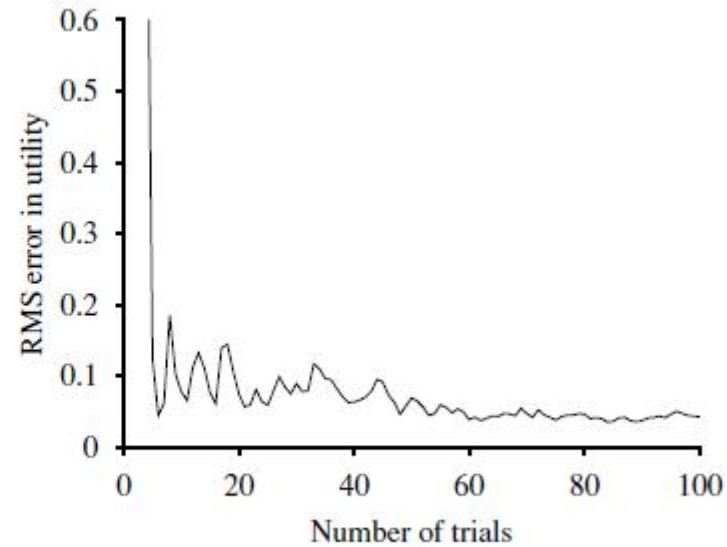
$$\sum_{t=1}^{\infty} \alpha(t) = \infty \text{ and } \sum_{t=1}^{\infty} \alpha^2(t) < \infty.$$

# TD agent applied to the 4x3 world

---



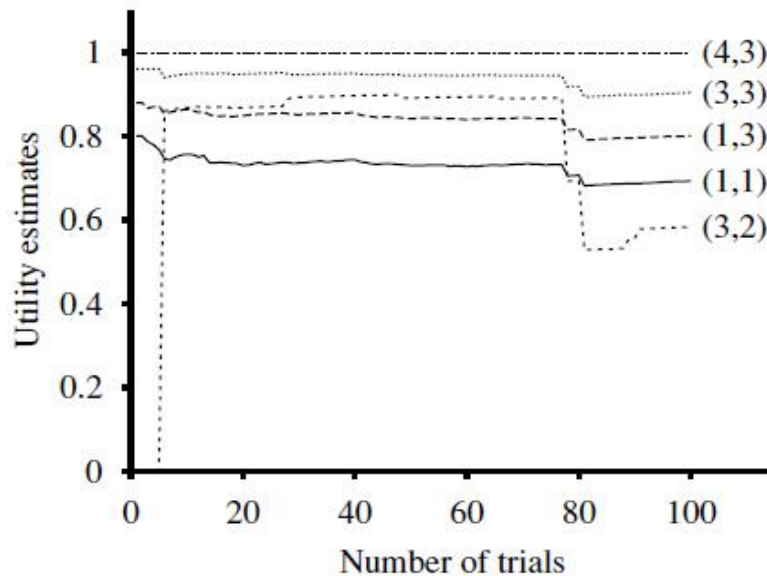
(a)



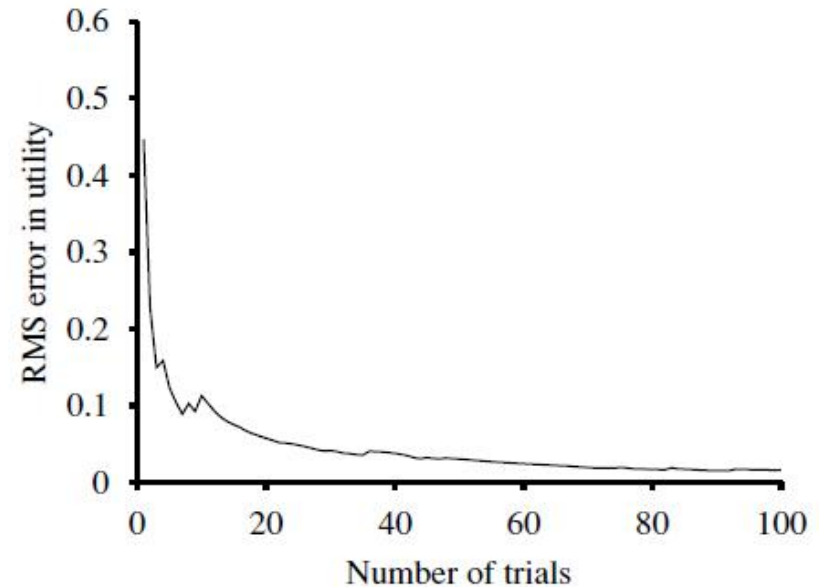
(b)

