Artificial Intelligence

Augustin Cosse.

Fall 2021

November 17, 2021

Reinforcement learning

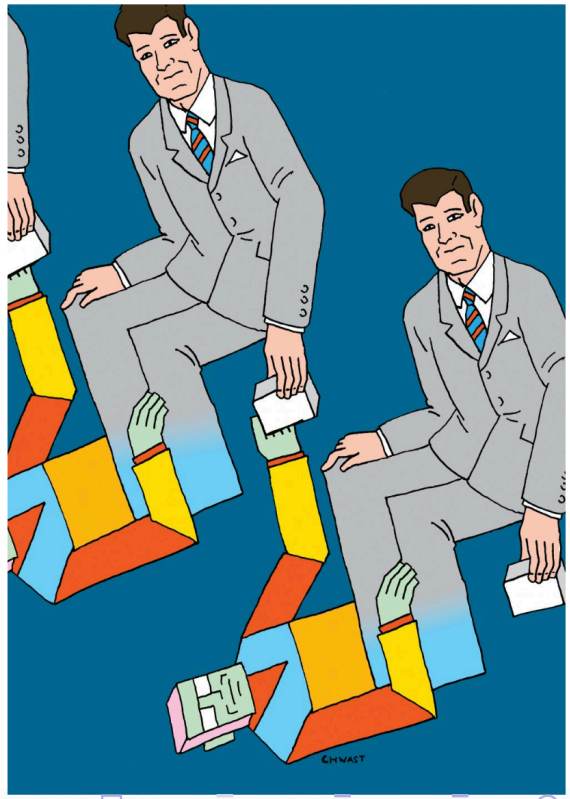
- Reinforcement learning is learning what to do so as to maximize a numerical reward signal
- "The learner is not told which action to take but instead must discover which action yield the most reward by trying them"
- Ex.1.: "A chess player makes a move. The choice is informed by planning (anticipation of possible replies and counterreplies) and by immediate intuitive judgements of the desirability of possible positions and moves"
- Ex.2. "A mobile robot decides whether it should enter a room in search of a target or start to find its way back to its battery charging station"

source: R. Sutton, A.G. Barto, Reinforcement learning: An introduction

10 BREAKTHROUGH TECHNOLOGIES

Reinforcement Learning

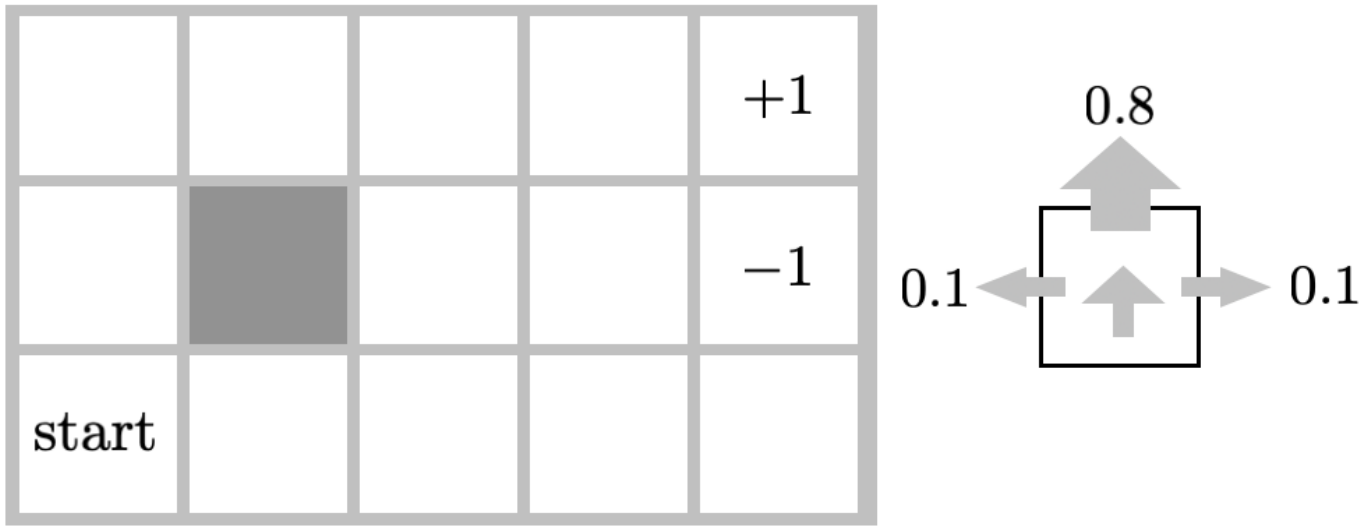
By experimenting, computers are figuring out how to do things that no programmer could teach them.



