Reinforcement Learning

—

Waïl BENFATMA (wail.benfatma@gmail.com)

Menu

1. Introduction to Reinforcement Learning
2. Markov Decision Process
3. Model-Free explanation
4. Reinforcement Learning in practice

Objectives

Get knowledge on RL problems and applications domains.

Organisation

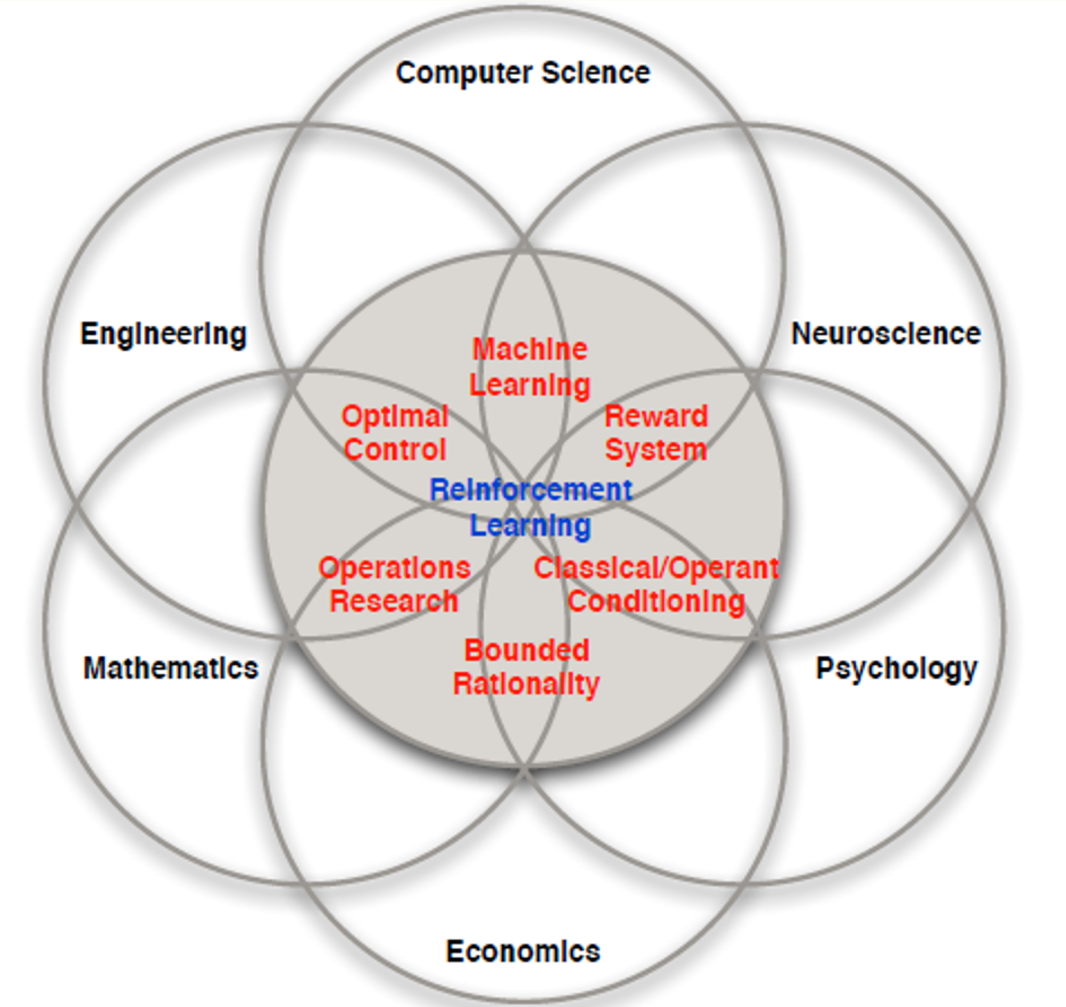
- 5 classes & 2 courseworks
- Appreciation of courseworks

Participate !

TextBook / Mooc

- An Introduction to Reinforcement Learning, Sutton and Barto, 1998 http://webdocs.cs.ualberta.ca/sutton/book/the-book.html
- Practical Reinforcement Learning Coursera
- Course on Reinforcement Learning
- David Silver - RL course

Introduction


Computer Science
Engineering
Neuroscience
Machine Learning
Optimal Control
Reward System
Reinforcement Learning
Operations Research
Classical/Operant Conditioning
Bounded Rationality
Mathematics
Psychology
Economics

Reinforcement Learning

Some problems cannot be solved by classical machine learning or optimisation algorithms

It’s not because of insufficient data or powerless computation capacity.
It’s because in some case it’s difficult to find the right action to do

Reinforcement Learning

RL is a modelisation that <u>TRY, FAIL, LEARN</u> the best way to solve a problem.

RL combines many science to solve these problems

Characteristics of RL

Particularities of RL compare to classic ML algorithms

- There is an interaction with the environment and it gets only a signal reward
- Feedback is delayed, can be long time after
- Time is important
- Agent's action affect the next steps and so the data it receives

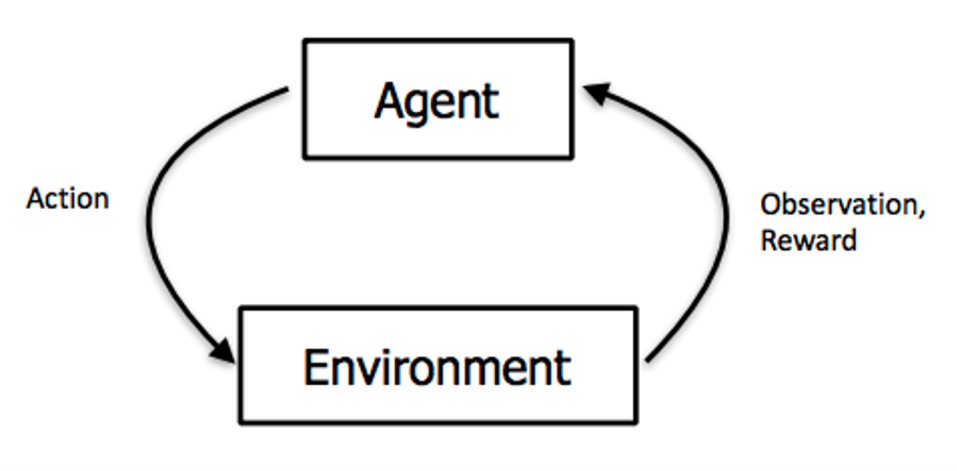
Examples of RL application

- Defeat world champion at Go Game
- Make a humanoid robot walk

  https://www.youtube.com/watch?v=n2gE7n11h1Y
- Play ATARI games
- Drive an autonomous car

How RL works

Learning by reward



Determine itself the ideal behavior according to a context to maximize its performance.

Reward

RL is based on the reward hypothesis i.e
All goals can be described by the maximisation of expected cumulative reward

- The reward R_t is a feedback signal and gives an indication on how well agent is doing at step t.

  The agent's goal is to maximize cumulative reward

Examples of RL rewards

- Defeat world champion at Go Game : +/- points for winning/losing a game
- Make a humanoid robot walk : + points for positive motion, - if it fails
- Play ATARI games : + points for increasing score
- Drive an autonomous car : + points for motion without crash

Action

Corresponds to how the agent interacts with the environment.
An action will most often change the state of the system for a new state that will have an associated reward.

- Easier if you have a limited and constant number of actions.

- Difficult to modelize continuous actions.

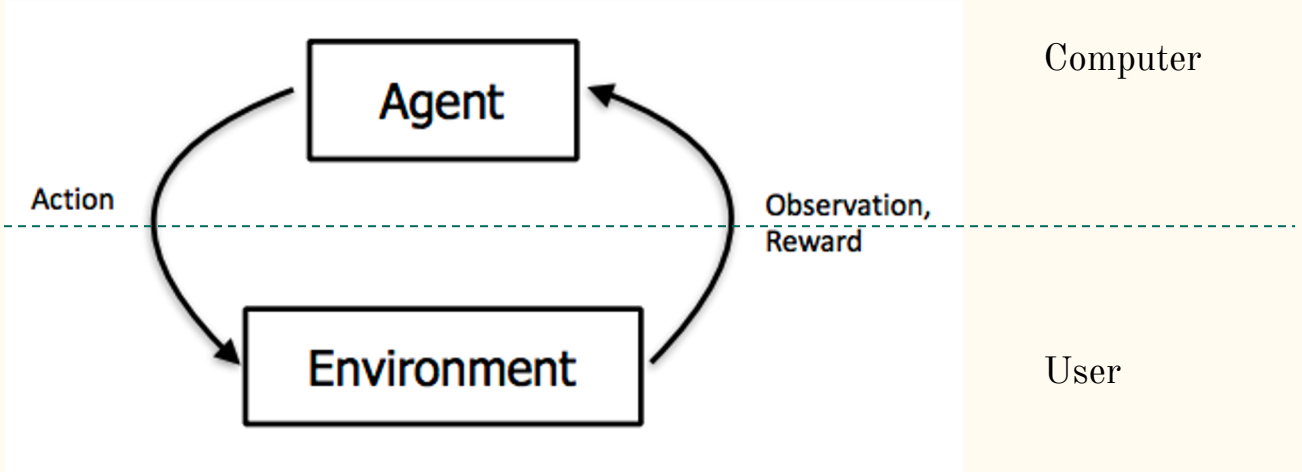
Sequential Decision Making

Select a succession of actions to maximize total future reward :

- Actions may have a long term consequences in + or - way
- Reward may be delayed
- It way be better to sacrifice immediate reward to gain more long-term reward (ex : Go or Chess)

Interactions in RL : banner example

Agent and Environment



At each step the agent execute an action, gets the reward and the new env's observation.
The Env receives the action and change according to it. Its gives the reward.

State and History

- The history is the sequence of actions, observations and rewards upon time

$$H_t = \{(A_1, O_1, R_1), \dots , (A_t, O_t, R_t)\}$$

- What happens next depends on the history : Agent selects action and Env selects next state and associated reward.

- State is the information used to determine what happens next.

- State is a function of the history :

$$S_t = F(H_t)$$

Environment state vs. Agent state

The environment state $S_{e,t}$ is the environment's internal representation

- It's used to select the next state/reward.
- It is not usually visible to the agent.
- It may contain irrelevant information for the agent.

The agent state $S_{a,t}$ is the agent's private representation

- It's used to select the next agent.
- It is the information used by RL algorithms.
- It can be a function of the history (modelisation)

Fully vs. Partially observable environments

Fully observable environment is when the agent directly observes the env state

- It's a Markov Decision Process (MDP)
- Majority of study cases

Partially observable environment is when the agent indirectly observes the env state

- More difficult
- Ex : autonomous car with camera vision, trading, poker
- Agent must construct its own state representation with complete history

Information State

<u>Definition :</u> An information state contains all useful information from the history. It is called as a Markov state, i.e.

$$P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_1, \dots, S_t]$$

Once the state is known, the history is useless.

The environment state is always Markov by definition

It's a MARKOV DECISION PROCESS (MDP)

Major components of an RL agent

An RL agent may include one or more of these components :

- Policy : agent's behaviour function.
- Value function : how good is each state and / or action.
- Model : agent's representation of the environment

RL : Policy

The policy is the mapping from state to action

- It is the agent's behavior for each state.
- Usually it is deterministic
- But could also be stochastic when an action is taken according to a probability.

The optimal policy is the mapping which maximize the sum of rewards you get from an initial state to the end.

RL : Value Function

The value function is an estimation / prediction of the future reward

- It is used to evaluate the goodness/badness of states.
- It is used to select between actions.

$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s \right]$$

π is the policy and γ is the discount in [0;1]

RL : Model

The <u>model</u> will predict what the environment will do next

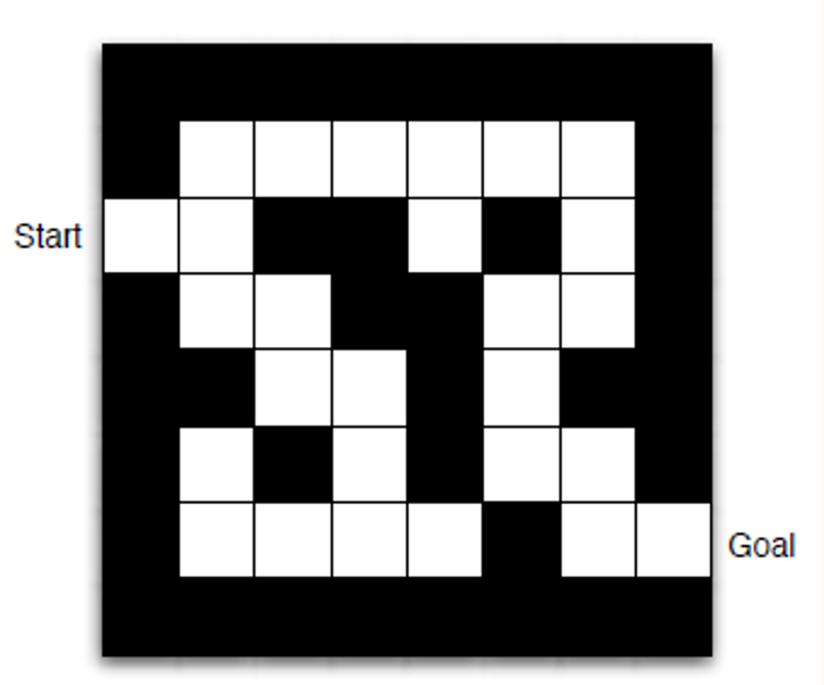
- P predicts the next state.
- R predict the next immediate reward.