- (a) The utility estimates for a selected subset of states, as a function of the number of trials
- b) The root-mean-square error in the estimate for  $U(1, 1)$ , averaged over 20 runs of 500 trials each. Only the first 100 trials are shown to be comparable to ADP agent

## Results - The passive ADP learning curves for the 4x3 world, given the optimal policy



(a)



(b)

- (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice the large changes occurring around the 78th trial—this is the first time that the agent falls into the -1 terminal state at (4,2)
- (b) The root-mean-square error in the estimate for  $U(1, 1)$ , averaged over 20 runs of 100 trials each



# TD vs passive ADP

---

- Notice that TD does not need a transition model to perform its updates
- The resulting utility estimates will approximate more and more closely those of ADP, but at the expense of increased computation time

# TD agents and Q-learning

---

- Time difference agents may have an optimized way to solve the underlying MDP and find  $U$  without  $p$ , but they still need the transition function  $p$ , in order to decide how to act, because:

$$\pi^*(s) = \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')$$

- There is an alternative TD method, called Q-learning, which learns an action-utility representation instead of learning utilities.
- We will use the notation  $Q(s, a)$  to denote the value of doing action  $a$  in state  $s$ .
- Q-values are directly related to utility values as follows

$$U(s) = \max_a Q(s, a)$$

# Q-function

---

- As with utilities, we can write a constraint equation that must at equilibrium when the Q-values are correct

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' | s, a) \max_{a'} Q(s', a')$$

- Q-functions may seem like just another way of storing utility information, but they have a very important property:
  - a TD agent that learns a Q-function does not need a model of the form  $P(s' | s, a)$ , either for learning or for action selection.

$$Q(s, a) \leftarrow Q(s, a) + \alpha (R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

$$\pi^*(s) = \operatorname{argmax}_{a \in A(s)} \sum_{s'} Q(s, a)$$

- For this reason, Q-learning is called a **model-free method**

# Defining the Q Function: value of the pair state-action

---

- Cumulative value  $V^\pi(s_t)$  achieved by following an arbitrary policy  $\pi$  from an arbitrary initial state  $s_t$

$$V^\pi(s_t) = r(s_t) + \gamma r(s_{t+1}) + \gamma^2 r(s_{t+2}) + \gamma^3 r(s_{t+3}) + \dots$$

$$= \sum_{i=0}^{\infty} \gamma^i r(s_{t+i})$$


- Optimal policy:

Notation:  $V^{\pi^*}(s) = V^*(s)$

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

Let's take an action  $a$  and then continue optimally

$$\pi^* = \arg \max_a [r(s,a) + \gamma V^*(s')], \forall s, s' \in S$$


$$Q^*(s,a)$$

## Q Function: value of the pair state-action

---

- $Q^*(s,a)$  is the expected discounted future reward for starting in state  $s$ , taking  $a$  as our first action, and then continuing optimally.

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} Q^*(s',a')$$

$$\pi^* = \arg \max_a Q^*(s,a)$$

- We will see an algorithm to estimate  $Q^*$

# Um algoritmo para o aprendizado da Função Q (Q-Learning)

---

- O algoritmo Q-Learning (Watkins, 1989) baseia-se em simulações de Monte Carlo e no algoritmo Robbins-Monro
  - Métodos Monte Carlo são um amplo grupo de algoritmos que baseiam-se na amostragem de distribuições aleatórias para obter resultados numéricos
  - Simulações Monte-Carlo em RL baseiam-se na amostragem de estados para estimar seus valores (abordagem força bruta)
  - Algoritmo Robbins-Monro permite aprender uma função onde um de seus parâmetros é uma variável aleatória com distribuição de probabilidade conhecida, utilizando uma taxa de aprendizagem  $\alpha$  que se altera ao longo do tempo segundo certas condições.
- Maiores informações sobre a dedução do Q-Learning, Monte Carlo e Robbins-Monro podem ser obtidas em:
  - Reinforcement Learning: An Introduction, Sutton, R. and Barto, A. MIT Press. 1998

