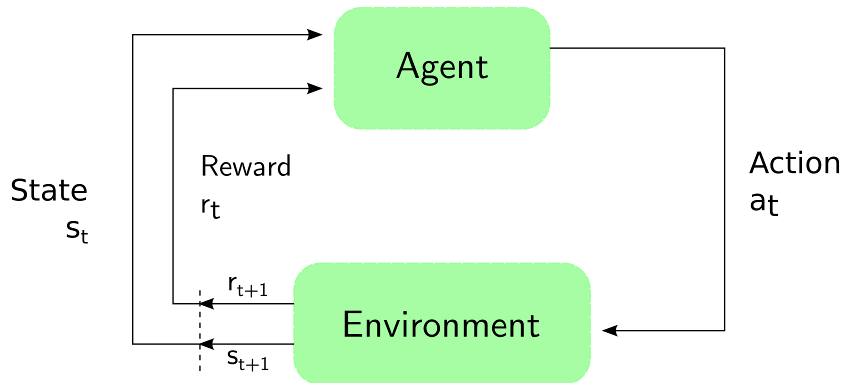
What is reinforcement learning?

The principles of reinforcement learning are simple

- An agent must solve a task and has a set of actions available
- The agent chooses the action or set of actions it believes can solve the task
- The agent receives feedback information from the environment about its performance
- The agent uses this information to modify its behaviour

Reinforcement Learning Agent



Where reinforcement learning is applied?

- Reinforcement learning is used when an agent can interact with the environment
- The environment has to be able to quantify the success or failure of the agent actions
- The state space of the problem can be discrete or continuous
- Different learning goals can be used (assess the quality of a set of actions, find the optimal set of actions) with different complexities (states or effects of the actions unknown, non deterministic effects of the actions, ...)

When reinforcement learning is used?

- It is used for problems too complex to be solved hand coded
- To generate a set of examples to learn the task by classical inductive methods is too costly
- It is more easy to allow the agent to learn the task by trial and error
- Applications: Robotics, computer games, sequencing of manufacturing processes, temporal sequencing, ...

Elements of reinforcement learning

- A set of states that describe the environment (discrete or continuous)
- A set of actions that the agent can perform to modify the environment
- A reinforcement function that informs to the agent of the result obtained from performing the action on the environment
- **Goal:** To find a policy that links states with actions and maximize the gains obtained from the reinforcements

A simple scenario

- Reinforcement learning can be seen as a decision problem
- We will begin with a simple learning scenario

- **N-armed bandit problem:**
  - We have a slot machine with N levers
  - Each lever is an action (a decision)
  - Our reward is the payoffs we obtain
  - The goal is to maximize the gains

N-armed bandit problem

- To learn in this scenario you need to learn the expected payoff of each decision
- This can be obtained by approximating the *action-value* function
- With this function a *greedy policy* can be followed (always the action with higher payoff)
- Following a greedy policy you are *exploiting* your knowledge
- Alternatively you can go off policy (seeking better policies) *exploring* non greedy actions
- The decision between exploitation and exploration will influence the quality of the learning (it depends on the problem)

Learning in the N-armed bandit problem

- We will consider $Q^*(a)$ the true action-value function (the immediate reward of the action)
- We will estimate $Q_t(a)$, the estimate of the action-value function
- Considering the mean of the rewards of an action as an estimate of the true value we can compute $Q_t(a)$ as:

$$Q_t(a) = \frac{r_1 + r_2 + \cdots + r_{k_a}}{k_a}$$

  being k_a the number of times that a has been selected during the t decisions
- With $k_a \rightarrow \infty$ $Q_t(a)$ will converge to $Q^*(a)$

Learning in the N-armed bandit problem

- We can learn the $Q_t(a)$ function by executing the decision problem several times
- A greedy policy (exploitation) can be used during the learning $\rightarrow$ Pick the action with largest reward
- A non greedy policy can be used (exploitation/exploration)
  - The simplest non greedy strategy is the one called ϵ-greedy
  - Given a parameter α ($0 \leq \alpha \leq 1$) a random suboptimal action is picked with probability α, otherwise the best action is used

Learning in the N-armed bandit problem

- For stationary problems (the payoffs do not change with time), the estimate can be incrementally (on-line) updated using:

$$Q_{k+1} = Q_k + \frac{1}{k+1}(r_{k+1} - Q_k)$$

  Being k the number of times we have executed an action
- There are problems that are non stationary, for them makes no sense to give different weights depending on time, the update rule can be changed to:

$$Q_{k+1} = Q_k + \alpha(r_{k+1} - Q_k)$$

  Where $0 \leq \alpha \leq 1$

Reinforcement Learning problems

- The n-armed bandit problem does not associate actions with different circumstances
- We can extend the problem:
  - Multiple n-armed bandits
  - You move from one to the other depending on the actions you take
- This is the model for Reinforcement Learning problems
- The goal is to learn a *policy* (π) that tells what action to perform to move from state to state