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$$
$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

RL : Example

Maze game

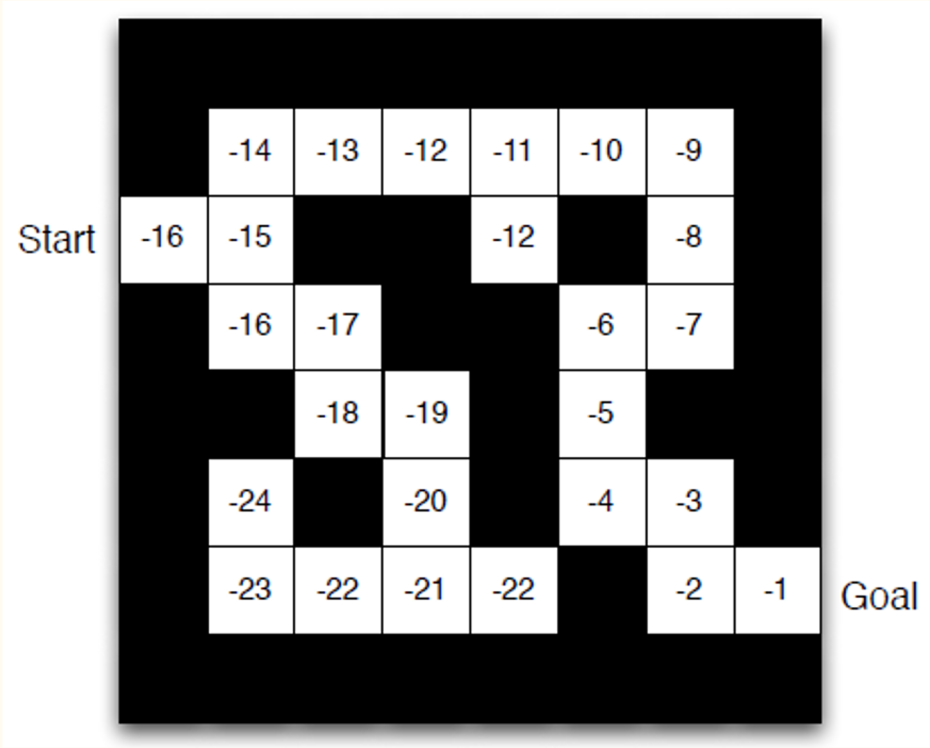
- Rewards : -1 per time-step
- Actions : N, W, E, S
- States : Agent's location



RL : Example

Value Function

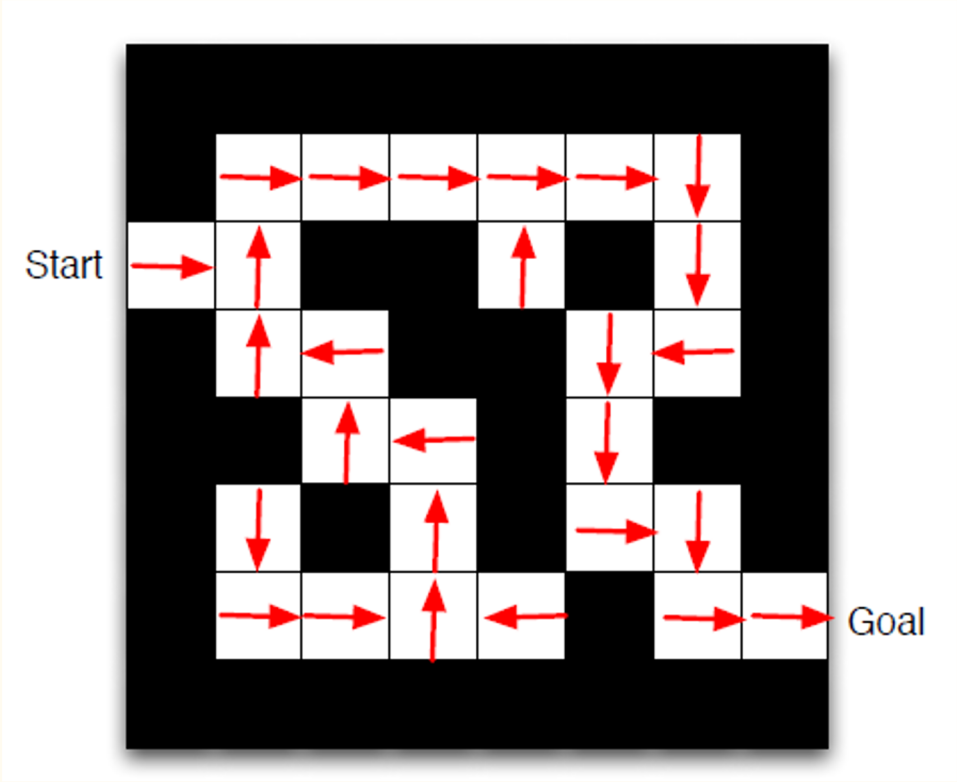
Represent the expected final reward from a location.



RL : Example

Policy

For each step, the best action to take to maximize the total reward.



Learning vs. Planning

Learning

- The environment is usually unknown.
- The agent interacts with the environment.
- The agent improves its policy.

Planning

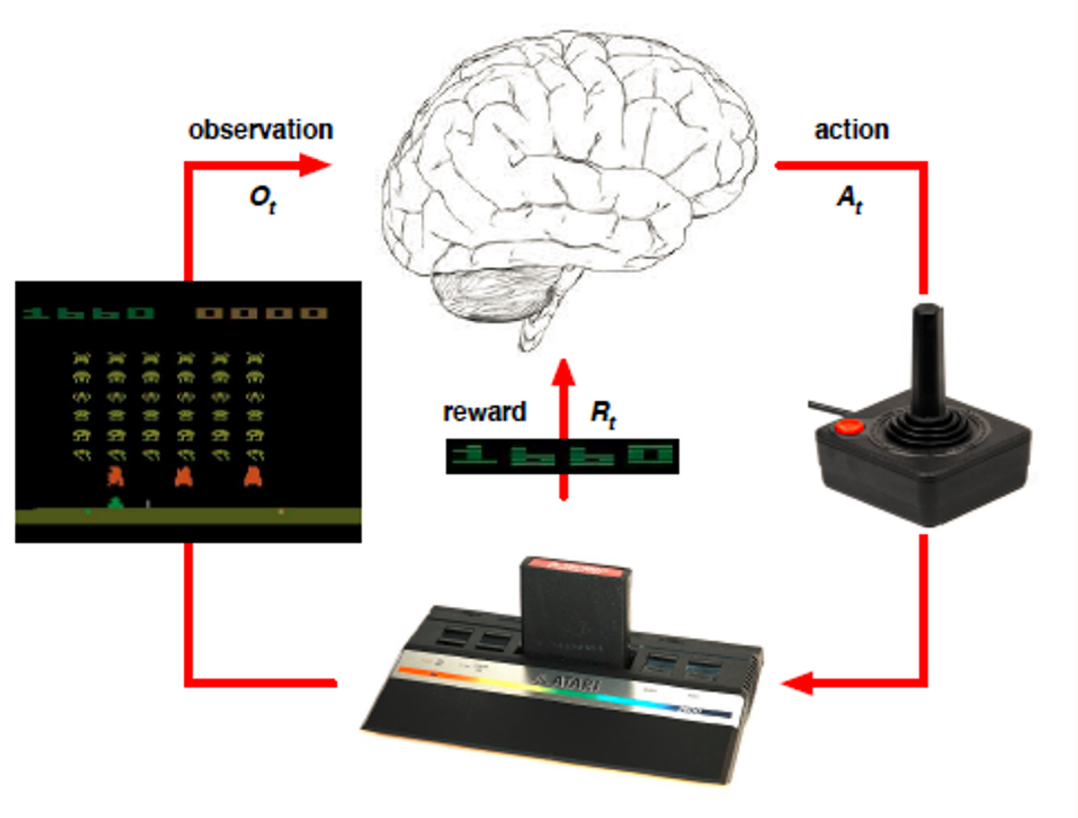
- A model of the environment is known.
- The agent performs computation with a predefined model (without external interaction).
- The agent can improve its policy.

RL vs. Planning : ATARI example

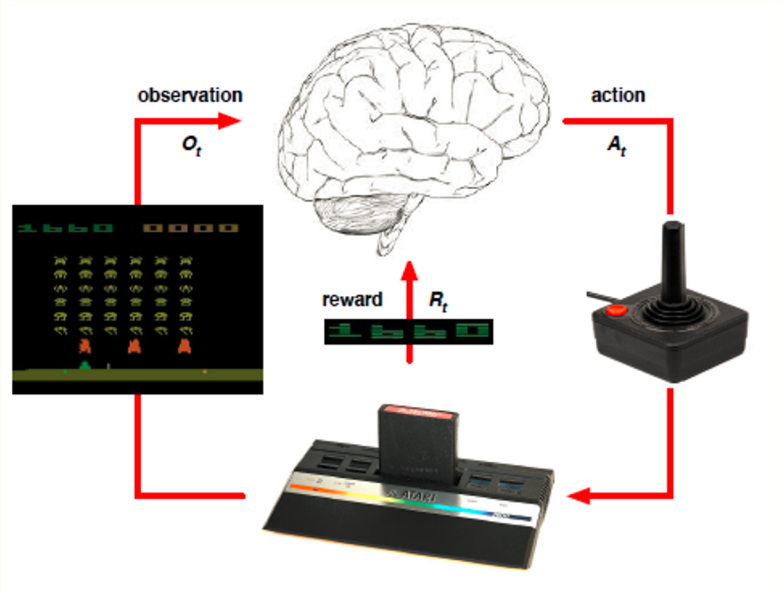
Define actions, env, rewards, ...

Difference between RL and Planning

RL vs. Planning : ATARI example



RL vs. Planning : ATARI example



Reinforcement Learning

- The environment is unknown.
- The rules are unknown.
- Learn from interactive play.

Planning

- The rules are known.
- Sequences of “perfect” actions defined in the model.

RL : Exploitation vs. Exploration

Exploitation

- Get more information about known environment.
- Update your value function to better evaluate the future

Exploration

- Discover the environment
- Try to find the best policy

Markov Decision Process

Markov Decision Process

Markov decision processes formally describe an environment for reinforcement learning

Where the environment is fully observable

i.e. The current state completely characterises the process

Almost all RL problems can be formalised as MDPs

MDP properties

A state is Markov is :

$$\mathbf{P}\,[S_{t+1} \mid S_t] = \mathbf{P}\,[S_{t+1} \mid S_1, \ldots, S_t]$$

- The state captures all relevant information from the history
- Once the state is known, the history may be thrown away
- The state is a sufficient statistic of the future

Markov Process

A Markov process is a sequence of states with the Markov property.

For a Markov state s and successor state s′, the state transition probability is defined by

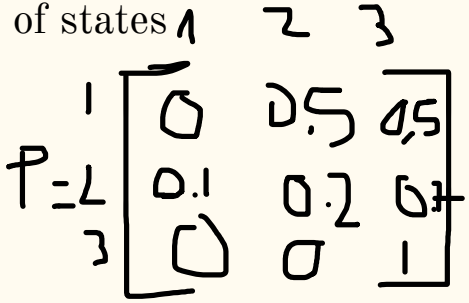
$$P_{ss'} = P[S_{t+1} = s' | S_t = s]$$

State transition matrix P defines transition probabilities from all

states s to all successor states s′,

A Markov Process (or Markov Chain) is a tuple ⟨S,P⟩ S is a (finite) set of states

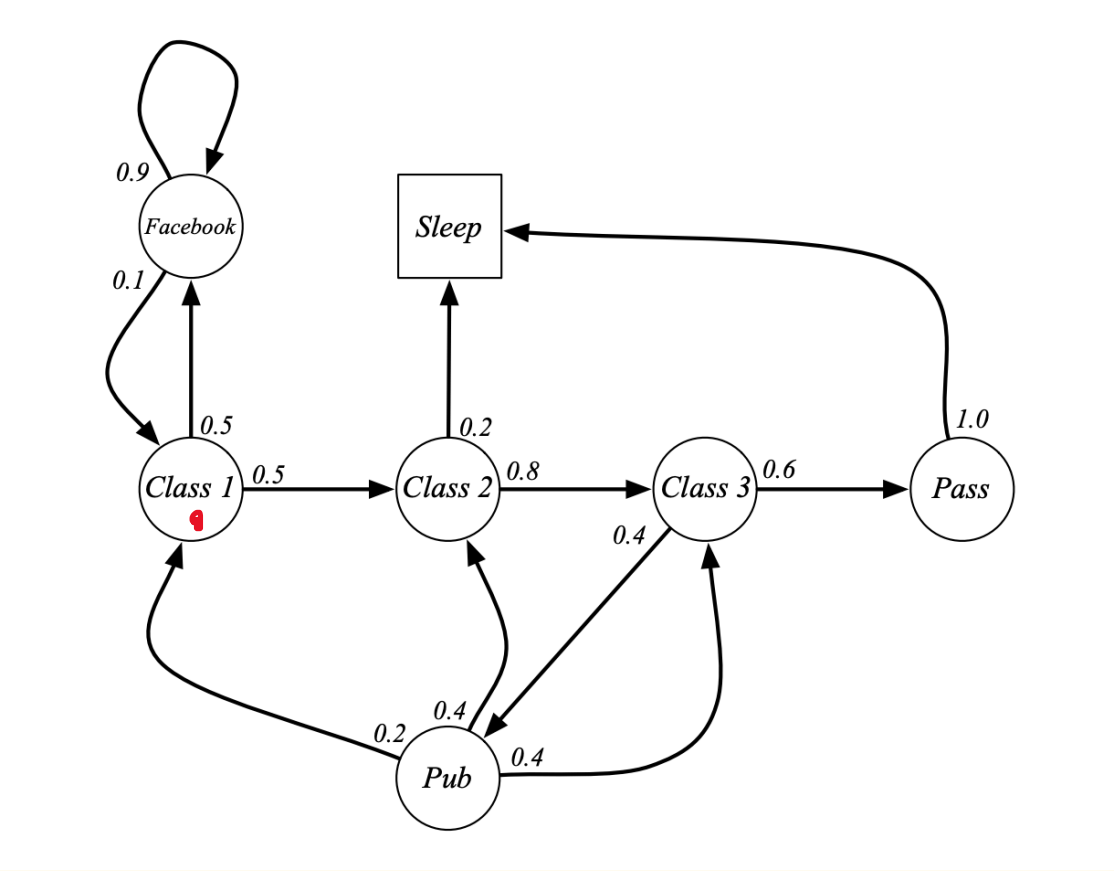
P is a state transition probability matrix, $P_{ss'} = P[S_{t+1} = s' | S_t = s]$



Markov Process

Example :

- play
- Transition matrix

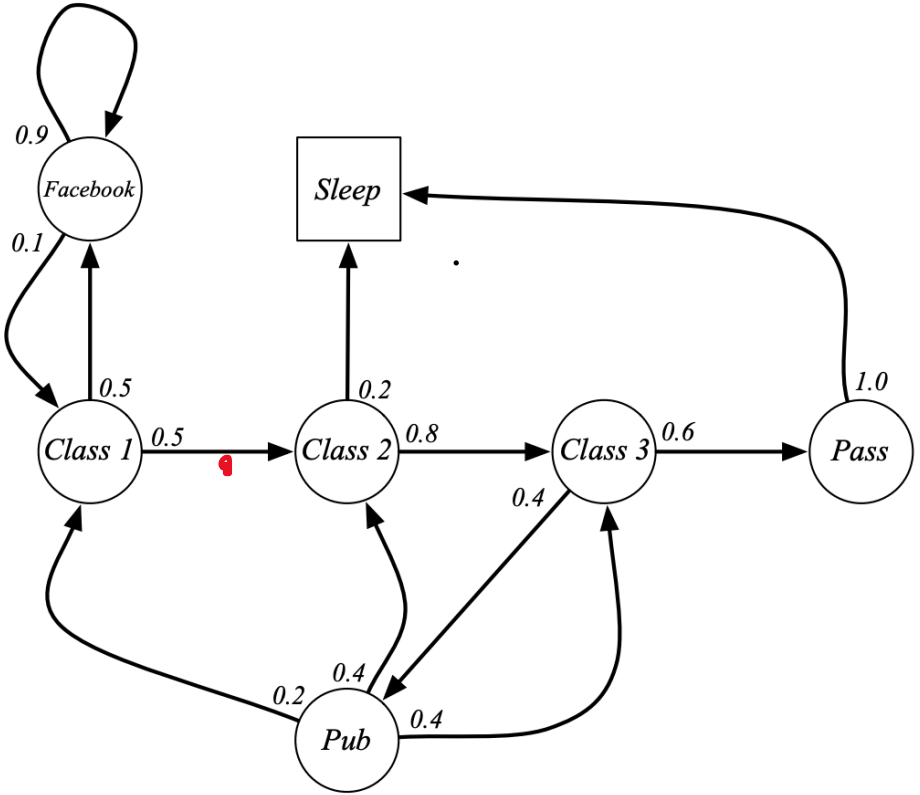


Markov Process

- C1 - > C2 -> C3 -> Pass -> S
- C1 -> FB -> FB -> FB -> C1 -> C2 -> S
- C1→C2 -> C3 -> Pub -> C2 -> C3 -> S

Example :

- play
- Transition matrix



	C1	FB.	C2.	C3.	Pub.	Pass.	Sleep
C1		0.5	0.5				
FB.	0.1	0.9					
C2.				0.8			0.2
C3					0.4.	0.6	
Pub	0.2.		0.4.		0.4		
Pass							1
Sleep							1

Reward and return

The reward function is the expected reward function at each state.

$$G_t = R_{t+1} + R_{t+2} + ...$$

We introduced the discount factor γ between 0 and 1 which correspond to the present value of future rewards. Close to 1 we look ahead.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$$

It avoids infinite loops, let immediate rewards be more important than later ones and simulate the uncertainty of the future.