Availability: 1 to 2 years

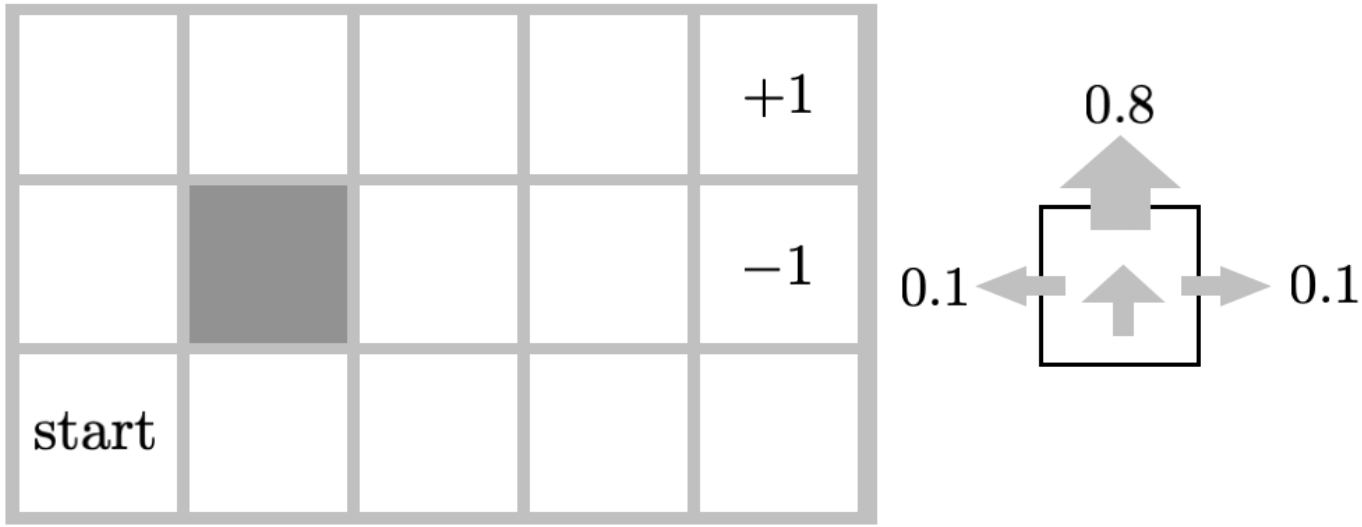
by Will Knight

Markov decision processs



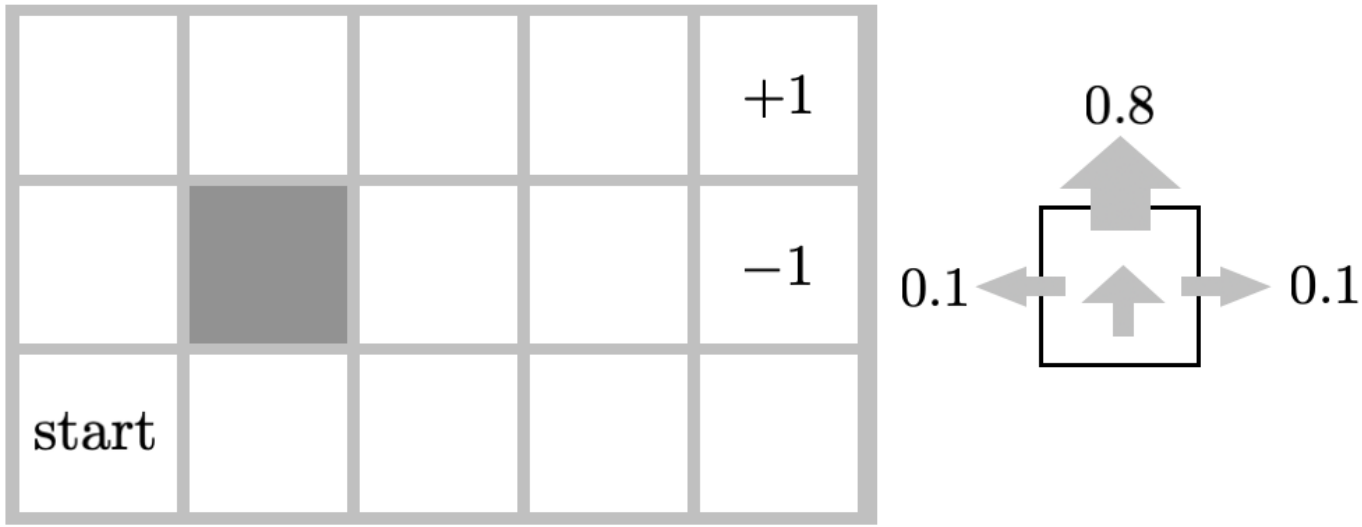
- Consider a simple environment as shown above, where the actions available to the agent in each state are encoded as $\textsc{action}(\textsc{s})$
- In every cell, the agent has the choice of moving up, down, left or right.
- We assume that the environment is fully observable so that the agent knows exactly where it is.

Markov decision processs



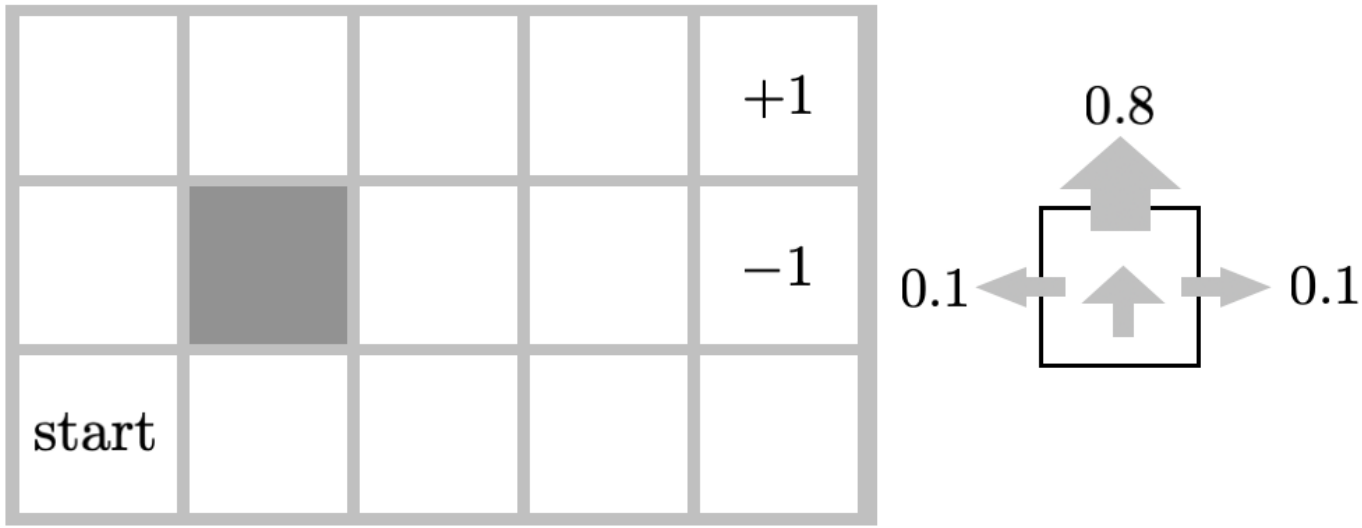
- In a deterministic framework, the optimal sequence of actions would therefore be given by Up-Up-Right-Right-Right-Right
- In this case, we assume that the actions are unreliable. In particular for any intended move, there is only a 80% chance that the action will achieve the intended effect.