# Learning rate and convergence

---

- The basic form of the update looks like this:

$$X_{t+1} \leftarrow (1-\alpha) X_t + \alpha New_t$$

$\alpha$  is a learning rate; usually it's something like 0.1.

- ◆ So, we're updating our estimate of  $X$  to be mostly like our old value of  $X$ , but adding in a new term  $New$ 
  - » This kind of update is essentially a running average of the new terms received on each step.
- ◆ The smaller alpha is, the longer term the average is. With a small alpha, the system will be slow to converge, but the estimates will not fluctuate very much.
- ◆ **It is quite typical (and, in fact, required for convergence), to start with a large alpha, and then decrease it over time.**



## *Guaranteed to converge to $Q^*$ if...*

---

- ◆ The optimal Q function is achieved if the world is really an MDP, if we manage the learning rate correctly, and if we explore the world in such a way that we never completely ignore some actions.



# Algorithm for Q-learning

---

- Q-learning can be seen as a Time difference method, as mentioned before. We can iterate Q values using an adapted version of time difference update equation

$$Q(s, a) \leftarrow Q(s, a) + \alpha (R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

- or

$$Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha [r(s, a) + \gamma \max_{a'} Q(s', a')]$$

# Q-Learning basic idea

---

- Estimates the  $Q^*$  function directly, without estimating the transition probabilities

Initialize  $Q(s,a)$  arbitrarily

Observe the current state  $s$

**do forever**

    select an action  $a$  and execute it

    receive immediate reward  $r(s,a)$

    observe the new state  $s'$

    update  $Q(s,a)$  as follows:

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha [r(s,a) + \gamma \max_{a'} Q(s',a')]$$

$\alpha$ : *learning rate*

- We need to manipulate the learning rate as shown in TD discussion

# Q-Learning

---

- There are **two iterative processes** going on:
  1. One is the usual kind of averaging we do, when we collect a lot of samples and try to estimate their mean (using the learning rate)
  2. The other is the dynamic programming iteration done by value iteration, updating the value of a state based on the estimated values of its successors.

# Q-learning Algorithm

---

**function** Q-LEARNING-AGENT(*percept*) **returns** an action

**inputs:** *percept*, a percept indicating the current state  $s'$  and reward signal  $r'$

**persistent:**  $Q$ , a table of action values indexed by state and action, initially zero

$N_{sa}$ , a table of frequencies for state–action pairs, initially zero

$s, a, r$ , the previous state, action, and reward, initially null

**if** TERMINAL?( $s$ ) **then**  $Q[s, None] \leftarrow r'$

**if**  $s$  is not null **then**

    increment  $N_{sa}[s, a]$

$Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$

$s, a, r \leftarrow s', \operatorname{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$

**return**  $a$

It is an active learner that learns the value  $Q(s, a)$  of each action in each situation. It uses the same exploration function  $f$  as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

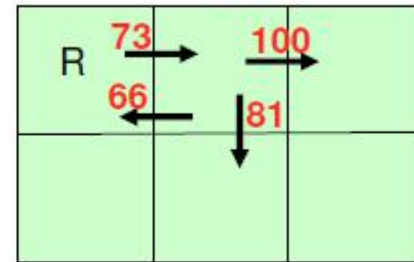
Consider the current  $Q$  values given in arrows and robot in

- Consider a single action (right) taken by the robot R in  $s_1$ :