Markov Reward Process

A Markov reward process is a Markov chain with reward values.

A Markov Reward Process is a tuple $\langle S,P,R,\gamma \rangle$ with

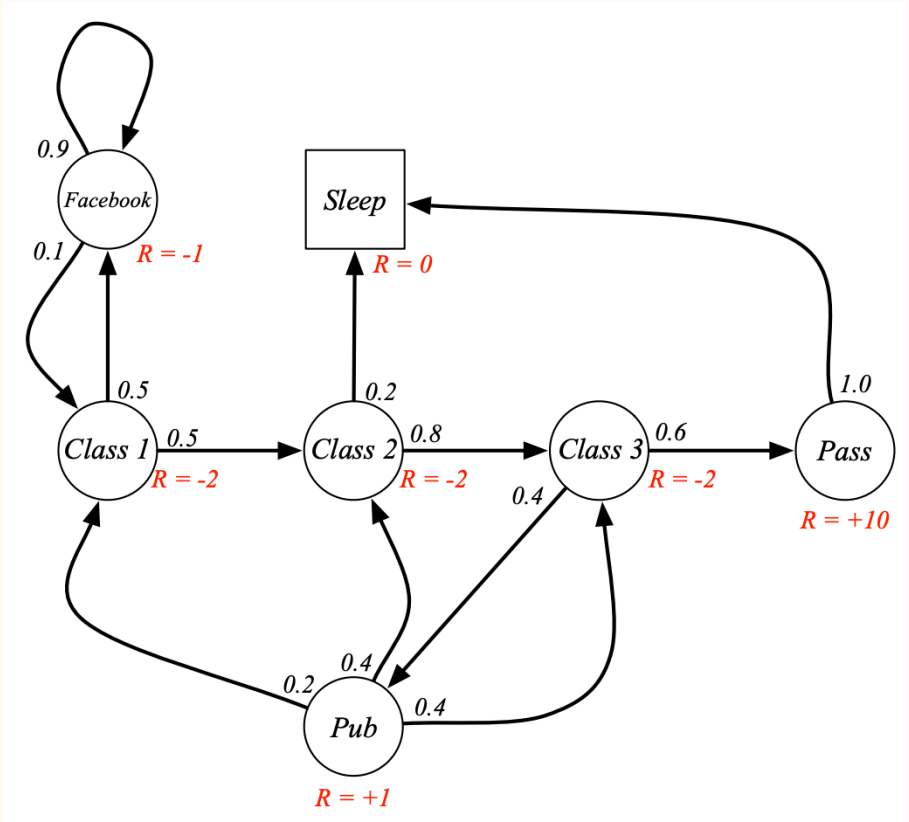
- S is a (finite) set of states
- P is a state transition probability matrix, $Pss' = P[S_{t+1}=s' | S_t=s]$
- R is the reward function $R_S = E\,[G_t \mid S_t=s]$
- γ is the discount factor

Markov Reward Process

- C1 - > C2 -> C3 -> Pass -> S
- C1 -> FB -> FB -> FB -> C1 -> C2 -> S
- C1→C2 -> C3 -> Pub -> C2 -> C3 -> S

Example :

- play
- Reward



Value function

Gives the long term value of a state.

$$v(s) = E[Gt \mid St = s]$$

The value function can be decomposed into two parts:

- immediate reward Rt+1
- discounted value of successor state

Value function

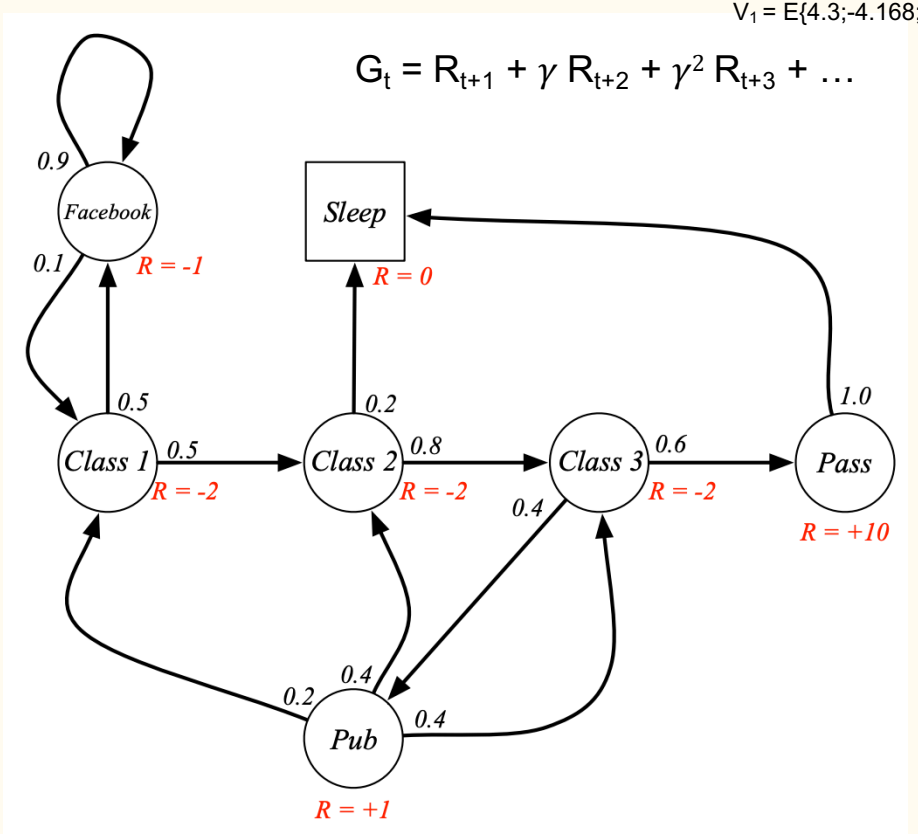
- C1 - > C2 -> C3 -> Pass -> S
- C1 -> FB -> FB -> FB -> C1 -> C2 -> S
- C1→C2 -> C3 -> Pub -> C2 -> C3 -> Pass -> S

$G_1 = -2 + (-2)*(0.9) + 10*(0.9)^2 + 0*(0.9)^3 = 4.3$
$G_1 = -1 + (-1)*(0.9) - 1*(0.9)^2 - 2*(0.9)^3 - 0*(0.9)^4 = -4.168$
$G_1 = -2 + (-2)*(0.9) + 1*(0.9)^2 - 2*(0.9)^3 - 2*(0.9)^4 + 10*(0.9)^5$
$= \ldots.$

$V_1 = E\{4.3; -4.168; \ldots\} =$

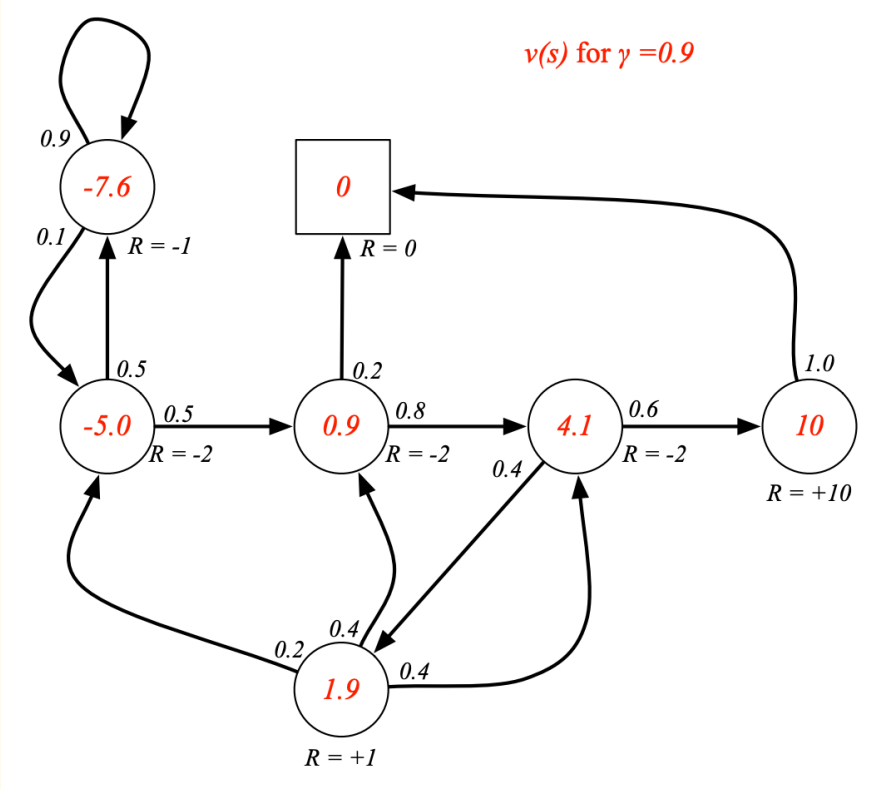
Example :

- play
- Value Function
  for y = 0
  for y = 0.9



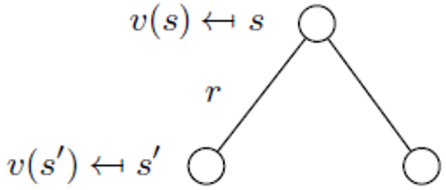
Value function

Example :



Bellman equation for MRP

$$v(s) = \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right]$$



$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

A step ahead evaluation of value function is easily verified

Bellman equation – matrix form

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{11} & \cdots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

$$\begin{aligned} v &= \mathcal{R} + \gamma \mathcal{P} v \\ (I - \gamma \mathcal{P})\, v &= \mathcal{R} \\ v &= (I - \gamma \mathcal{P})^{-1} \mathcal{R} \end{aligned}$$

The Bellman equation is a linear equation and can be solved directly for small problems or with iterative process such as Dynamic Programming / Monte-Carlo evaluation / Temporal Difference

Markov Decision Process

A Markov Decision Process (MDP) is a Markov Reward Process with decisions and Markov states.

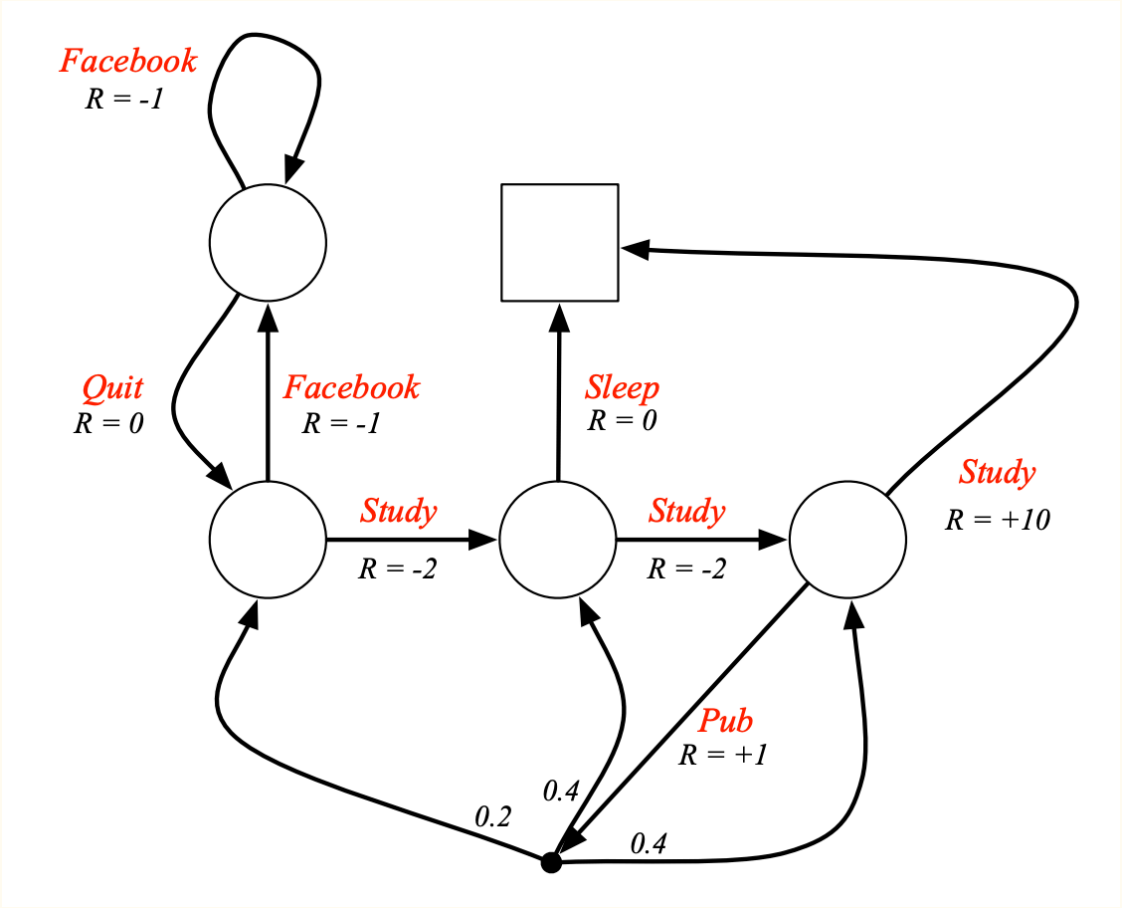
To previous definition to MRP we add a finite set of Actions A.