Reinforcement Learning problems

- The goal of an agent is formalized using the feedback from the environment, the signal that we call the *reward*
- An agent must maximize not only the immediate, but the future cumulative reward
- We define the cumulative reward of choosing an action after time t as:

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

- This works for *episodic* tasks, T is the time when the final state is reached
- In this kind of problems we consider that the task can be divided in sequences of actions that end in absorbing states

Reinforcement Learning problems

- For *continuous* tasks (no absorbing states) this value could be infinite, so the *discounted* reward is considered ($0 \leq \gamma \leq 1$):

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

- We can unify these two problems using the definition:

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$$

including the possibilities of $T = \infty$ or $\gamma = 1$ (but not both)

 cba

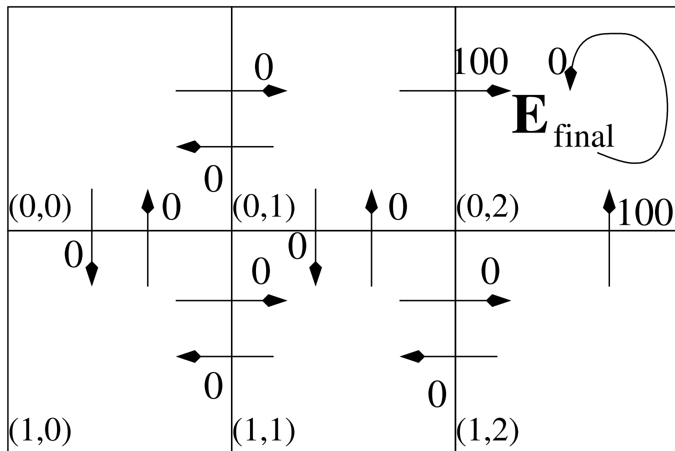
Markov Decision Processes

- There are several possible scenarios for Reinforcement Learning
- We are going to consider:
  - The state can be observed (by the sensors of the agent)
  - The environment provides a reward function
  - We have a set of actions to change the state
  - The task holds the Markov property (future decisions do not depend on the past)
- These problems are called Markov Decision Processes (MDP)
- These problems can be defined by their states, actions and a one step dynamic

Markov Decision Processes

- An agent can perceive a set S of possible states
- There is a set A of actions that can be performed
- Each interval of time (discrete) the agent observes the current state (s_t) and picks an action to perform (a_t)
- The environment gives the agent a reward corresponding with the state and the action ($r_t = r(s_t, a_t)$) and changes to the next state ($s_{t+1} = \delta(s_t, a_t)$)
- r and δ are functions of the environment and are not necessarily known by the agent

Markov Decision Processes (r and δ)



Markov Decision Processes

- A *policy* is a set of actions to be performed to solve a problem
  - For example: The sequences of moves in a game, the path to exit a maze, the control actions for a process, ...
- For a problem we will learn the best policy (π)
- $\pi(s, a)$ will represent the probability of taking action a from state s
- In order to estimate a policy we define functions that estimate:
  - How good is to be in a specific state (*state-value function*)
  - How good is to take an action from a specific state (*action-value function*)
- These functions are defined in terms of the future expected reward
- With these functions the quality of a policy can be evaluated

Markov Decision Processes

- The *state-value function* for a policy π can be defined as:

$$V^{\pi}(s) = E_{\pi}\left\{R_t | s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\}$$

- The *action-value function* for a policy π can be defined as:

$$Q^{\pi}(s) = E_{\pi}\left\{R_t | s_t = s, a_t = a\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\}$$

Markov Decision Processes

- Considering the probability of each possible next s' state given an state s and an action a:

$$\mathcal{P}^a_{ss'} = P(s_{t+1} = s' | s_t = s, a_t = a)$$

- and the expected reward for a state s and an action a with any next state s':

$$\mathcal{R}^a_{ss'} = E\left\{ r_{t+1} || s_t = s, a_t = a, s_{t+1} = s' \right\}$$

- From Dynamic programming it is known that these functions hold a particular recursive relationships (Bellman equations)

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s') \right]$$

Markov Decision Processes

- Solving the Reinforcement Learning problem means to find:

$$V(s) = \max_{\pi} V^{\pi}(s)$$

- There are three main approaches to solving this problem
  - Dynamic programming
  - Monte Carlo approximation
  - Temporal Difference learning

Dynamic Programming

- This technique needs a perfect model of the environment
- That means:
  - The problem has a finite number of states
  - We know the set of states and actions possible at each state
  - We know the transition probabilities $\mathcal{P}^a_{ss'}$
  - We know the expected rewards $\mathcal{R}^a_{ss'}$
- Dynamic Programming provides algorithms for:
  - Policy evaluation (expected reward if we have a policy)
  - Policy improving (changes to a policy to improve its reward)
- The main drawback is the computational cost needed for convergence