**Calcular atualização de  $Q$**   
( $\alpha=1$  e MDP determinístico)

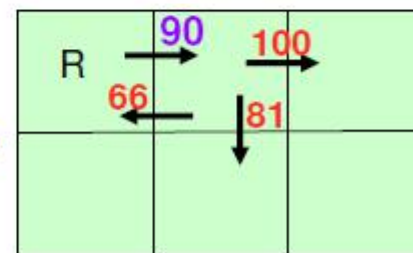
$$\begin{aligned}
 Q(s_1, a_{\text{right}}) &\leftarrow r + \gamma \max_{a'} Q(s_2, a') \\
 &\leftarrow 0 + 0.9 \max \{66, 81, 100\} \\
 &\leftarrow 90
 \end{aligned}$$

$\gamma = 0.9$   
 $Q(s_1, a)$  values



State  $s_1$

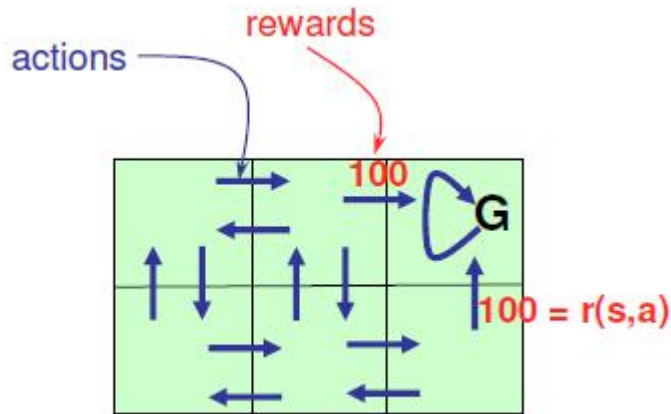
Action: right  
 $r(s_1, \text{right})=0$



State  $s_2$



# Complete Example (deterministic transitions)



G: absorbing state

$\gamma = 0.9$

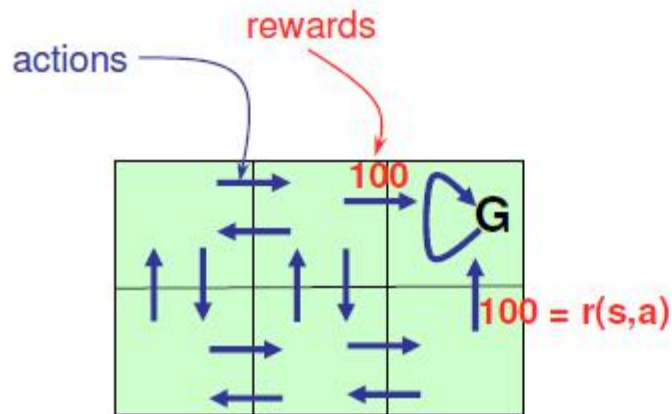
$r(s,a)=0$  otherwise

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha [r(s,a) + \gamma \max_{a'} Q(s',a')]$$

- Let's start with zero Q values and use the update equation
- Remember also the Value iteration for estimating  $V^*(s)$

$$V_{i+1}(s) \leftarrow \max_a r(s,a) + \sum_{s'} \gamma p(s'|s,a) V_i(s')$$

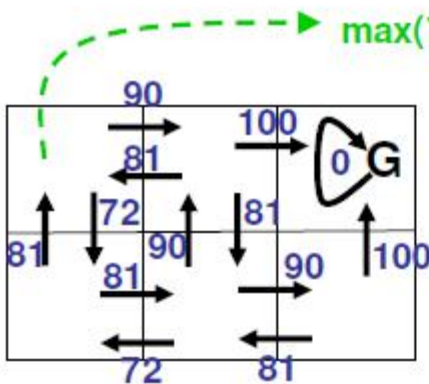
# Example (deterministic transitions)