- The state transition probability matrix depends of action
  $$P^{a}_{ss'} = P\,[S_{t+1} = s' \mid S_t = s,\, A_t = a]$$
- The reward function depends of action
  $$R^{a}_{s} = E\,[R_{t+1} \mid S_t = s,\, A_t = a]$$

A Markov Decision Process is a tuple $\langle S, A, P, R, \gamma \rangle$ with A the set of action

MDP

Example :


Facebook
R = -1
Quit
R = 0
Facebook
R = -1
Sleep
R = 0
Study
R = -2
Study
R = -2
Study
R = +10
Pub
R = +1
0.2
0.4
0.4

Value functions

The state value function Vp(s) is the expected return starting from state s and following policy p.

$$v_\pi(s) = E_\pi [G_t \mid S_t = s]$$

$$\begin{aligned} v_\pi(s) &= E_\pi [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid S_t = s] \\ &= E_\pi [R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + ...) \mid S_t = s] \\ &= E_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \end{aligned}$$

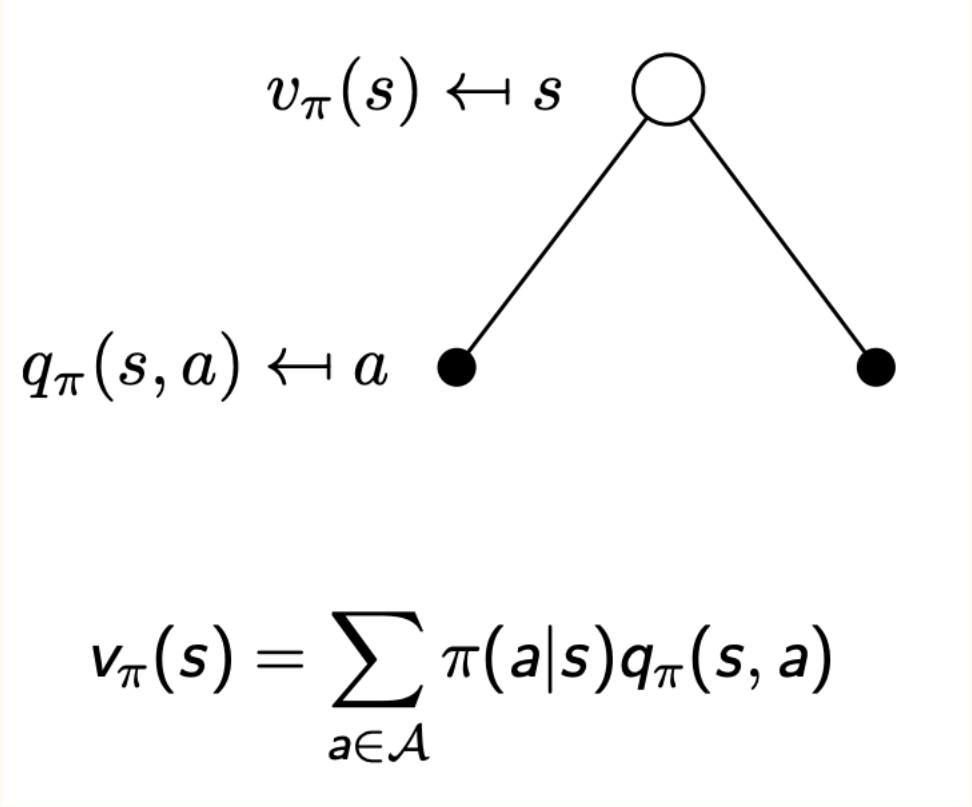
The action value function Qp(s,a) is the expected return starting from s, taking action a and following policy p.

$$q_\pi(s, a) = E_\pi [G_t \mid S_t = s, A_t = a]$$

-> bellman equations associated :

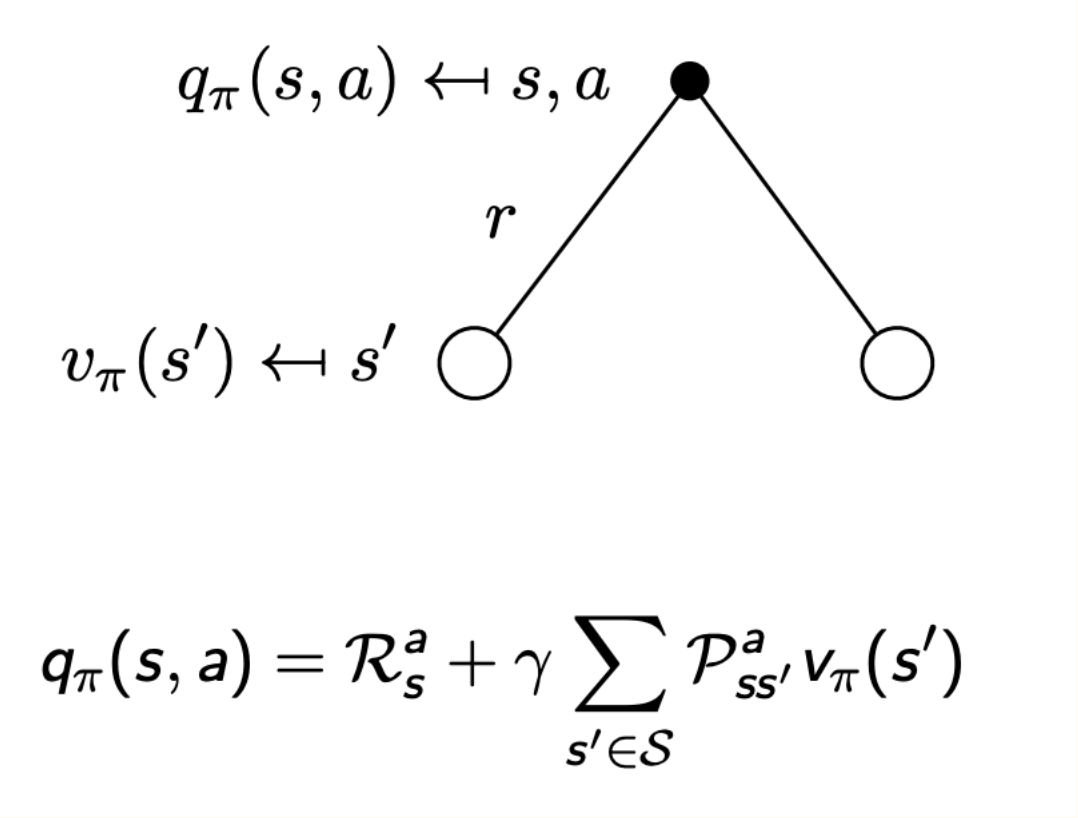
$$v_\pi(s) = E_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \quad \& \quad q_\pi(s,a) = E_\pi [R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

Bellman equation for value function



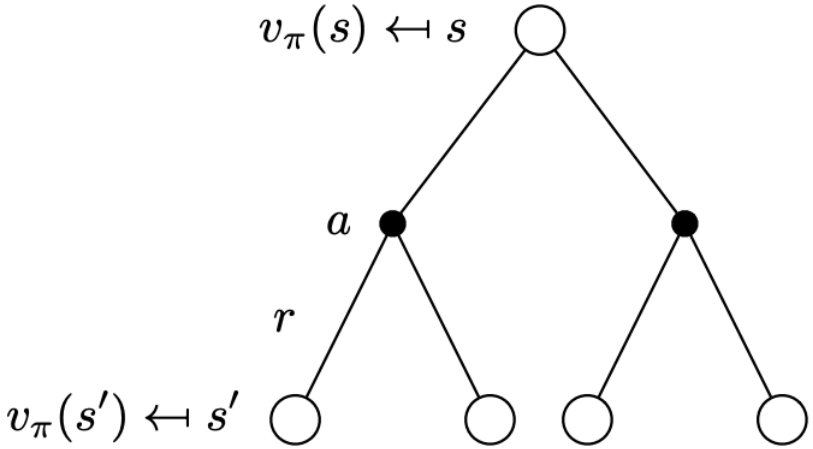
$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a)$$

Bellman equation for action value function



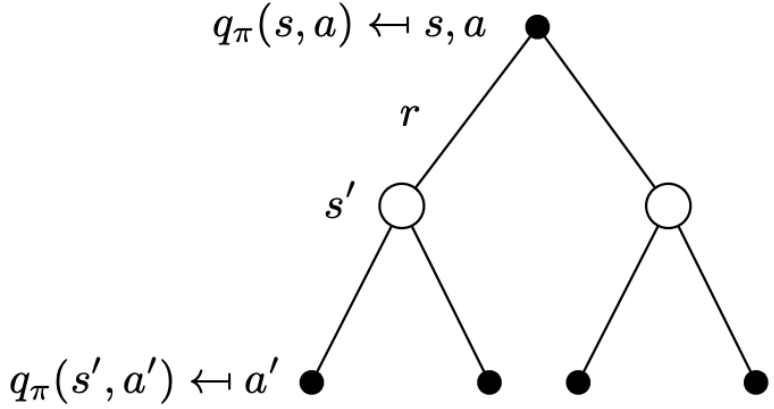
$$q_\pi(s,a) = \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} v_\pi(s')$$

Bellman equation for value function



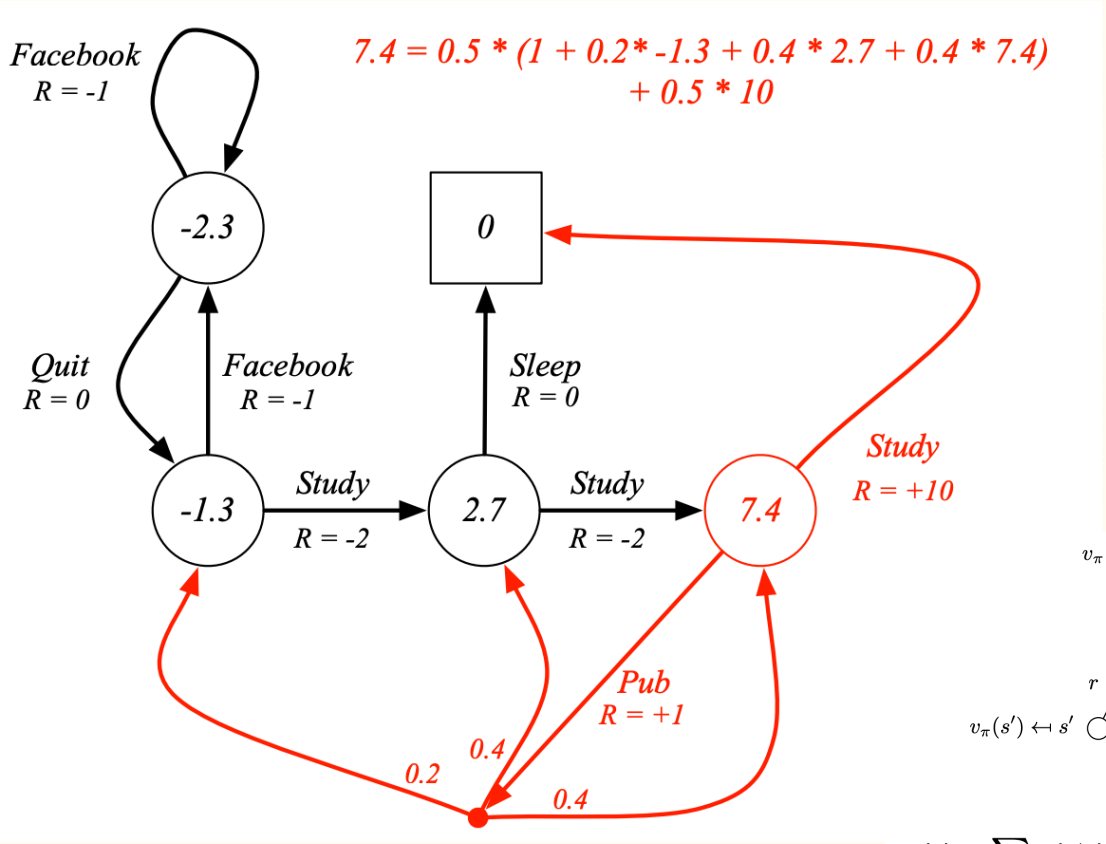
$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

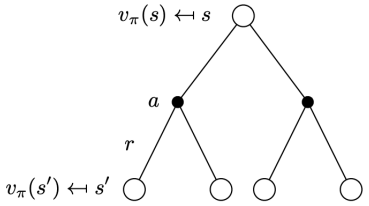
Bellman equation for action value function



$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$$

Example :





$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

Policies

A policy is a set of actions given states.

It defines the behaviour of the agent throughout states.

Policies depend on current state, memoryless of the past.

Policies are independent of the time.