Markov decision processs



- We can then define a transition model $P(s'|s,a)$ that encodes the probability of reaching state s' if action a is performed in state s
- In a first approach $P(s'|s,a)$ can be define as a $3D$ table containing probabilities. In more refined frameworks, the model can be encoded as a dynamic Bayesian network

Markov decision processs



- We will assume that the transition model is Markovian, in the sense that the probability of reaching state s' from s depends only on s and not on the history of earlier states.

Markov decision processs

- To complete our definiton of the task environment, we must also specify the utility function.
- Since the problem is sequential in this case, the utility will depend on a sequence of states (known as an environment history)
- For now, we will simply encode this utility as a reward $R(s)$ that the agent receives at every step, which may be positive or negative.
- For now, the utility of environment history is just the sum of the rewards received.

Markov decision processs

- A sequential decision problem for a fully observable stochastic environment with a Markovian transition model and additive rewards is called a Markov Decision Process (**MDP**)
- The next question we need to address is : what does a solution to the problem look like?
- A solution should specify what the agent should specify what the agent should do for any state that the agent might reach. A solution of this type is called a policy. It is traditional to denote a policy with the greek letter π. $\pi(s)$ is thus the action recommended by the policy π for state s.
- If the agent has a complete policy, no matter what the outcome of an action is, teh agent will always know what to do next.

Markov decision processs

- The quality of a policy is measured by the expected utility of the possible environment history generated by that policy
- An optimal policy is a poicy that yields the highest expected utility. We usually use the symbol π^* to denote an optimal policy.
- Given π^*, the agent decides what to do by consulting its current percepts, which tells it the current state s, and then executing the action $\pi^*(s)$.

Markov decision processs

- When the optimal action in a given state could change over time, we say that the optimal policy is nonstationnary
- When the optimal action depends only on the current state, we say that the optimal policy is stationnary
- The next thing we need to decide is how to compute the utility of a state sequence.
- Each state s_i can be viewed as an attribute of the state sequence $[s_0, s_1, s_2, \ldots]$. To obtain a simple expression of our utility as a function of the attributes, we will make some simplifying assumption.

Markov decision processs

- We will first assume that the agent's preferences between state sequences are stationnary
- This implies that if two sequences $[s_0, s_1, s_2, \ldots]$ and $[s'_0, s'_1, s'_2, \ldots]$ begin with the same state (i.e. $s_0 = s'_0$), then the two sequences should be preference ordered the same way as the sequences $[s_1, s_2, \ldots]$ and $[s'_1, s'_2, \ldots]$.
- In other words, if you prefer one future to another starting tomorrow, then you should still prefer that future if it were to start today instead.

Markov decision processs

- Stationnarity seems like an innocuous assumption but it has important consequences
- It turns out that under stationnarity there are only two coherent ways to assign utilities to sequences:
  - Additive rewards according to which the utility of a state sequence is defined as

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \ldots$$

  - Discounted rewards according to which the utility of a sequence is defined as

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

    Where the discount factor γ is a number between 0 and 1. The discount factor describes the preference of an agent for current rewards over future rewards.

Markov decision processs

- When the utility of a given state sequence is the sum of discounted rewards (hence is finite), we can compare policies by comparing their expected utilities .
- We will assume that the agent is in some initial state s and we will define as S_t (a random variable), the state that the agent reaches at time t when executing a particular policy π. Obviously $S_0 = s$
- The probability distribution over state sequences $S_1, S_2, \ldots,$ is determined by the initial state s, the policy π, and the transition model for the environment.

Markov decision processs

- The expected utility obtained by executing π starting at s is then given by

$$U^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

  Here the expectation is taken over state sequences determined by s and π.

- Now out of all the policies that the agent could choose to execute starting in s, one (or more) will have higher expected utlity than the others. We will use π^* to denote one of those policies

$$\pi_s^* = \underset{\pi}{\text{argmax}}\ U^{\pi}(s)$$

Markov decision processs

- Note that the expected utility of an action given some evidence e (e.g. the fact that we are in a state s), is simply the average value of the possible outcomes, weighted by the probability that the particular outcome will occur:

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

- The optimal policy can be computed through an approach known as value iteration

- Recall that the utility of being in a state can be defined as the expected sum of the discounted rewards from that point onwards

Markov decision processs

- From this it follows that one can decompose the utility of a state as the immediate reward for that state (that the agent fot when it reached that state) plus the expected discounted utility of the next states (given that the agent always choose its action so as to maximize this utility):

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

The equation above is known as the Bellman equation named after Richard Bellman (1957)

Markov decision processs: The value iteration algorithm

- The Bellman equation is the basis of the value iteration algorithm for solving MDPs.
- If there are n possible states, there are n possible Bellman equations, for each state
- Moreover, each equation contains n unknowns corresponding to the utilities of each state
- The only problem is that the equations are non linear because the 'max' operator is not a linear operator
- Although one cannot solve the system through simple linear algebra techniques (as we could have for a system of linear equations), one can try to use an iterative approach

Markov decision processs: The value iteration algorithm

- As we will see, there are several approaches at solving the MDP. The approaches most often discussed include
  - Value iteration
  - Policy iteration
  - Temporal difference learning
- In Value iteration, we start with arbitrary initial values for the utilities of each state, calculate the right handside of the Bellman equation and then update the left handside with this right handside

Value iteration algorithm

- Unlike policy iteration, Value iteration does not require any policy evaluation (hence avoid the multiple sweeps needed in this second approach)
- Let $U_i(s)$ denote the utility value for the state s at the i^{th} iteration. The Value Iteration step, also known as Bellman Update then reads as