G: absorbing state

$\gamma = 0.9$

$r(s,a) = 0$  otherwise

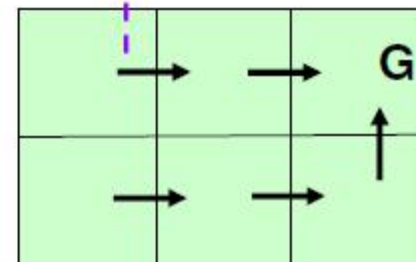


Q(s,a) values

90	100	0
81	90	100

$V^*(s)$  values

actions  $\downarrow$  and  $\rightarrow$ , choose  $\rightarrow$



One optimal policy

# Exemplos de aplicações

---

- ◆ **TD Gammon:** starts out not knowing anything about backgammon. It plays more than 2 **million** games of backgammon against itself. It can now draw the human world-champion backgammon player.
- ◆ **Elevator scheduling:** in a building with many floors and many elevators, there is a serious control problem in deciding which elevators to send to which floors next. The input to the system is the locations of the elevators and the set of all buttons that have been pressed. The output is a direction for each elevator so that the throughput of people could be maximized. The learned policies are considerably more effective than the ones that are standardly built in by the elevator companies.



# Problemas com Q-Learning

---

- **Large or continuous state spaces**

- ◆ Just like value iteration, it requires that  $S$  and  $A$  be drawn from a small enough set that we can store the  $Q$  function in a table.
- ◆ Possible solutions: use of function approximators (e.g. neural network) to store the  $Q$  function. Such approaches are no longer theoretically guaranteed to work, and they can be a bit tricky, but sometimes they work very well.

# Problemas com Q-Learning - 2

---

- **Slow convergence**

- ◆ Because of this, most of the applications of Q learning have been in very large domains for which we actually know a model:
  1. we use the known model to build a simulation.
  2. then, using Q learning plus a function approximator, we learn to behave in the simulated environment, which yields a good control policy for the original problem.

# Outro algoritmo para RL: SARSA

---

- Q-learning has SARSA a close relative called SARSA (for State-Action-Reward-State-Action).
- The update rule for SARSA is very similar to Q-Learning

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma Q(s', a') - Q(s, a))$$

- The difference is that there is a policy for SARSA and it waits to define the next action to do the update.
- Q-learning uses the best Q-value, it pays no attention to the actual policy being followed—it is an **off-policy** learning algorithm

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

- In fact, if SARSA uses a greedy policy that always takes the action with best Q-value, the two algorithms are identical

## SARSA: State action Reward State action

---

- Despite the fact, that sarsa uses a policy the values  $Q(s, a)$  converge for that optimal values, given that  $\alpha$  decreases slowly as discussed before
- The algorithm for SARSA is basically the same algorithm for Q-learning seen before by using some arbitrary initial policy  $\pi$  and changing the update function to

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma Q(s', a') - Q(s, a))$$

# Q-Learning x SARSA (Outro algoritmo)

---

- Q-Learning
  - é o método mais usado
  - É do tipo off-policy (não é necessário seguir uma política)
- Sarsa
  - Por eliminar o uso de uma função de maximização, tende a ser mais rápido que Q-Learning, quando há grande número de ações possíveis
  - Tem basicamente as mesmas condições de convergência
  - Permite descontar diferenças temporais gerando um Sarsa( $\lambda$ ) similar a algoritmos TD( $\lambda$ )

# Q-learning, SARSA and ADP agent

---

- Both Q-learning and SARSA learn the optimal policy for the  $4 \times 3$  world, but do so at a much slower rate than the ADP agent.
- This is because the local updates do not enforce consistency among all the Q-values via the model.
- The comparison raises a general question:
  - is it better to learn a model and a utility function or...
  - to learn an action-utility function withno model?

# Model the world or Not??

---


- Some researchers, both inside and outside AI fields, argue model-free methods such as Q-learning means that the **knowledge-based approach is unnecessary.**
- *“There is, however, little to go on but intuition. Our intuition, for what it’s worth, is that as the environment becomes more complex, the advantages of a knowledge-based approach become more apparent..”*
  - Russel. S. and Norvig. P. 3th ed. AI: A modern approach
- What do you think about it?