Optimal Value function

The optimal state-value function v∗(s) is the maximum value function over all policies

$\mathbf{v^*(s) = \max v_\pi(s)}$ over $\boldsymbol{\pi}$

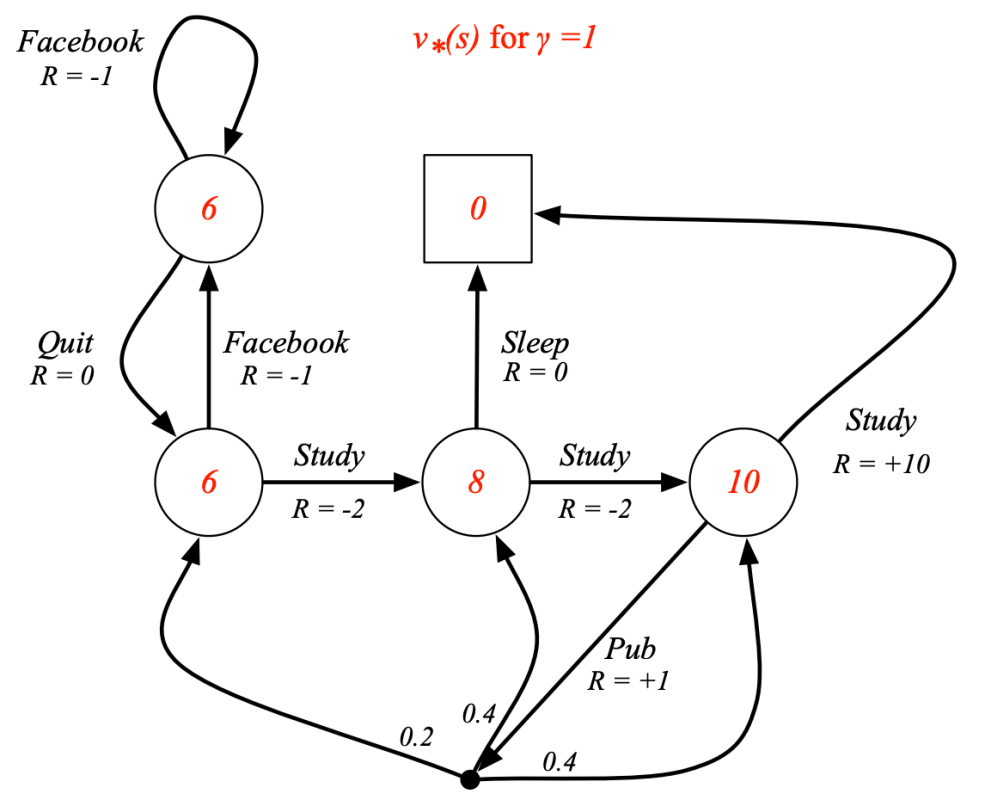
The optimal action-value function q∗(s,a) is the maximum action-value function over all policies

$\mathbf{q^*(s, a) = \max q_\pi(s, a)}$ over $\boldsymbol{\pi}$

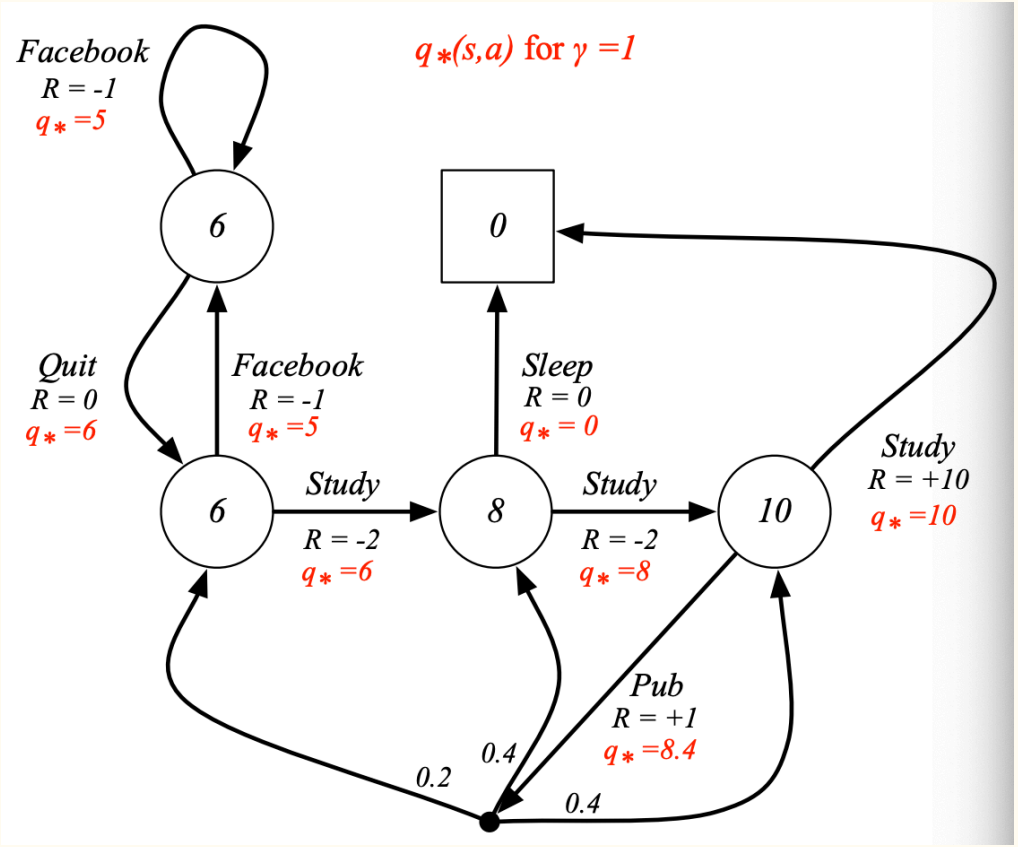
The optimal value function specifies the best possible performance in the MDP.

An MDP is “solved” when we know the optimal value function.

Example : Optimal Value Function



Example : Optimal Action Value Function



Optimal Policy

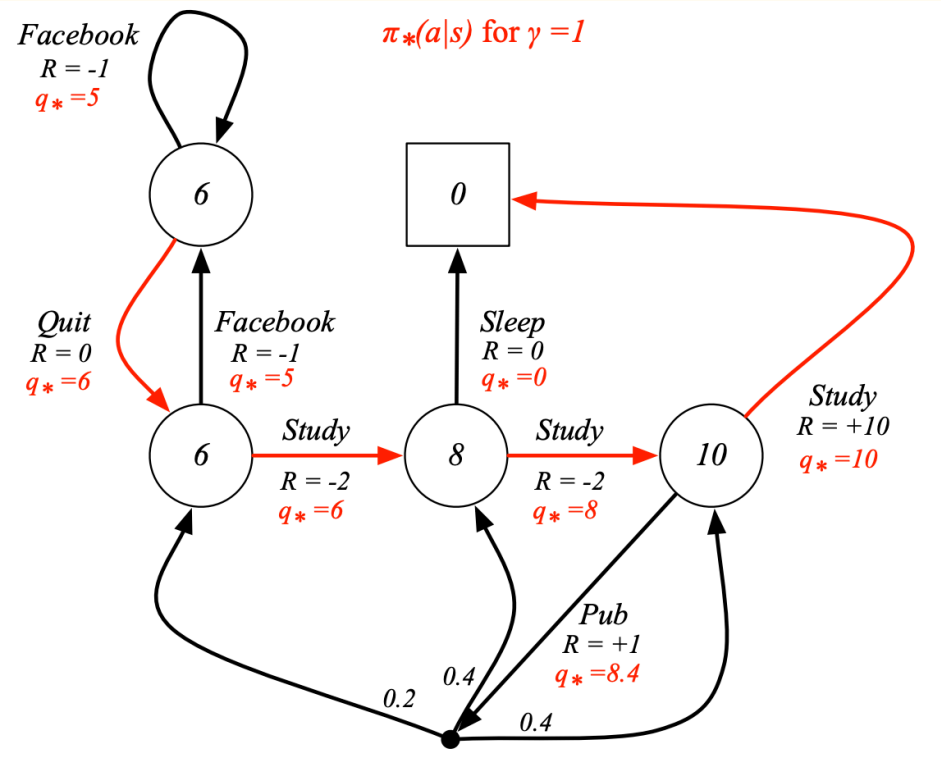
Define a partial ordering over policies

$\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s), \forall s$

For any Markov Decision Process

- There exists an optimal policy $\pi*$ that is better than or equal to all other policies, $\pi* \geq \pi$, over π
- All optimal policies achieve the optimal value function $v_\pi^*(s) = v^*(s)$
- All optimal policies achieve the optimal action-value function, $q_\pi^*(s, a) = q^*(s, a)$

Example : Optimal Policy



Finding an Optimal Policy

An optimal policy can be found by maximising over q∗(s,a),

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\text{argmax}}\, q_*(s,a) \\ 0 & \text{otherwise} \end{cases}$$

There is always a deterministic optimal policy for any MDP. If we know q∗(s,a), we immediately have the optimal policy

->Bellman equation

Bellman Optimality Equation is non-linear :
- No closed form solution (in general)
- Many iterative solution methods : Value Iteration, Policy Iteration, Q-learning

Dynamic Programming

Refresh :

Dynamic programming can solve optimization problems by combining solutions of sub-problems.

This type of algorithm "divide to better rule" stock the sub-problem values to avoid recalculations.

The sub-trajectory of a optimal trajectory is still optimal.

DP : Prediction & control

Dynamic programming :

- Needs full knowledge of the MDP

- Can be use for prediction by determining value function :
    - from a MDP
    - with a policy

- Can be use for control by determining the optimal policy :
    - from a MDP
    - by computing optimal value function

Policy evaluation

To evaluate a given policy we will apply iterative Bellman expectation backup using synchronous backups :

- At each iteration k + 1
- For all states s in S
- Update $v_{k+1}(s)$ from $v_k(s')$
  where s' is a successor state of s

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

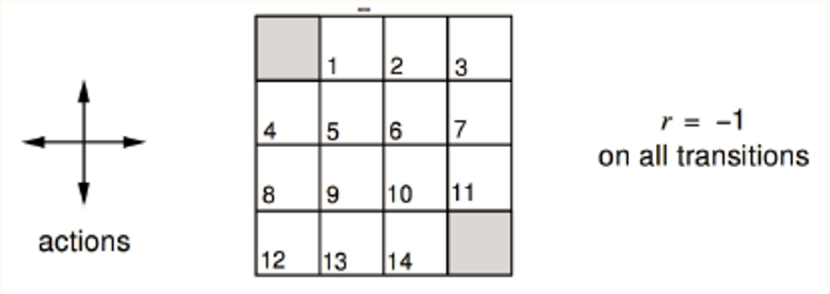
Policy evaluation

Example

Gridworld where agent starts from a state and try to find the end state.

Reward is -1 unless the terminal state is reached and agent can't get off the grid.

Agent follow random policy.



Policy evaluation

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

v_k for the Random Policy

$k = 0$

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

V(s) = (1/4) * (-1 + γ*(1*0)) (gauche)
+ (1/4) * (-1 + γ*(1*0)) (droite)
+ (1/4) * (-1 + γ*(1*0)) (haut)
+ (1/4) * (-1 + γ*(1*0)) (bas)

Policy evaluation

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

v_k for the Random Policy

$k = 0$

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

$k = 1$

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

V(s) = (1/4) * (-1 + 1*(1*(-1))) (gauche)
+ (1/4) * (-1 + 1*(1*(-1))) (droite)
+ (1/4) * (-1 + 1*(1*(-1))) (haut)
+ (1/4) * (-1 + 1*(1*0)) (bas)
= -1/2 + -1/2 + -1/2 -1/4 = -1,75

Avec gamma = 1

Policy evaluation

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

v_k for the Random Policy

$k = 0$

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

$k = 1$

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

$k = 2$

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

Policy evaluation

v_k for the Random Policy

$k = 0$

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

$k = 1$

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

$k = 2$

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

$k = 3$

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

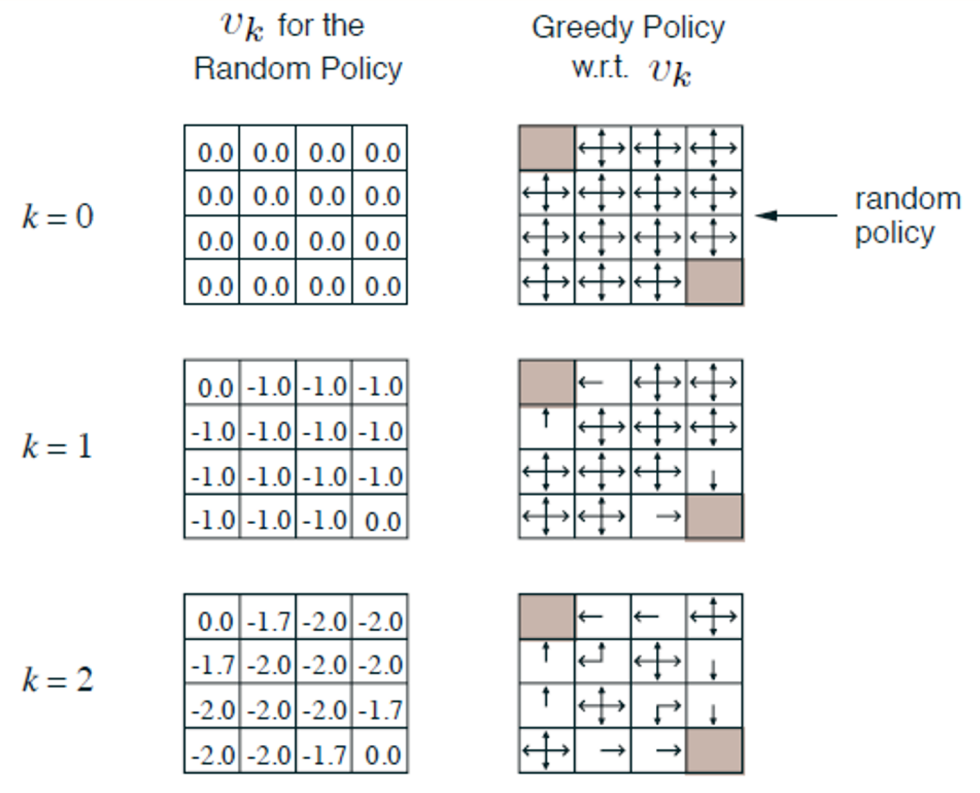
$k = 10$

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

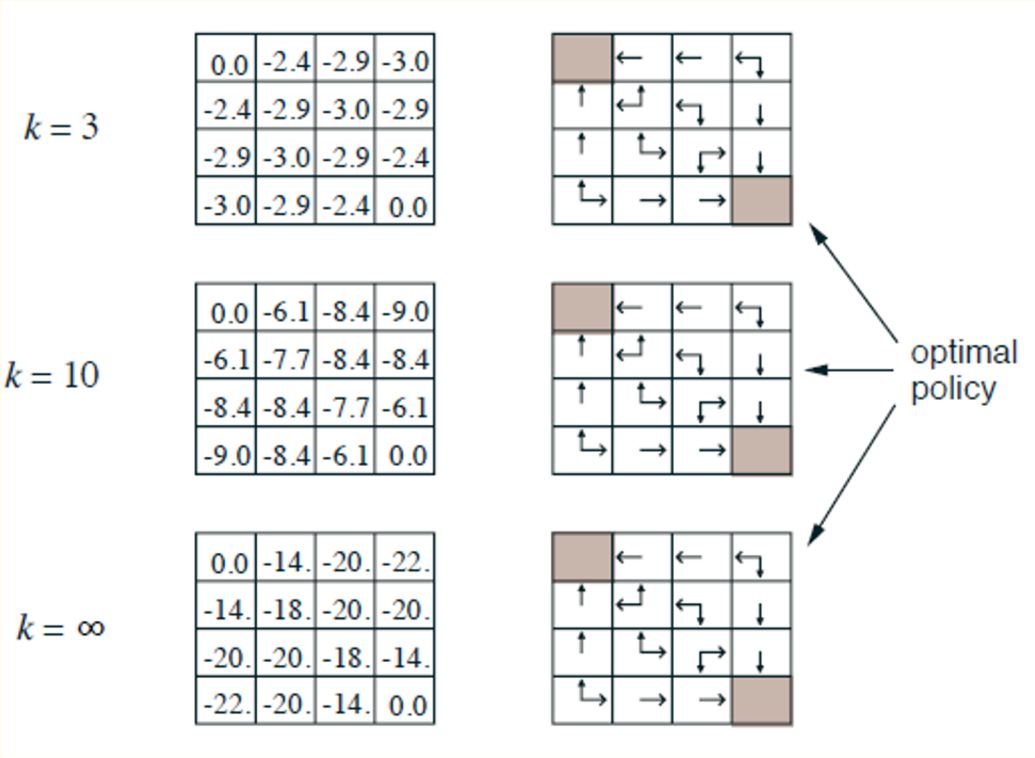
$k = \infty$

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

Policy evaluation



Policy evaluation



Policy iteration

How to Improve a Policy :