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

Function `Value-Iteration(`*mdp, varepsilon*`)`**:**

input : *mdp*: an mdp with states S, actions $A(s)$, transition model $P(s'|s,a)$, reward $R(s)$, discount γ
ε, the maximum error allowed in the utility of any state

local variables: U, U': vectors of utilities for states in S, initially zero
δ: the maximum change in the utility of any state at any iteration

while $\delta < \varepsilon(1-\gamma)/\gamma$ **do**
  $U \leftarrow U'; \delta \leftarrow 0$ **for** *each state s in S* **do**
    $U'[s] \leftarrow R[s] + \gamma \max_{a \in A[s]} \sum_{s'} P(s'|s,a) U[s']$
    if $|U'[s] - U[s]| > \delta$ **then**
      $\delta \leftarrow |U'[s] - U[s]|$
    end
  end
end
return U

Value iteration algorithm

- If we apply the Bellman update infinitely often, we are guaranteed to reach an equilibrium in which case the final utility values must be solutions to the Bellman equations
- To show convergence of the Value iteration, we will need the notion of contraction.
- In this case, roughly speaking, a contraction $f(x)$ can be understood as a function of one argument that, when applied to two different inputs produces outputs that are closer together than the original inputs,

$$\|f(x) - f(y)\| \leq \gamma \|x - y\|$$

- A point that is unchanged by the application of the contraction is called a fixed point

Value iteration algorithm

- A contraction has only one fixed point. If there were two fixed points, those would not get closer together when the function is applied.

- When the function is applied to any argument, the result must be close to the fixed point (since the application of the contraction to the fixed point gives the point itself), i.e. for any fixed point x^*, and any y we have

$$\|f(x^*) - f(y)\| = \|x^* - f(y)\| \leq \gamma \|x^* - y\|$$

We now encode the Bellman update as

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

$$U(s) = \mathcal{B} U(s)$$

Value iteration algorithm

- The proof of convergence will consist in showing that for the max norm $\|U\| = \max_s |U(s)|$ the Bellman update $U \leftarrow \mathcal{B}U$ is a contraction

$$\|\mathcal{B}U_i - \mathcal{B}U_i'\| \leq \gamma \|U_i - U_i'\|$$

- Note that once we know that $\mathcal{B}$ is a contraction, we can use it to measure the speed of convergence to any fixed point. I.e. for any fixed point U, we have

$$\|\mathcal{B}U_i - U\| \leq \gamma \|U_i - U\|$$

The equation above shows that the value iteration converges exponentially fast to the fixed point.

Value iteration algorithm

- To prove the convergence to a fixed point, we first show that for any two functions f and g, we have

$$\left|\max_a f(a) - \max_a g(a)\right| \leq \max_a |f(a) - g(a)|$$

  To see this first note that

  - $\left|\max_x f(x) - g(\operatorname{argmax}_x f(x))\right| \leq \max_a |f(a) - g(a)|$
  - $\left|\max_x g(x) - f(\operatorname{argmax}_x g(x))\right| \leq \max_a |f(a) - g(a)|$

Value iteration algorithm

- Then note that if $\max_x f(x) \geq \max_x g(x)$, we can write
  - $\left|\max_x f(x) - \max_x g(x)\right| \leq \left|\max_x f(x) - g(\operatorname{argmax}_x f(x))\right|$
- And if $\max_x f(x) \leq \max_x g(x)$, we have
  - $\left|\max_x g(x) - \max_x f(x)\right| \leq \left|\max_x g(x) - f(\operatorname{argmax}_x g(x))\right|$

  which gives the conclusion of the first claim.

Value iteration algorithm

- It then remains to apply the first claim to the Bellman update, which gives

$$
\begin{aligned}
&\left\|\mathcal{B}U_i - \mathcal{B}U_i'\right\| \\
&= \max_s \left| \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s') - \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i'(s') \right| \\
&\leq \max_s \max_{a \in A(s)} \left| \gamma \sum_{s'} P(s'|s,a) U_i(s') - \gamma \sum_{s'} P(s'|s,a) U_i'(s') \right| \\
&\leq \gamma \max_s \max_{a \in A(s)} \sum_{s'} P(s'|s,a) \left| U_i(s') - U_i'(s') \right| \\
&\leq \gamma \max_s \max_{a \in A(s)} \max_{s'} \underbrace{\left| U_i(s') - U_i'(s') \right|}_{\|U_i - U_i'\|} \sum_{s'} P(s'|s,a) \\
&\leq \gamma \|U_i - U_i'\|
\end{aligned}
$$

Value iteration algorithm: Existence of fixed point

- To prove the existence of the fixed point (althgouh formally we should be more precise), note that if recursively apply the Bellman update to an initial utility U_0, we get

$$\left\|\mathcal{B}^n U_0 - \mathcal{B}^{n-1} U_0\right\| \leq \gamma^{n-1} \left\|\mathcal{B} U_0 - U_0\right\|$$

  In particular if we define $U_u \equiv \mathcal{B} U_0$, $U_{n-1} \equiv \mathcal{B}^{n-1} U_0$ as well as $U_1 \equiv \mathcal{U}_0$, this implies

$$\left\|\mathcal{B} U_{n-1} - U_{n-1}\right\| \leq \gamma^{n-1} \left\|U_1 - U_0\right\|$$

  which for a sufficiently small γ and a sufficient number of Bellman updates gives $\lim_{n\to\infty} \mathcal{B} U_{n-1} = \lim_{n\to\infty} U_{n-1}$

Value iteration algorithm

- Recall that in the value update, we repeat the update for each possible state $s \in S$

- We could wonder how far we would be from the fixed point if at step i, we stoped the value iterations and started executing our estimate of the optimal policy, that is by taking each time the action that maximizes $\sum_{s' \in S} P(s'|s,a)U(s')$. It turns out that in this case, we have

$$\|U_i - U\| < \varepsilon \quad \Rightarrow \quad \|U^{\pi_i} - U\| < 2\varepsilon\gamma/(1-\gamma)$$

Note that U^i (which is the estimate of U at step i) is different from the expected utility U^{π_i} which, if we let $S_0 = s$ and denote by S_t the state reached at step t by executing the policy π^i form step S_{t-1} is given by

$$U^{\pi_i}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

Policy Iteration

- From the Value iteration approach, we saw that it was possible to get an optimal policy even when our estimate of the utility is not yet optimal.
- If one action is better than the others, then we might want to focus on this action although our estimate of the utility might not be optimal yet
- This suggest an alternative way to find the optimal policy, by alternating between the following two steps:
  - **Policy evaluation**: given a policy π_i, calculate $U_i = U^{\pi_i}$, the utility of each state if π_i were to be executed
  - **Policy Improvement**: Calculate a new Maximum Expected Utility policy using

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

Policy Iteration

- This second approach is known as **policy iteration**.
- The algorithm terminates when the policy improvement step yields no change in the utility. At this point, we know that we have reached a fixed point of the Bellman. As a consequence U_i must be a solution of this equation and π_i must be an optimal policy
- The policy improvement step is easy but how do we implement POLICY-EVALUATION?