# Conclusões sobre Aprendizado por Reforço


---

- Aprendizado por reforço permite que se aprenda a política ótima, mesmo sem saber previamente a função de probabilidade de transição ( $p$ ) ou a função de recompensa imediata ( $r$ )
- Aprendizado por reforço tem dificuldades em lidar com grande número de estados ou grandezas contínuas, vários algoritmos alternativos (sarsa, por exemplo) tentam obter treinamento mais rápido
- Aproximações da função  $Q$  (redes neurais) ou mais recentemente deep neural networks (Deep Reinforcement Learning – DQN) tem sido pesquisados com resultados promissores...
- Mais referências:
  - Reinforcement Learning: An Introduction, Sutton, R. and Barto, A. MIT Press. 1998
  - Bertsekas, D. and Tsitsiklis, J.N. Neurodynamic programming. Athena Scientific. Belmont. Massachusetts. 1996





*Partially observed Markov  
Decision Process*



# PARTIALLY OBSERVABLE MDP (POMDP)

---

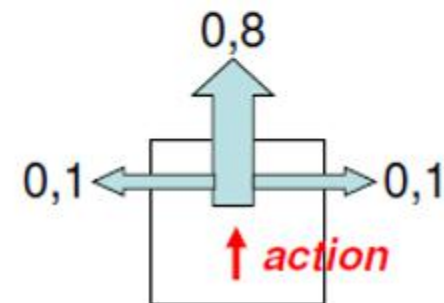
- So far, we have assumed that the environment was fully observable, i.e., the agent always knows which state it is in.
- When the environment is only partially observable, the situation is much less clear
- The agent does not necessarily know which state it is in, so it cannot execute the action  $\pi(s)$  recommended for that state.
- Furthermore, the utility of a state  $s$  and the optimal action in  $s$  depend not just on  $s$ , but also on how much the agent knows when it is in  $s$
- For these reasons, partially observable MDPs are usually viewed as much more difficult than ordinary MDPs
- However, we cannot avoid POMDPs because the real world is one

# POMDP - 4x3 world (but the agent does perceives its current state)

- **Utility function** for the agent depends on a sequence of states (environment *history*)
- In each state  $s$ , the agent receives a **reinforcement**  $r(s)$ , which may be positive or negative, but must be **bounded**.
- Utility = sum of the rewards received
- Here:  $r(s) = -0.04 \quad \forall s$  except  $r(4,3) = +1$  and  $r(4,2) = -1$

-0,04	-0,04	-0,04	+1 <i>goal</i>
-0,04		-0,04	-1
-0,04	-0,04	-0,04	-0,04
<i>start</i>			

Transition model:



# POMDP - 2

---

- A POMDP has the same elements as an MDP—the transition model  $P(s'|s, a)$ , actions  $A(s)$ , and reward function  $R(s)$ —but, it also has a sensor model  $P(e|s)$
- The sensor model specifies the probability of perceiving evidence  $e$  in state  $s$   $P(e|s)$
- In POMDPs, the belief state  $b$  becomes a probability distribution over all possible states
- For the 4x3 world (version POMDP), the initial belief state could be the uniform distribution over the nine nonterminal states, i.e.,  $\langle 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 0, 0 \rangle$ ,  $b(s)$  is the probability of being in state  $s$  given by the belief state  $b$

# POMDP - 3

---

- The agent can calculate its current belief state as the conditional probability distribution over the actual states given the sequence of percepts and actions so far
- If  $b$  was the previous belief state, and the agent does action  $a$  and then perceives evidence  $e$ , then the new belief state is given by ( $\alpha$  is the normalization factor)

$$b'(s') = \alpha P(e|s') \sum_s P(s'|s, a) b(s)$$

- We can define a  $b'$  equation like that for each possible state  $s$ . Note that we don't need to know the current state, the next belief state  $b'$  can be defined as:
  - $b' = \text{Forward}(b, a, e)$

# POMDP - 4

---

- How can one find the optimal policy in POMDP?
- The fundamental insight required to understand POMDPs is this: the optimal action depends only on the agent's current belief state
- That is, the optimal policy can be described by a mapping  $\pi^*(b)$  from belief states to actions.

- 
- The decision cycle of a POMDP agent can be broken down into the following three steps:
    1. Given the current belief state  $b$ , execute the action  $a = \pi^*(b)$ .
    2. Receive percept  $e$ .
    3. Set the current belief state to

$$b'(s') = \alpha P(e|s') \sum_s P(s'|s, a) b(s)$$

4. go back to step 1