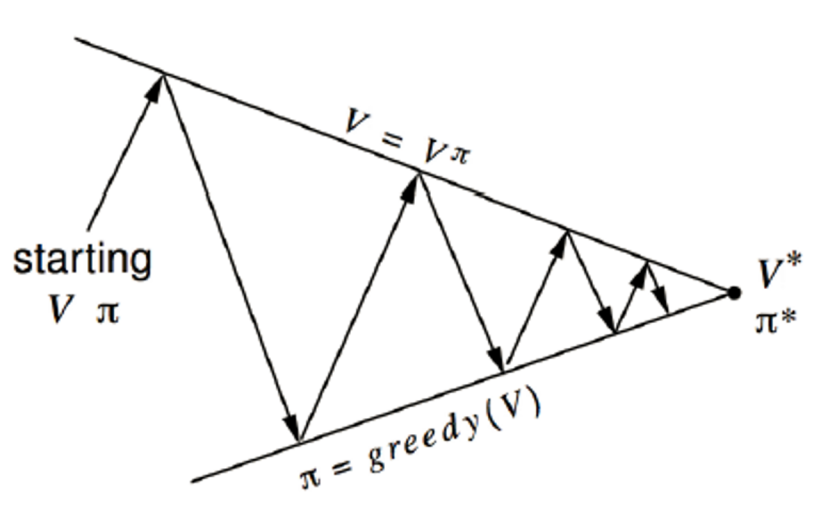
- Start from a given policy
  - Evaluate the policy using Policy Evaluation and estimate value function
  - Improve the policy using the computed value function

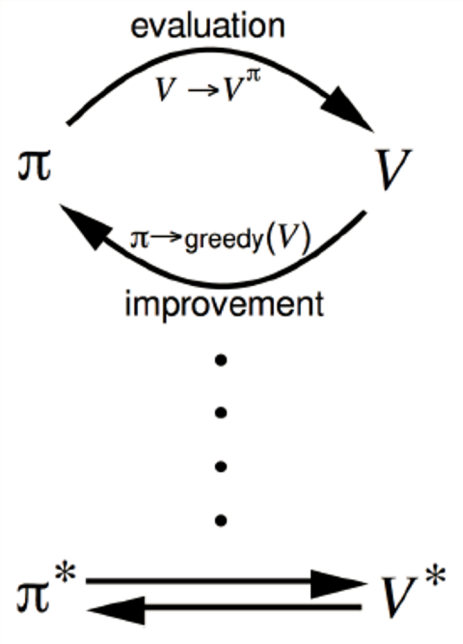
In the gridworld example, the policy was optimal after iteration 3 but in general we will need more iterations.

This process of policy iteration always converges to optimal policy

Policy iteration

Iterations :





We finish when improvement stops

Value iteration

Optimality principle :

- An optimal policy can be subdivided into two components :
  - An optimal first action a*
  - Followed by an optimal policy at next state s’

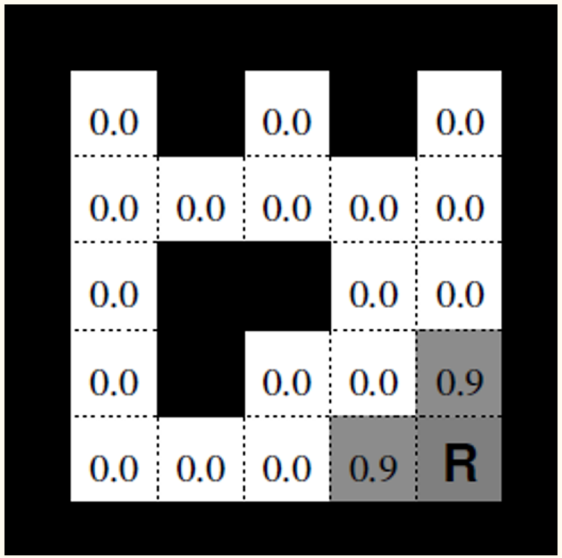
If we know the solution to subproblem v*(s’) then solution v*(s) can be found by one-step lookahead. We apply these updates iteratively

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Problem that sometimes we don’t know the end

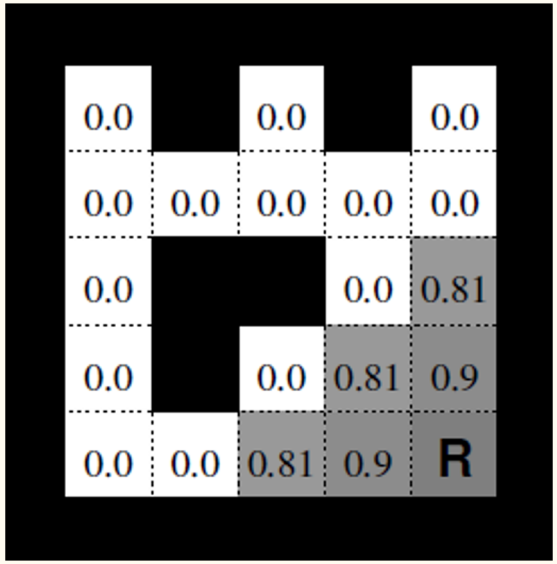
Value iteration

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$



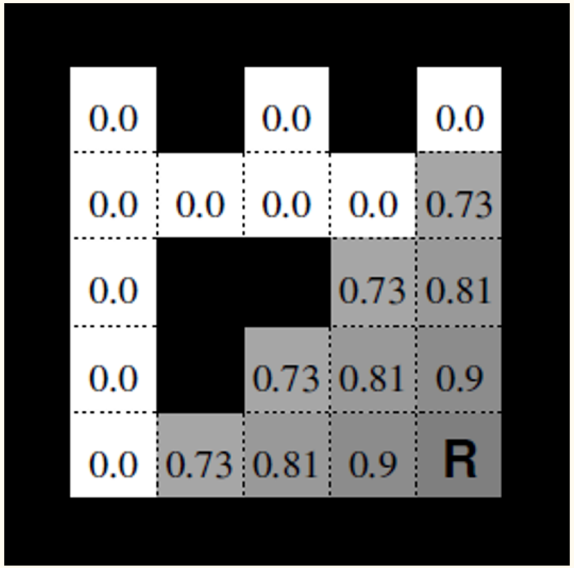
Value iteration

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$



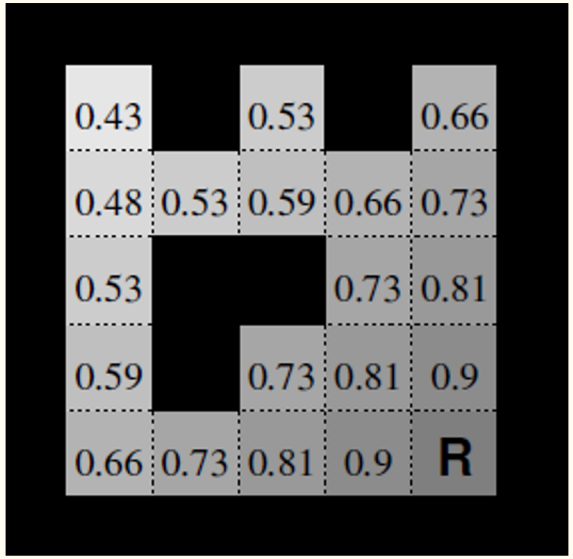
Value iteration

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$



Value iteration

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$



Value iteration

Iterative application of Bellman optimality backup.

- Use synchronous backups:
  - At each iteration k+1
  - For states s in S
  - Update $v_{k+1}(s)$ from $v_k(s')$ with s' after s.

Limitations :

- Unlike policy iteration there is no explicit policy.
- Intermediate value function do not correspond to any policy
- We do not know always the location of the final state for the problem.

Limitations

Until know we have a good representation of the environment / system (states, transitions, probabilities, ...)

But in real life we don’t have a knowledge of the complete system...

Model-Free Reinforcement Learning

Model Free

When you don’t have a knowledge of the complete system.

It is an algorithm which does not use the transition probability distribution (and the reward function) associated with the Markov decision process (MDP).

Algorithm learns directly from episodes of experience.

Monte Carlo Reinforcement Learning

MC methods learn directly from episodes of experience by computing the mean of obtained rewards over all games.

MC is model-free: no knowledge of MDP transitions / rewards

MC learns from complete episodes: no bootstrapping

MC uses the simplest possible idea: value = mean return

Can only apply MC to episodic MDPs

-> All episodes must terminate

Monte Carlo Policy Evaluation

Goal: learn value function V from episodes of experience under policy

Recall that the return is the total discounted reward

Recall that the value function is the expected return:

$$v(s) = E\left[G_t \mid S_t = s\right]$$

Monte-Carlo policy evaluation uses empirical mean return instead of expected return

First visit Monte Carlo Policy Evaluation

To evaluate state s

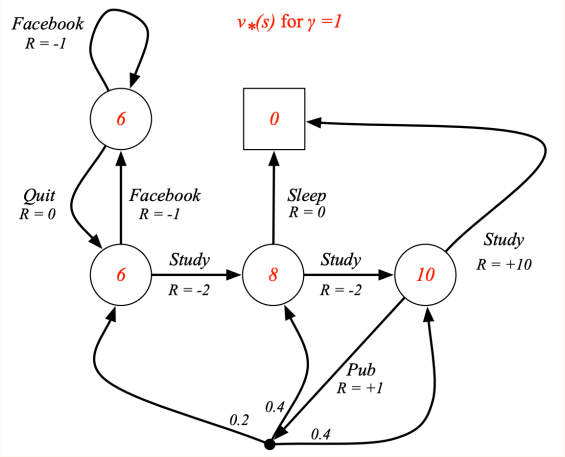
The first time-step t that state s is visited in an episode,

Increment counter $N(s) <- N(s) + 1$

Increment total return $S(s) <- S(s) + G_t$

Value is estimated by mean return $V(s) = S(s)/N(s)$

By law of large numbers, $V(s) -> vp(s)$ as $N(s) ->$ infinite



Every visit Monte Carlo Policy Evaluation

To evaluate state s

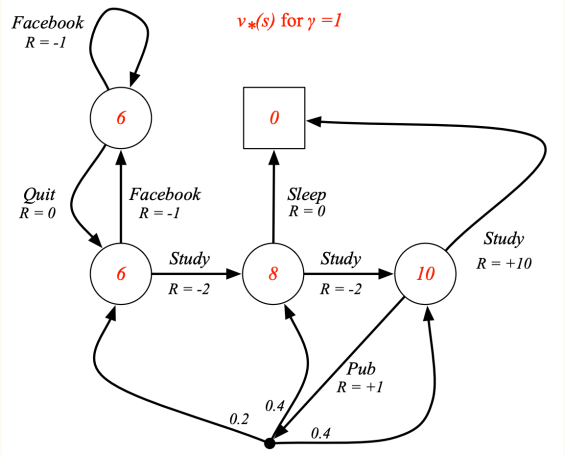
Every time-step t that state s is visited in an episode,

Increment counter $N(s) <- N(s) + 1$

Increment total return $S(s) <- S(s) + G_t$

Value is estimated by mean return $V(s) = S(s)/N(s)$

By law of large numbers, $V(s) -> vp(s)$ as $N(s) ->$ infinite



Incremental Monte Carlo updates

We will use a formula to update V(s) after 1 episode (a sequence of states/actions)

$$N(S_t) \leftarrow N(S_t) + 1$$

$$V(S_t) \leftarrow V(S_t) + (1/N(S_t)) * (G_t - V(S_t))$$

(à démontrer)

Incremental Monte Carlo updates

$$
\begin{aligned}
V_k(S_t) &= (1/k) * \sum_{i=1}^{k}(G_i) \\
&= (1/k) * (G_k + \sum_{i=1}^{k-1}(G_i)) \\
&= (1/k)*G_k + (1/k)* \sum_{i=1}^{k-1}(G_i) \\
&= (1/k)*G_k + (1/k)*((k\text{-}1)/(k\text{-}1))* \sum_{i=1}^{k-1}(G_i) \\
&= (1/k)*G_k + ((k\text{-}1)/k)*(1/(k\text{-}1))* \sum_{i=1}^{k-1}(G_i) \\
&= (1/k)*G_k + ((k\text{-}1)/k)* V_{k\text{-}1}(S_t) \\
&= (1/k)*G_k + V_{k\text{-}1}(S_t) \text{ - } (1/k)* V_{k\text{-}1}(S_t) \\
&= V_{k\text{-}1}(S_t) + (1/k)* (G_k \text{ - } V_{k\text{-}1}(S_t))
\end{aligned}
$$

$$V(S_t) \leftarrow V(S_t) + (1/N(S_t)) * (G_t - V(S_t))$$

Temporal Difference Learning

TD methods learn directly from episodes of experience

TD is model-free: no knowledge of MDP transitions / rewards

TD learns from incomplete episodes, by bootstrapping

TD updates a guess towards a guess

Temporal Difference Formula

TD methods combine MC and DP to update the value function toward estimated return $R_{t+1} + g*V(S_{t+1})$.