Policy Iteration

- Note that since our actions are defined by the policy $\pi_i(s)$, the Bellman equation simplifies to

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

- In particular, this implies that the Bellman equations are now linear in the utility. Since the 'max' operator has been removed, we now have a set of n linear equations in n unknowns which can be solved in time $O(n^3)$ by relying on linear algebra solvers.

- In general, for small state spaces, policy evaluation using exact solution methods will be the favored approach.

Policy Iteration

- For large state space (i.e. when $O(n^3)$ becomes prohibitive, we will instead rely on a number of simplified value iteration steps of the form

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

  which we can repeat k times to get an our next utility estimate

- This last (iterative) approach is known as **modified policy iteration**.

Function `Policy-Iteration(mdp)`:

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

local variables: U a vector ot utilities for states in S, initially zero
π, a policy vector indexed by state, intially random

while *unchanged* **do**

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

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**

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

unchanged? $\leftarrow$ false

end

end

end

return π

Partially observable MDPs (brief intro)

- The approaches that we have just discussed both assumed that the environment was **fully observable**. In other words, the agent always knows which state it is in.
- Together with the Markov assumption on the transition model, this means that the policy depends only on the current state
- When the environment is only partially observable, the agent does not know which state it is in and hence cannot execute the action $\pi(s)$ corresponding to that state.
- On top of this, the utility of a state and the optimal action in that state depends not only on the state, but also on how much the agent knows when it is in that state.

Reinforcement Learning

- The main objective in reinforcement learning is to use observed rewards (that we already introduced in MDPs) to learn an optimal policy for the environment
- When discussing MDPs, we assumed that the agent had a complete model of the environment (so that the rewards associated to each state were known in advance), in the general RL framework, we assume no prior knowledge at all.
- You can think of a RL agent as playing a new game whose rule it does not know. After a hundred or so moves, its opponents anounces that it lost the game.
- In many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels

Reinforcement Learning

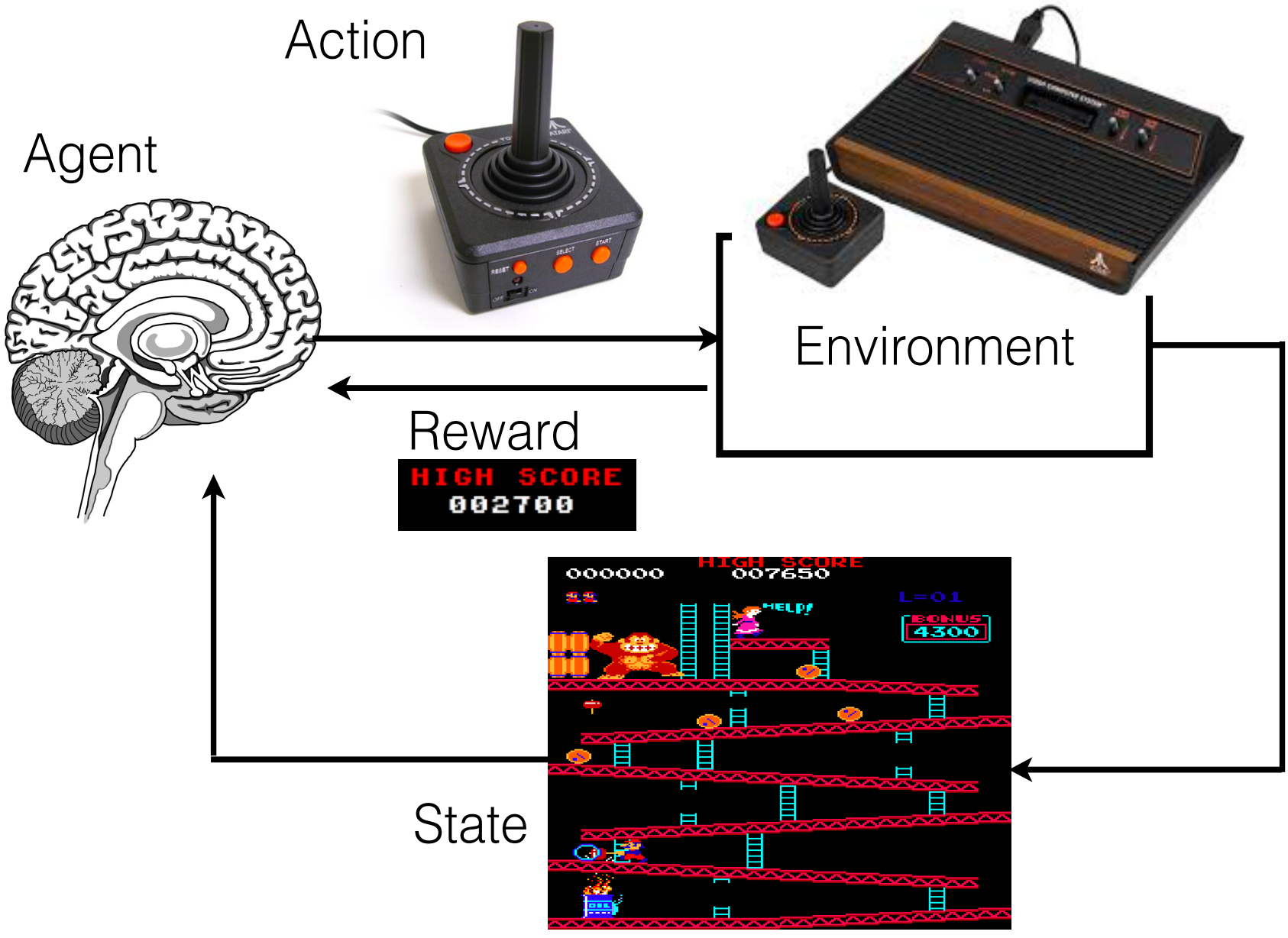
- A typical example of this is game playing. It is usually very hard for a human to provide accuracte and consistent evaluations of large number of positions, which could be used to train a model from examples such as in supervised learning.
- The beauty of reinforcement learning is that in these circumstances, the program can be told when it has won or lost and it can then use this information to learn an evaluation function that gives reasonably accurate estimates of the probability of winning from any given position.