Dynamic Programming: Value Iteration algorithm

Procedure: Value Iteration (Policy improving)

Input: $V(s) = 0$ for all $s \in \mathcal{S}$

repeat

  $\Delta \leftarrow 0$

  foreach $s \in \mathcal{S}$ **do**

    $v \leftarrow V(s)$

    $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$

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

  end

until $\Delta < \theta$

Output: Deterministic policy such that

$$\pi(s) = \text{argmax}_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$$

- The algorithm needs to pass through all states several times after convergence

Monte Carlo Methods

- Assumes episodic tasks
- The transition probabilities and the expected rewards are not needed
- These methods use random episodes to estimate the value functions
- If we have a model of the problem we can estimate $V^{\pi}(s)$ for a policy
  - For several random episodes, the reward for each visit of state s is averaged
- If a model is not available we can estimate $Q^{\pi}(s, a)$ for a policy
  - For several random episodes, the reward for each visit of state s transitioning with action a is averaged

Monte Carlo methods: State Value approximation

Procedure: Monte Carlo first visit (policy evaluation)

Input: $\pi \leftarrow$ policy to be evaluated
Input: $V(s) \leftarrow$ arbitrary function
Input: $Returns(s) \leftarrow$ empty list for all $s \in \mathcal{S}$
while *true* **do**
  Generate an episode from π
  foreach *state s in the episode* **do**
    $R \leftarrow$ *return* obtained from the first occurrence of s
    add R to $Returns(s)$
    $V(s) \leftarrow average(Return(s))$
  end
end

Monte Carlo methods: Action Value approximation

Procedure: Monte Carlo (policy improving)

Input: $\pi \leftarrow$ arbitrary policy
Input: $Q(s, a) \leftarrow$ arbitrary function
Input: $Returns(s) \leftarrow$ empty list for all $s \in \mathcal{S}$
while *true* **do**
  Generate an episode from π
  foreach *state s in the episode* **do**
    $R \leftarrow$ *return* obtained from the first occurrence of s
    add R to $Returns(s)$
    $Q(s, a) \leftarrow average(Return(s))$
  end
  foreach *state s in the episode* **do**
    $\pi(s) \leftarrow \text{argmax}_a Q(s, a)$
  end
end

Temporal Difference Learning

- Monte Carlo methods assume that we obtain the reward at the end of an episode
- Temporal Difference Learning uses the return obtained from the immediate states
- The simplest Temporal Difference Learning is TD(0) that uses the estimate

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

- This methods do not need a model of the task (unlike DP)
- This methods can be implemented on-line, do not need to end an episode to obtain a reward (unlike MC)

Temporal Difference TD(0): State Value approximation

Procedure: Temporal Difference TD(0) (policy evaluation)

Input: $\pi \leftarrow$ policy to be evaluated
Input: $V(s) \leftarrow$ arbitrary function
foreach *episode* **do**
  $s \leftarrow$first state of the episode
  foreach *step in the episode* **do**
    $a \leftarrow$ action given by π for s
    Take action a
    Observe reward r and next state s'
    $V(s) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$
    $s \leftarrow s'$
  end
end

 cbna

Q-learning

- Q-learning is a type of Time Difference Learning that does not need a model of the task
- The goal is to approximate the action-value function (Q) by observing the rewards obtained from episodes
- Each step the immediate reward is used to update the function Q

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

- The learned policy will be the one that always picks the action that maximizes the reward from a state

Q-learning: Action Value approximation

Procedure: Q-learning (policy improving)

Input: $Q(s,a) \leftarrow$ arbitrary function

foreach *episode* **do**

  $s \leftarrow$first state of the episode

  foreach *step in the episode* **do**

    Choose $a \leftarrow$ from the policy derived from Q (greedy, ϵ-greedy)

    Take action a

    Observe reward r and next state s'

    $Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$

    $s \leftarrow s'$

  end

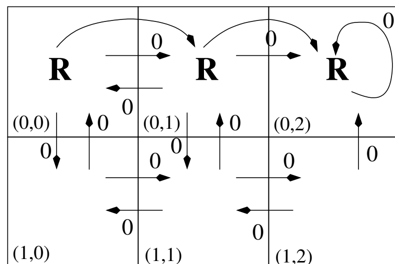
end

The training process

- To learn the agent has to be trained in the environment
- The idea is to let the agent play in the environment (episodes) for a period of time (usually plenty) until the function Q converges
- After certain number of steps Q converges to Q^* if:
  - The problem can be modeled as as MDP
  - The reinforcement is bounded by a value (c)
  - Each pair (state, action) is visited an infinite number of times

Example (1st Training)