For one step ahead TD formula is :

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

α & γ are parameters of the model to adjust to reach convergence

MC and TD methods

Goal: learn Value Function online from experience under policy

Incremental every-visit Monte-Carlo

Update value V(St) toward actual return Gt

Simplest TD learning algorithm

Update value V(St) toward estimated return $\mathbf{R_{t+1} + \gamma V(S_{t+1})}$

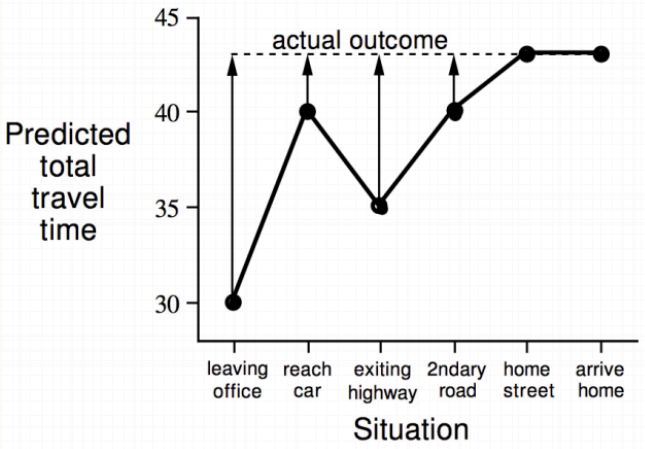
Example

Driving from work to home

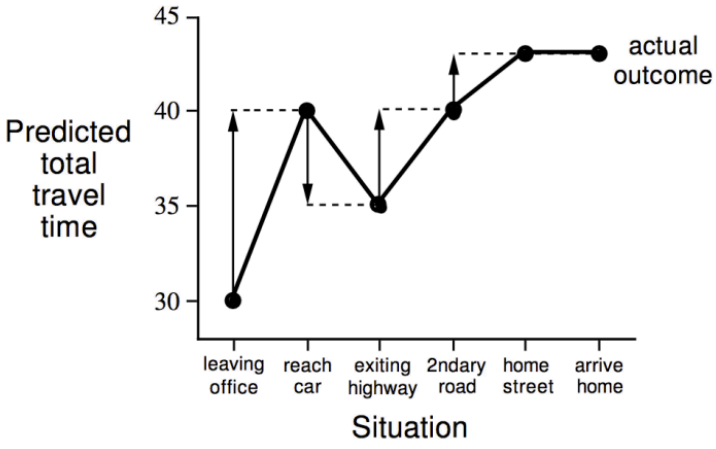
State	Elapsed Time	Predicted remaining Time	Predicted total Time
Leaving office	0	30	30
Car parking	5	35	40
Enter highway	20	15	35
Exit highway	30	10	40
Home street	40	3	43
Arrival home	43	0	43

MC vs TD

Changes recommended by Monte Carlo methods (α=1)



Changes recommended by TD methods (α=1)



Monte Carlo computation

Goal: Define the value function (ie total travel time) for each state

We have to wait the end of the episode to update all value state functions.

We update each value function toward actual travel time :

- No biais because we are using the actual travel time –> an observed value.
- High variance because very sensitive to observations.
- We have to wait the end of the episode

Temporal Difference computation

Goal: Define the value function (ie total travel time) for each state

We update at each state the value function.

We update each value function toward estimated return $\mathbf{R_{t+1} + \gamma V(S_{t+1})}$:

- Biais because we are using value functions −> an initialized value
- Low variance because not sensitive to observations.
- No need to wait the end

MC vs TD : pros & cons

TD can learn before knowing the final outcome

- TD can learn online after every step
- MC must wait until end of episode before return is known

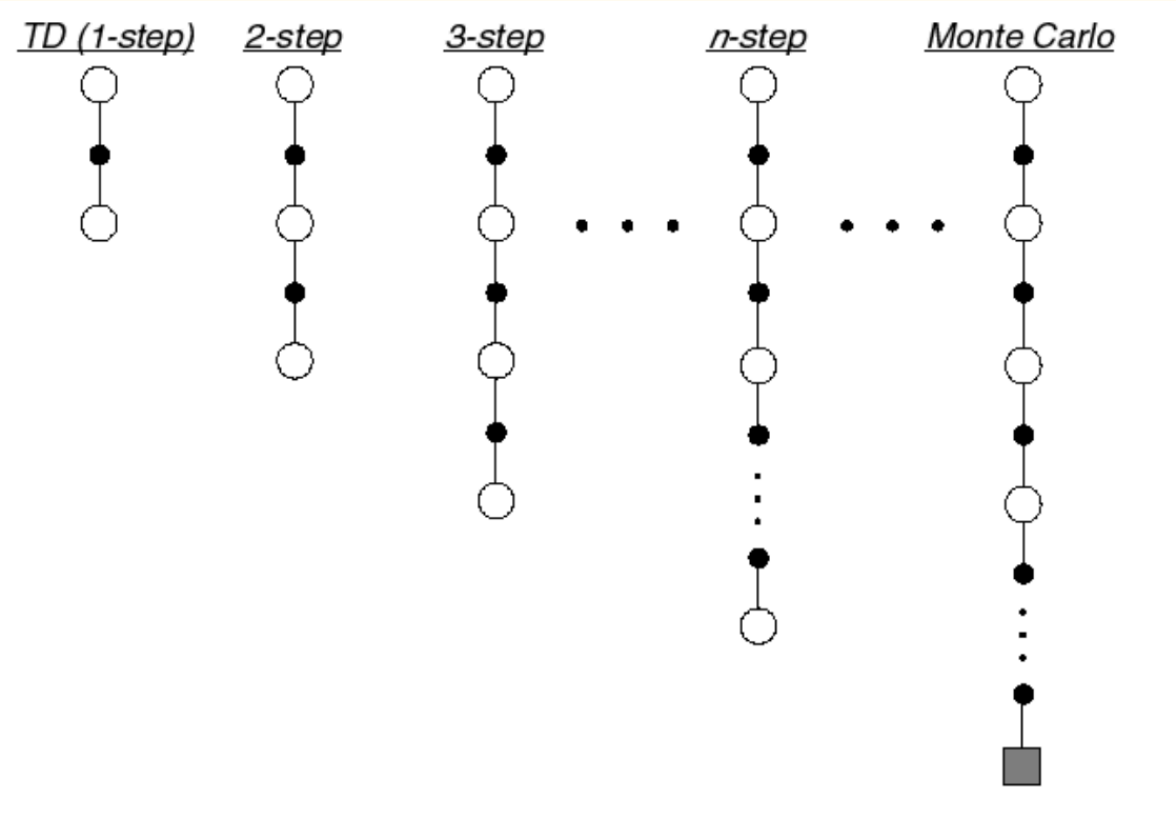
MC has high variance, zero bias

- Good convergence properties
- Not very sensitive to initial value
- Very simple to understand and use

TD has low variance, some bias

- Usually more efficient than MC
- More sensitive to initial value

n-steps TD



N-steps TD

Consider the following n-step returns for $n = 1, 2, \infty$:

$$
\begin{aligned}
n = 1 \quad (\text{TD}) \qquad & G^{(1)}_t = R_{t+1} + \gamma V(S_{t+1}) \\
n = 2 \qquad\qquad\ \ & G^{(2)}_t = R_{t+1} + \gamma(R_{t+2} + \gamma V(S_{t+2})) \\
& \qquad = R_{t+1} + \gamma\, R_{t+2} + \gamma^2\, V(S_{t+2}) \\
& \ldots \\
n = \infty \quad (\text{MC}) \qquad & G^{(\infty)}_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T
\end{aligned}
$$

For step n we define n-step Temporal Difference :

$$\mathbf{V(St) \leftarrow V(St) + \alpha\,(G^{(n)}_t - V(S_t))}$$

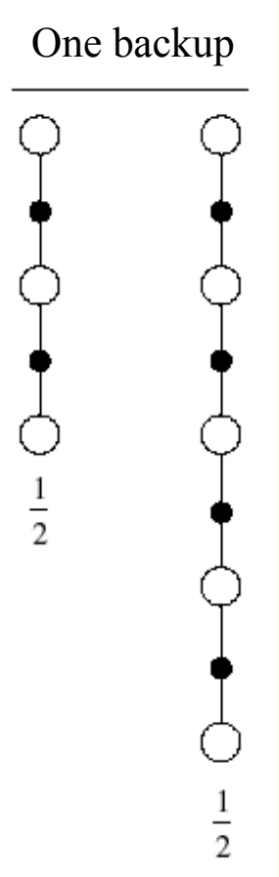
Averaging n-steps TD

We can average n-step returns over different n
(average the 2-step and 4-step returns)

$$\tfrac{1}{2}G^{(2)} + \tfrac{1}{2}G^{(4)}$$

Combines information from two different time-steps

Can we efficiently combine information from all time-steps?



λ-return

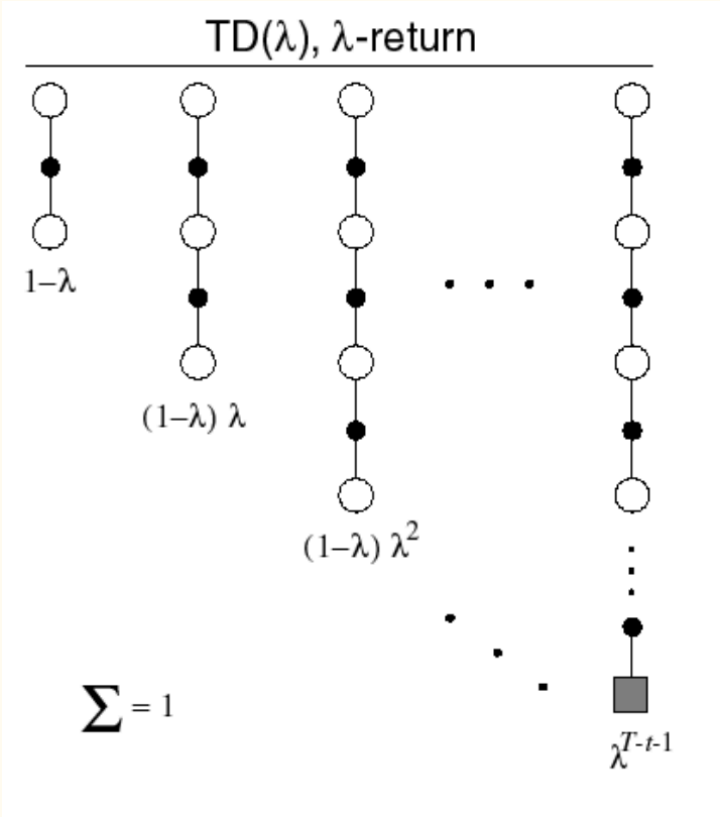
We can average n-step returns over different n
(average the 2-step and 4-step returns)

$$\tfrac{1}{2}G^{(2)} + \tfrac{1}{2}G^{(4)}$$

The λ-return G^{λ}_t combines all n-step returns $G^{(n)}_t$ using weight $(1-\lambda)\lambda^{n-1}$

$$G^{\lambda}_t = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G^{(n)}_t$$

$$V(St) \leftarrow V(St) + \alpha\,(G^{\lambda}_t - V(S_t))$$



Policy iteration & model free control

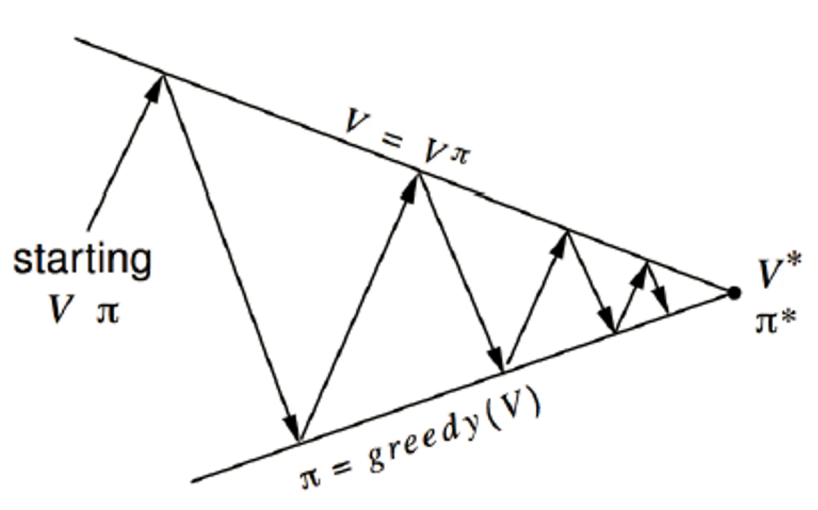
Goal: Define the best model free control using model-free estimation and policy iteration

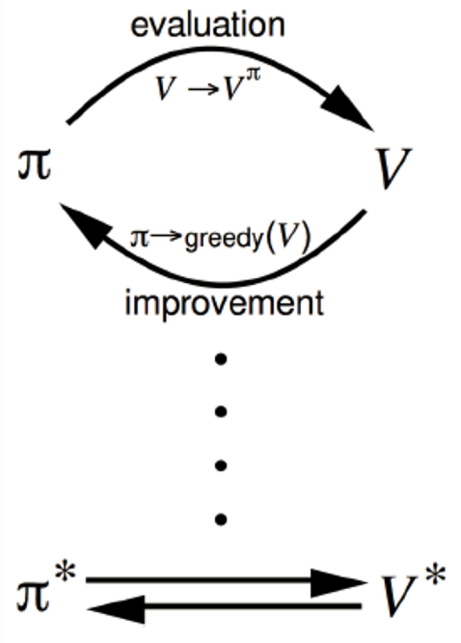
Used to solve either :

- Unknown MDP model : we learn from experience the best way to take action
- Known MDP model : Sometimes there are too large to explore every state/action and model free control can solve these problem.

Policy iteration - refresh

Iterations :





We finish when improvement stops.

BUT GREEDY ACTION SELECTION IS LIMITED !

Limitation of greedy improvement

Greedy improvement cannot always improve our policy because :
- We can get stuck in an local optimum
- We have to know the MDP model

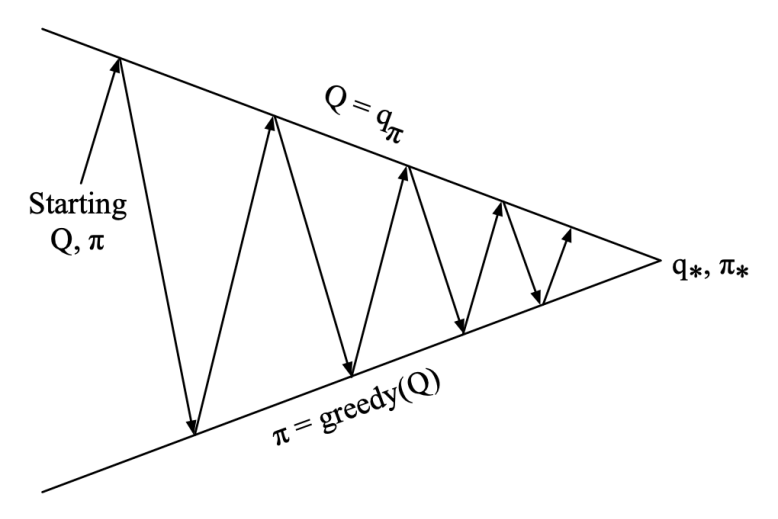
$$\pi'(s) = \underset{a \in \mathcal{A}}{\text{argmax}}\ \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$

Instead we rather use the action value function in model free control

$$\pi'(s) = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s, a)$$

Policy iteration using action value function

Iterations :



Still here the Greedy action selection is limited

ε-Greedy exploration

Goal: Introduction randomness in Greedy action selection.