Reinforcement Learning

- Reinforcement learning can be considered to encompass all of AI: an agent is placed in an environment and must learn to behave successfully therein.
- When discussing RL agents, we willl focus on relatively simple environments. In particular, we will assume that the environment is fully observable (hence the current state is supplied by each percept)
- We will however also assume that the agent does not know how the environment works or what its actions do (and we will allow for probabilitic action outcomes)

Reinforcement Learning

- Within the framework of reinforcement learning, we will consider three of the agents designs that were introduced at the beginning of the course:
  - A utility based agent which will learn a utility function on states and uses it to select action that maximize the expected outcome utility
  - A Q-learning agent learns an action utility function or Q-function.
  - A reflex agent learns a policy that maps directly from states to action



Inspired from https://keon.io/deep-q-learning/, Deep Q-Learning with Keras and Gym

Reinforcement learning

- In passive learning the agent's policy is fixed and the task is to learn the utilities of the states
- In active learning, the agent must also learn what to do
- The main issue is exploration: the agent must experience as much as possible of its environment in order to learn how to behave in it.

Passive Reinforcement learning

- In passive learning, the agent's policy $\pi(s)$ is fixed : in state s, the agent always executes the action $\pi(s)$
- The only action of the agent is to learn how good the policy is, that is to say to learn the utility function $U^{\pi}(s)$
- Passive learning is similar to the policy evaluation part of the policy iteration algorithm.
- The only difference is that a passive learning agent does not know the transition model $P(s'|s,a)$. Nor does it know the reward function.
- Recall that the utility is defined as the expected sum of (discounted) rewards obtained if policy π is followed:

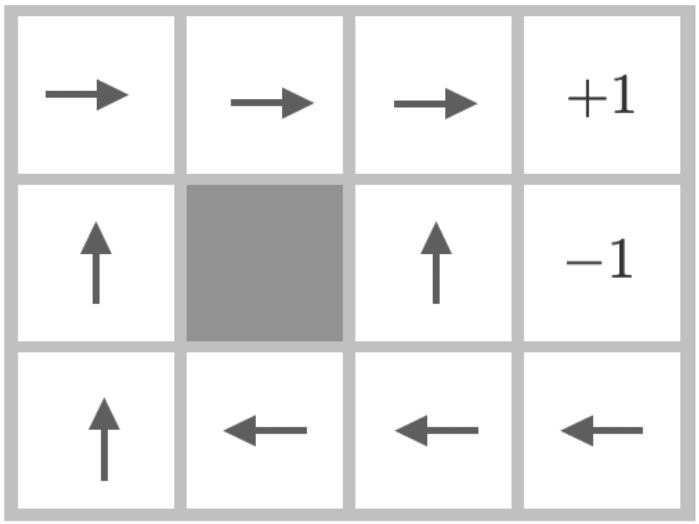
$$U^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

Passive Reinforcement learning: Direct utility estimation

- A simple method for direct utility estimation was invented in the area of adaptive control by Widrow and Hoff
- The idea is that the expected utility of a state is the expected total reward from that state onward
- Each trial thus provides a sample of this quantity for each state visited

Passive Reinforcement learning: Direct utility estimation

- As an example, a trial such as the one shown below in red (and for $\gamma = 1$) provides a sample total reward of 0.72 for state $(1, 1)$, two samples of 0.76 and 0.84 for state $(1, 2)$, two samples of 0.8 and 0.88 for $(1, 3)$ and so on

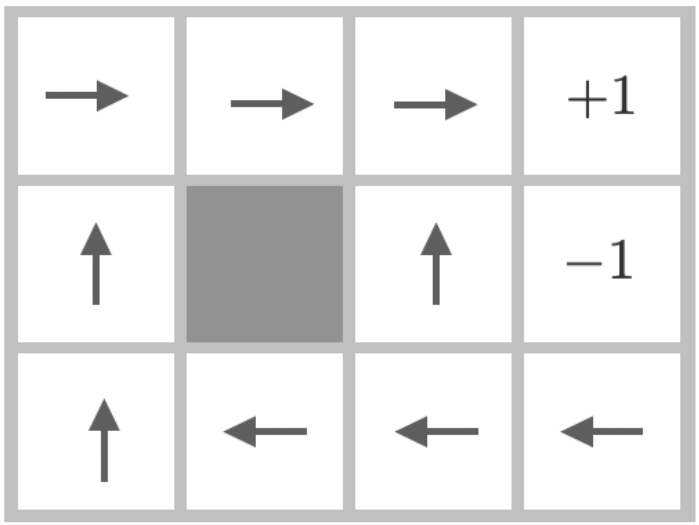


0.812	0.868	0.918	+1
0.762		0.660	−1
0.705	0.655	0.611	0.388

$$(1,1) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (2,3) \underset{-.04}{\rightarrow} (3,3) \underset{-.04}{\rightarrow} (4,3) \underset{+1}{\rightarrow}$$

Passive Reinforcement learning: Direct utility estimation

- From this we can define an algorithm that, at then end of each sequence, will calculate the expected reward to go for each state and update the utility for that state accordingly