- We will consider this scenario

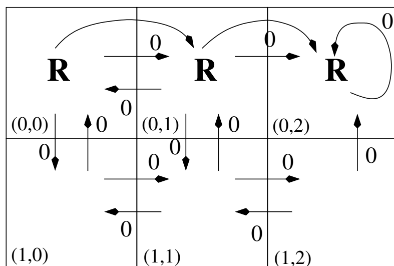


- We will assume that $\alpha = 1$, so the update expression simplifies to:

$$Q(s_t, a_t) = r_{t+1} + \gamma \max_a Q(s_{t+1}, a)$$

- We will use $\gamma = 0.9$

Example (1st episode)



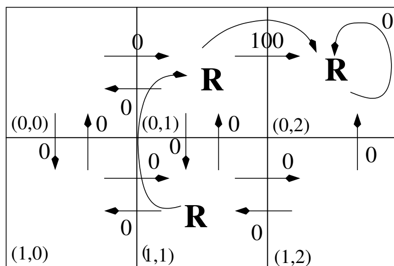
1. $s = (0,0)$, action$=a_{right}$, $s' = (0,1)$, $r = 0$, $\gamma = 0.9$

$$Q((0,0), a_{right}) = r((0,0), a_{right}) + 0.9 \cdot \max\{Q((0,1), a_{right}), Q((0,1), a_{left}), Q((0,1), a_{down})\} = 0 + 0.9 \cdot \max\{0,0,0\} = 0$$

2. $s = (0,1)$, action$=a_{right}$, $s' = (0,2)$, $r = 100$, $\gamma = 0.9$

$$Q((0,1), a_{right}) = 100 + 0.9 \cdot \max\{0\} = 100$$

Example (2nd episode)



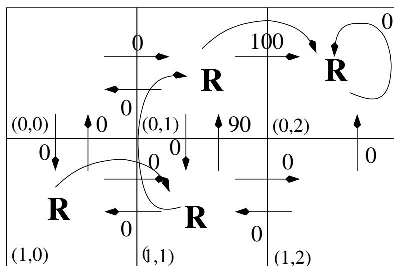
1. $s = (1,1)$, action$=a_{up}$, $s' = (0,1)$, $r = 0$, $\gamma = 0.9$

$$Q((1,1), a_{up}) = 0 + 0.9 \cdot \max\{0, 0, 100\} = 90$$

2. $s = (0,1)$, action$=a_{right}$, $s' = (0,2)$, $r = 100$, $\gamma = 0.9$

$$Q((0,1), a_{right}) = 100 + 0.9 \cdot \max\{0\} = 100$$

Example (3rd episode)



1. $s = (1,0)$, action$=a_{right}$, $s' = (1,1)$, $r = 0$, $\gamma = 0.9$

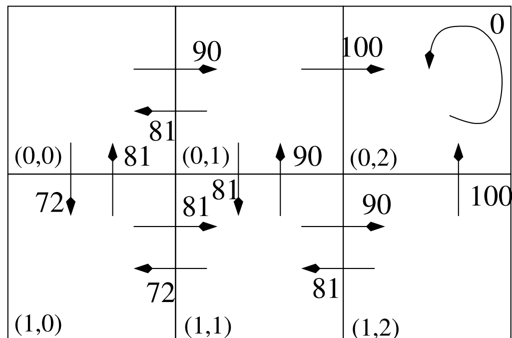
$$Q((1,0), a_{right}) = 0 + 0.9 \cdot \max\{0,0,90\} = 81$$

2. $s = (1,1)$, action$=a_{up}$, $s' = (0,1)$, $r = 0$, $\gamma = 0.9$

$$Q((1,1), a_{up}) = 0 + 0.9 \cdot \max\{0,0,100\} = 90$$

3. ...

Example - After convergence



RL in continuous domains

- We have considered only discrete domains, RL can be extended to continuous domains
- We have to obtain a parameterized continuous function of the description of the state that approximates the value functions
- If the state s can be describe by features $s = \{s_1, s_2, \ldots, s_n\}$, assuming we are using a linear function

$$V_\theta(s) = \theta_0 + \theta_1 s_1 + \theta_0 s_2 + \cdots + \theta_0 s_n$$

- The RL algorithm has to adjust the parameters of the function so the value functions approximate the observed rewards
- For continuous problems, every time we change the function we change the value for all states

RL in continuous domains

- The simplest approach is to derive an on-line rule that can be used to update the parameters
- The usual way is to define an error function (how wrong the function is for an state respect the observed value)
- From the error function a gradient descent update can be computed (just like in neural networks)
- For example, the update rule of the parameters for the state value function for TD:

$$\theta_i = \theta_i + \alpha[r_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)]\frac{\partial V_\theta(s_t)}{\theta_i}$$

- Depending of the function used (linear, non-linear) the computation of the update can be expensive