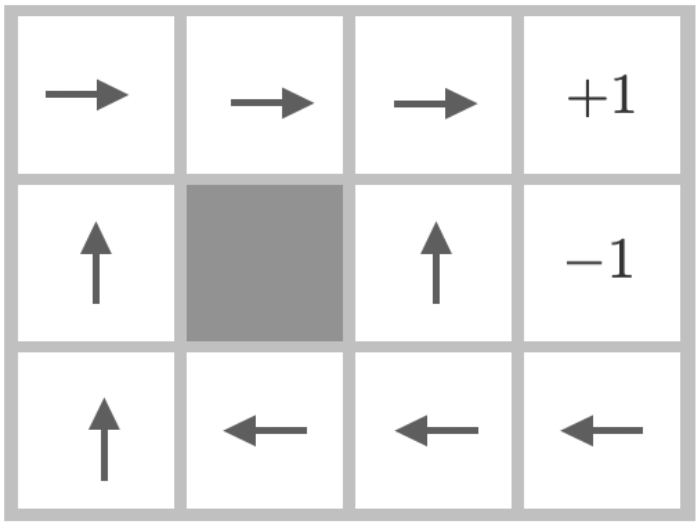


0.812	0.868	0.918	+1
0.762		0.660	−1
0.705	0.655	0.611	0.388

$$(1,1) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (2,3) \underset{-.04}{\rightarrow} (3,3) \underset{-.04}{\rightarrow} (4,3) \underset{+1}{\rightarrow}$$

Passive Reinforcement learning: Direct utility estimation

- In the limit of infinitely many trials, the sample average will converge to the true expectation



0.812	0.868	0.918	+1
0.762		0.660	−1
0.705	0.655	0.611	0.388

$$(1,1) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (2,3) \underset{-.04}{\rightarrow} (3,3) \underset{-.04}{\rightarrow} (4,3) \underset{+1}{\rightarrow}$$

Passive Reinforcement learning: Direct utility estimation

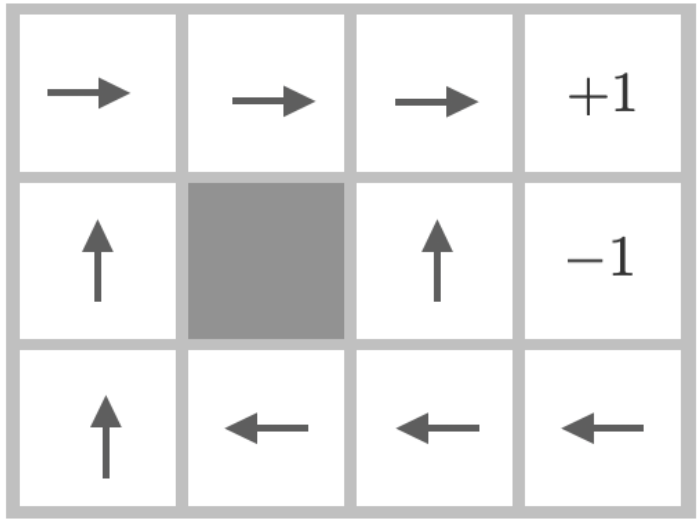
- Direct utility estimation reduces the reinforcement learning problemto an inductive learning problem but misses a very important aspect : the fact that the utilities of states are not independent
- That is, the the utility of each state equals its own reward plus the expected utility of its successor states

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

Passive Reinforcement learning: Direct utility estimation

- By ignoring the connection between states, direct utility estimation misses opportunities for learning

- As an example, from the second trial below, the agent will know that the state $(3, 2)$ has a high utility because it leads to the state $(3, 3)$ which itself leads to the goal state $(4, 3)$



0.812	0.868	0.918	+1
0.762		0.660	−1
0.705	0.655	0.611	0.388

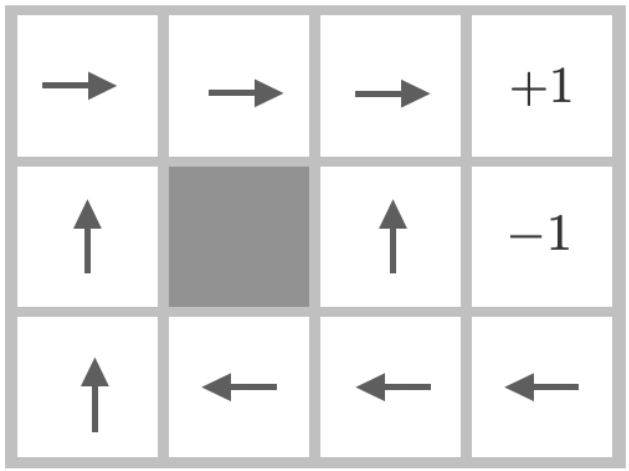
$$(1,1) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (2,3) \underset{-.04}{\rightarrow} (3,3) \underset{-.04}{\rightarrow} (4,3) \underset{+1}{\rightarrow}$$

$$(1,1) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (2,3) \underset{-.04}{\rightarrow} (3,3) \underset{-.04}{\rightarrow} (3,2) \underset{-.04}{\rightarrow} (3,3) \underset{-.04}{\rightarrow} (4,3) \underset{+1}{\rightarrow}$$

Passive Reinforcement learning: Direct utility estimation

- This connection was in fact already highlighted by the Bellman equation due to the connectivity between states but in direct utility estimation, the agent will need to wait until the second experiment to realize the importance of $(3, 2)$

- For this reason, convergence of direct utility estimation is often slow in practice



0.812	0.868	0.918	+1
0.762		0.660	−1
0.705	0.655	0.611	0.388

$$(1,1) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (2,3) \underset{-.04}{\rightarrow} (3,3) \underset{-.04}{\rightarrow} (4,3) \underset{+1}{\rightarrow}$$

$$(1,1) \underset{-.04}{\rightarrow} (1,2) \underset{-.04}{\rightarrow} (1,3) \underset{-.04}{\rightarrow} (2,3) \underset{-.04}{\rightarrow} (3,3) \underset{-.04}{\rightarrow} (3,2) \underset{-.04}{\rightarrow} (3,3) \underset{-.04}{\rightarrow} (4,3) \underset{+1}{\rightarrow}$$

Function `Passive-ADP-Agent(`*percepts*`)`**:**

input : percept, a percept sequence indicating
the current state s' and the reward signal r'

persistent : π, a fixed policy, $mdp(P, R, \gamma)$
U, a table of utilities (initially empty)
N_{sa}, a table of frequencies for sate-action pairs (init. zero)
$N_{s'|s,a}$, a table of outcome frequencies for state action pairs
s, a, the previous state and action

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

Adaptive dynamic programming

- An adaptive dynamic programming (ADP) agent takes advantage of the constraints among the utilities of states, by learning the transition model that connects them and then solving the resulting Markov Decision Process by using a dynamic programming method
- For a passive agent (i.e. known policy $\pi(s)$), this means plugging the learned transition model $P(s'|s, \pi(s))$ and the observed reward $R(s)$ into the Bellman equation to calculate the utilities of the states.
- Since the equations are linear, they can be solved using any linear algebra solver. Alternatively, as we say, we can also use a modified policy iteration.

Adaptive dynamic programming

- The process of learning the model $P(s'|s,a)$ is easy, because the environment is fully observable
- Concretely this means that we have a supervised learning task where the input is a state-action pair and the output is the resulting state.
- In the simplest case, we can represent the transition model as a table of probabilities
- We keep track of how often each action outcome occurs and estimate the transition probability $P(s'|s,a)$ from the frequency with which s' is reached when executing a in s.

Adaptive dynamic programming

- One can in fact show that such an ADP agent uses maximum likelihood to estimate the transition model.
- Note that by choosing a policy based solely on the estimated model, it is acting as if the model were correct, which is not necessarily a good idea
- As an example, a taxi agent that does not know about traffic lights might ignore a red light once or twice with no ill effects and then formulate a policy to ignore red lights from then on
- In order to avoid such a situation, it might be better to choose a policy that, while not necessarily optimal for the mode estimated by maximum likelihood, will however work well for a whole range of models that have a reasonable chance of being the true model.

Adaptive dynamic programming

- There are two mathematical approaches that rely on this idea:

  - **Bayesian reinforcement learning** assumes a prior probability $P(h)$ for each hypothesis h about what the true model is. The posterior $P(h|\text{evidence})$ is obtained using Bayes rule and the observations to date. If the agent has decided to stop learning, the optimal policy is the one that gives the highest expected utility. I.e let u_h^π denote the expected utility, averaged over all possible start states, obtained by executing policy π in model h, we get

$$\pi^* = \operatorname*{argmax}_\pi \sum_h P(h|\text{evidence}) u_h^\pi$$

  - **Robust Control theory** considers a set of possible models $\mathcal{H}$ and defines an optimal robust policy as one that gives the best outcome in the worst case over $\mathcal{H}$

$$\pi^* = \operatorname*{argmax}_\pi \min_h u_h^\pi$$

Temporal Difference Learning (TD)

- Updating the transition probabilities and solving the MDP as in ADP is not the only way to use the Bellman equations
- An alternative is to use the observed transitions to adjust the utilities of the observed states so that they agree with the constraint equations
- This can be done through the following update (known as **temporal difference update**)

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \eta \left(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s)\right)$$

which reduces the difference between the LHS and the RHS in the Bellman equation. η is the learning rate.

Temporal Difference Learning (TD)

- Although this looks like an attractive approach, there is some subtelty involved
  - First, note that the update only involves the observed successors of state s while the actual equilibrium equation involves all the successors (Fortunately, since rare transitions will occur only rarely, the average value of $U^{\pi}(s)$ will converge to the correct value)
  - Second, note that if we change α from a fixed parameter to a function that decreases with the iterations, $U^{\pi}(s)$ (and not its average) will converge to the correct value.
- Possible update rule for α include $\alpha(n) = C/(C+n)$ where C is a large constant (e.g. 60)

Function `Passive-TD-Agent(`*percepts*`)`**:**

input : percept, a percept sequence indicating the current state s' and the reward signal r'

persistent : π, a fixed policy, $mdp(P, R, \gamma)$
U, a table of utilities (initially empty)
N_s, a table of frequencies for states (init. zero)
s, a, r, the previous state and action and reward

if s' *is new* **then**
  $U[s'] \leftarrow r'$
end
if s *is not null* **then**
  increment $N_s[s]$
  $U[s] \leftarrow U[s] + \eta(N_s[s])(r + \gamma U[s'] - U[s])$
end
if $s'.\text{TERMINAL?}$ **then**
  $s, a, r \leftarrow$ null
end
else
  $s, a, r \leftarrow s', \pi[s'], r'$
end
return a

Temporal Difference Learning (TD)

- The ADP and the TD approach are actually closely related. Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors
- One difference, as we saw is that TD adjusts a state to agree with its observed successor while ADP adjusts the state to agree with all of the successors
- As we also saw this difference disappears when the effects of TD adjustments are averaged over a large number of transitions
- Another more important difference is that whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore consistency between the utility estimate U and the environment model P

Active reinforcement learning

- A passive agent has a fixed policy, an active agent must decide what actions to take
- The first difference is that the agent must now learn a complete model with outcome probabilities for all actions rather than just the model for the fixed policy (one approah is to re-use the simple learning mechanism from the ADP agent)
- Next we need to take into account the fact that the agent has a choice of actions

Active reinforcement learning

- The utilities it needs to learn are those defined by the optimal policy. They obey the Bellman equations

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

  Those equations can be solved to obtain the utility function U using the value iteration algorithm or the policy iteration algorithm

- Having obtained the utility function U that is optimal for the learned model, the agent can extract an optimal action by one-step look ahead to maximize the expected utility.

- But is this approach always optimal ?

Active reinforcement learning

- An agent that follows the recommendations of the optimal policy for the learned model at each step is known as a greedy agent.
- A greedy agent very seldomly converges to the optimal policy for a given environment because the learned model is often suboptimal for the true environment.

Active reinforcement learning

- What can be done, then, to improve our agent?
- What the greedy agent overlooks 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 its understanding of the model, the agent might receive greater rewards in the future.
- An efficient agent should therefore always make a tradeoff between exploitation (maximization of its immediate reward) and exploration (which will contribute to the maximization of its long term well being)