Regarding all possible actions we define :

- Choose Greedy action (ie best action possible) with probability 1- ε
- Choose a random action with probability ε

Policy improvement is assumed (no demonstration)

GLIE Monte Carlo control

GLIE : Greedy in the Limit with Infinite Exploration

- We explore all state-action pair if iterations are infinite.
- We update value function according to MC formula

$$N(S_t ; A_t) = N(S_t ; A_t) + 1$$

$$Q(S_t ; A_t) = Q(S_t ;A_t) + (1/(N(S_t ; A_t)) *(G_t - Q(S_t ;A_t))$$

- ε must reduces to 0 when iterations are getting large : ex $\varepsilon = 1/k$

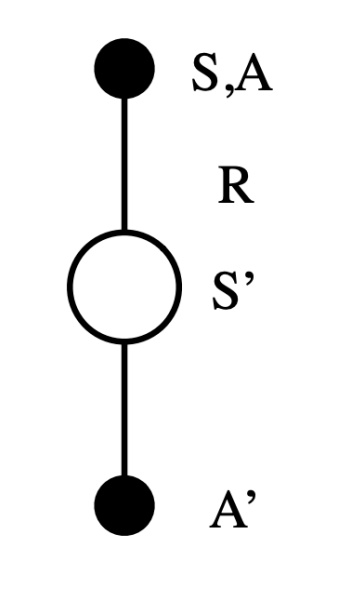
From MC to TD control

We saw that TD has some advantages over MC :

- Lower variance
- Online
- Learn from incomplete sequence

We will use TD to action value function Q(S,A) using ε – greedy policy improvement with update at each time-step

TD control



$$Q(S_t, A_k) \leftarrow Q(S_t, A_k) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{k+1}) - Q(S_t, A_k))$$

SARSA control

Updating action value function using TD learning

$$Q(S_t, A_k) = Q(S_t, A_k) + \alpha(R_{t+1} + \gamma Q(S'_{t+1}, A'_k) - Q(S_t, A_k))$$

SARSA converges to the optimal action-value function using ε-greedy if ε reduces to 0 when iterations become large.

There is also extension to n-steps SARSA looking n steps ahead

SARSA algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Repeat (for each step of episode):
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until S is terminal

N-steps SARSA

Consider the following n-step returns for $n = 1, 2, \infty$:

$$
\begin{aligned}
n = 1 \quad (\text{Sarsa}) \quad & q^{(1)}_t = R_{t+1} + \gamma Q(S_{t+1}, A) \\
n = 2 \quad & q^{(2)}_t = R_{t+1} + \gamma(R_{t+2} + \gamma Q(S_{t+2})) \\
& \qquad = R_{t+1} + \gamma\, R_{t+2} + \gamma^2\, Q(S_{t+2}) \\
& \ldots \\
n = \infty \quad (\text{GLIE}) \quad & q^{(\infty)}_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T
\end{aligned}
$$

For step n we define n-step Sarsa update :

$$\mathbf{Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\,(q^{(n)}_t - Q(S_t, A_t))}$$

N-Steps SARSA algorithm

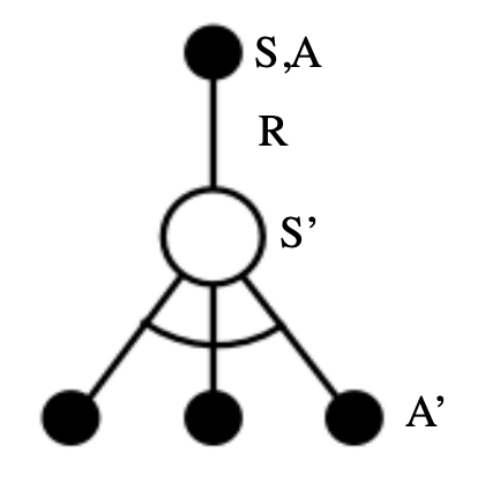
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
    $E(S, A) \leftarrow E(S, A) + 1$
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha \delta E(s,a)$
      $E(s,a) \leftarrow \gamma \lambda E(s,a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

From On-Policy to Off-Policy Learning

Off-Policy Learning : Learning from observing someone's else strategy.

- Evaluate target policy $\pi(A|S)$ to compute $Q(S, A)$.
- While following behaviour policy $\mu(A|S)$
- Learn about optimal policy while following exploratory policy
- Learn about multiple policies while following one policy

Q-learning control



$$Q(S_t, A_k) = Q(S_t, A_k) + \alpha(R_{t+1} + \gamma \max Q(S_{t+1}, A'_k) - Q(S_t, A_k))$$

Q-Learning - SARSAMAX

We improve now both behaviour and target policies

- The target policy π is greedy

$$\pi(S_{t+1}) = \text{argmax } Q(S_{t+1}; A')$$

- The behaviour policy μ is ε-greedy

Formula :

$$Q(S_t, A_k) = Q(S_t, A_k) + \alpha(R_{t+1} + \gamma \text{argmax} Q(S'_{t+1}, A'_k) - Q(S_t, A_k))$$

Q learning algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
        $S \leftarrow S'$;
    until S is terminal

Reinforcement Learning in practice

Function Approximation

Before :

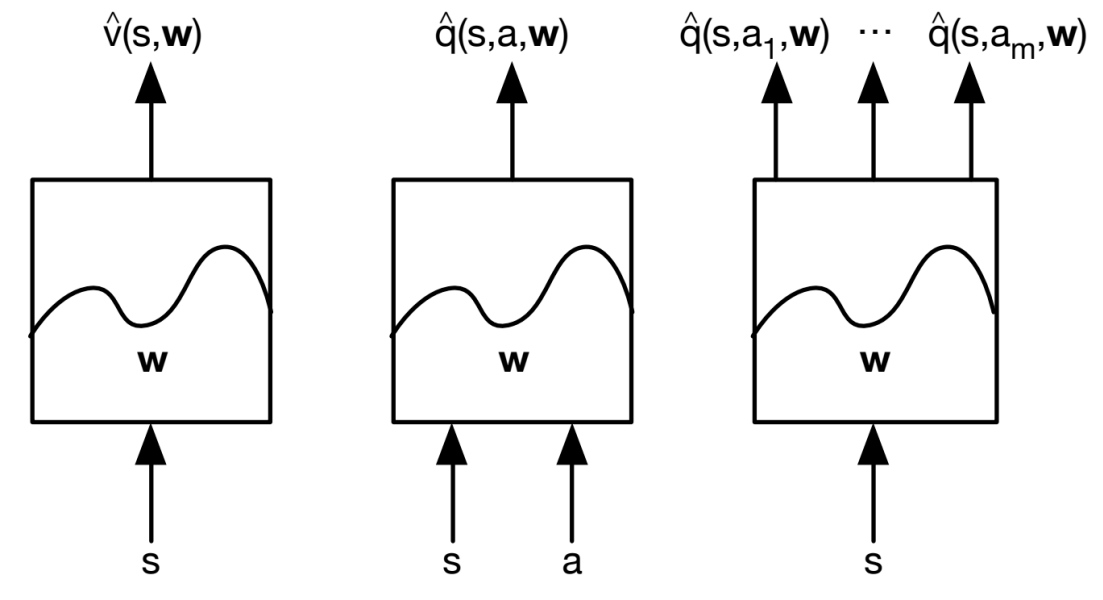
- Every state S has an associated value function V(S)
- Every state - action (S,A) has an associated action value function Q(S,A)

Problem is for large systems are too many data to store in memory and too slow to learn the values.

Now :

Estimation of a *function approximation* for both V(S) and Q(S,A) known as V(S,w) and Q(S,A,w) with w parameters to estimate.

Function Approximation



Function Approximator

Many functions can be used to approximate $V(S)$ and $Q(S,A)$.

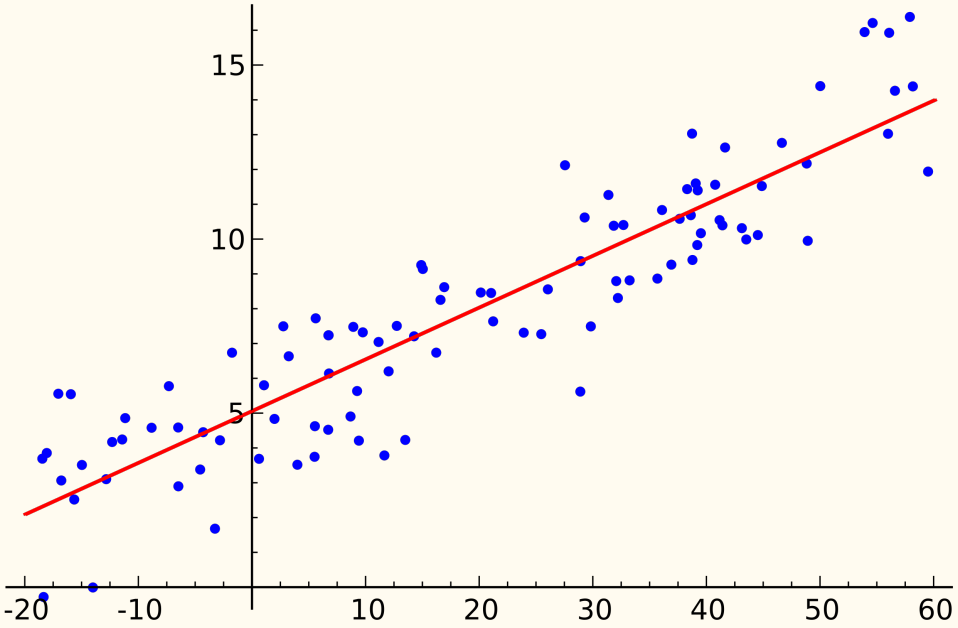
We use linear function in this course because it is easy to differentiate parameter **w** :

- Linear functions
- Neural Networks

Then we use Gradient Descent to update parameter **w.**

Function Approximator : Linear Regression

Estimate a linear function H_θ to fit a set of observed elements (x_i, y_i):



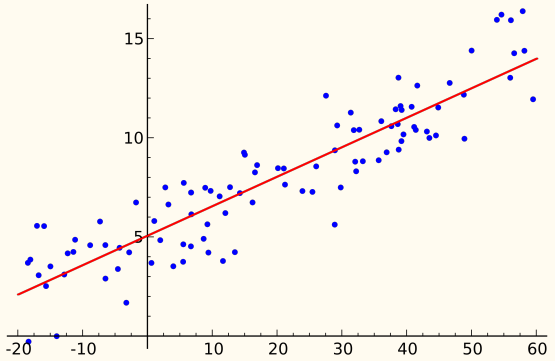
Function Approximator : Linear Regression

Estimate a linear function H_θ to fit a set of observed elements (x_i, y_i):

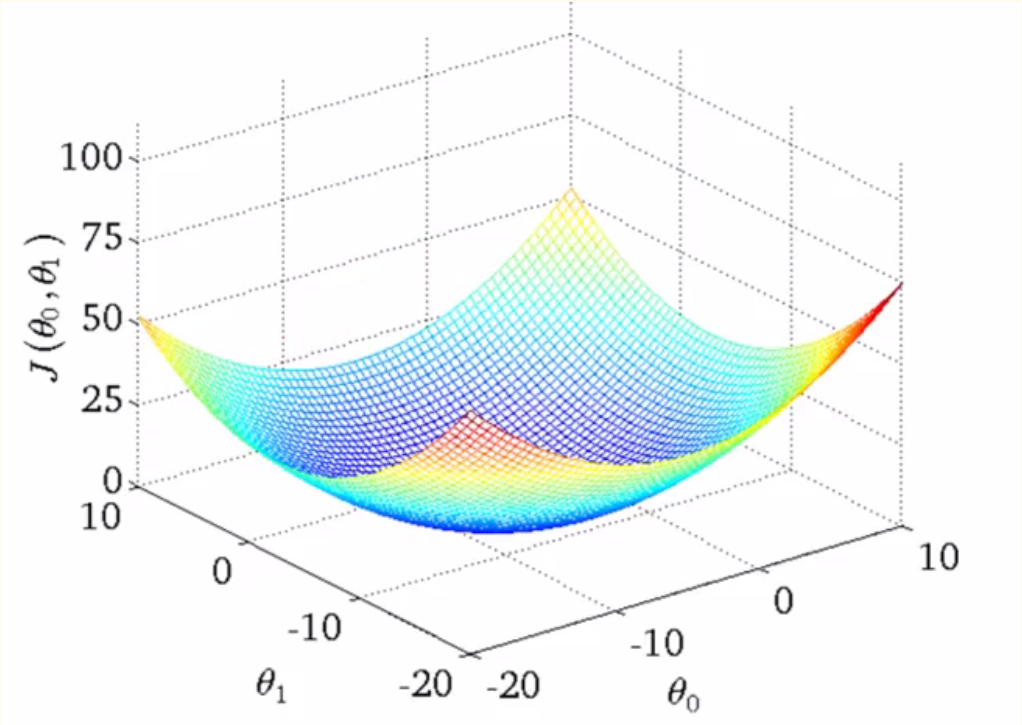
Let's define a cost function to evaluate the accuracy of our estimator H_θ .
Having $H_\theta(x) = \theta_0 + x * \theta_1$

The cost function associated is :

$$J(\theta_0, \theta_1) = \left(\frac{1}{2m}\right) \sum_{k=0}^{m} (H_\theta(x_k) - y_k)^2$$



Cost function



Gradient descent

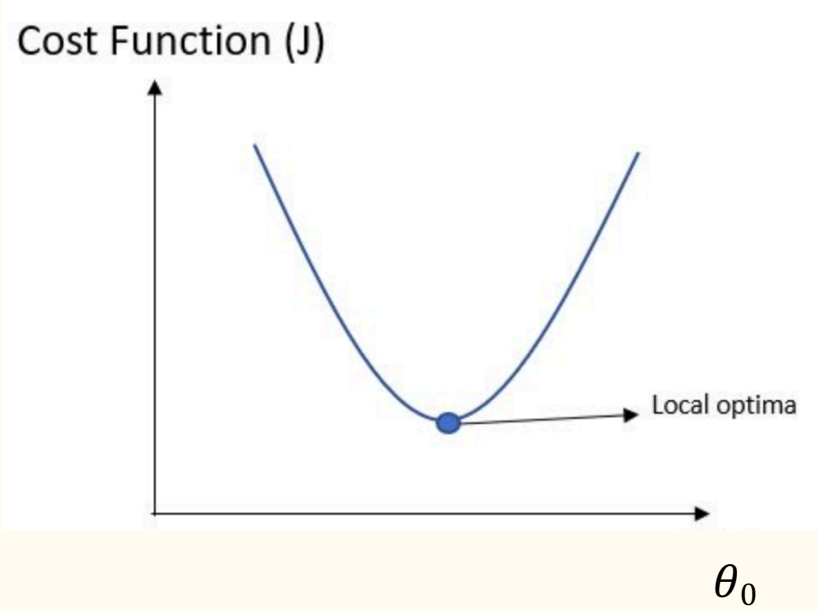
Find parameters θ_0 , θ_1 to minimize cost function :

- First : start with random values θ_0 , θ_1 and compute J
- Differentiate J according to each parameter θ_0 , θ_1 and move in the way to reduce cost function

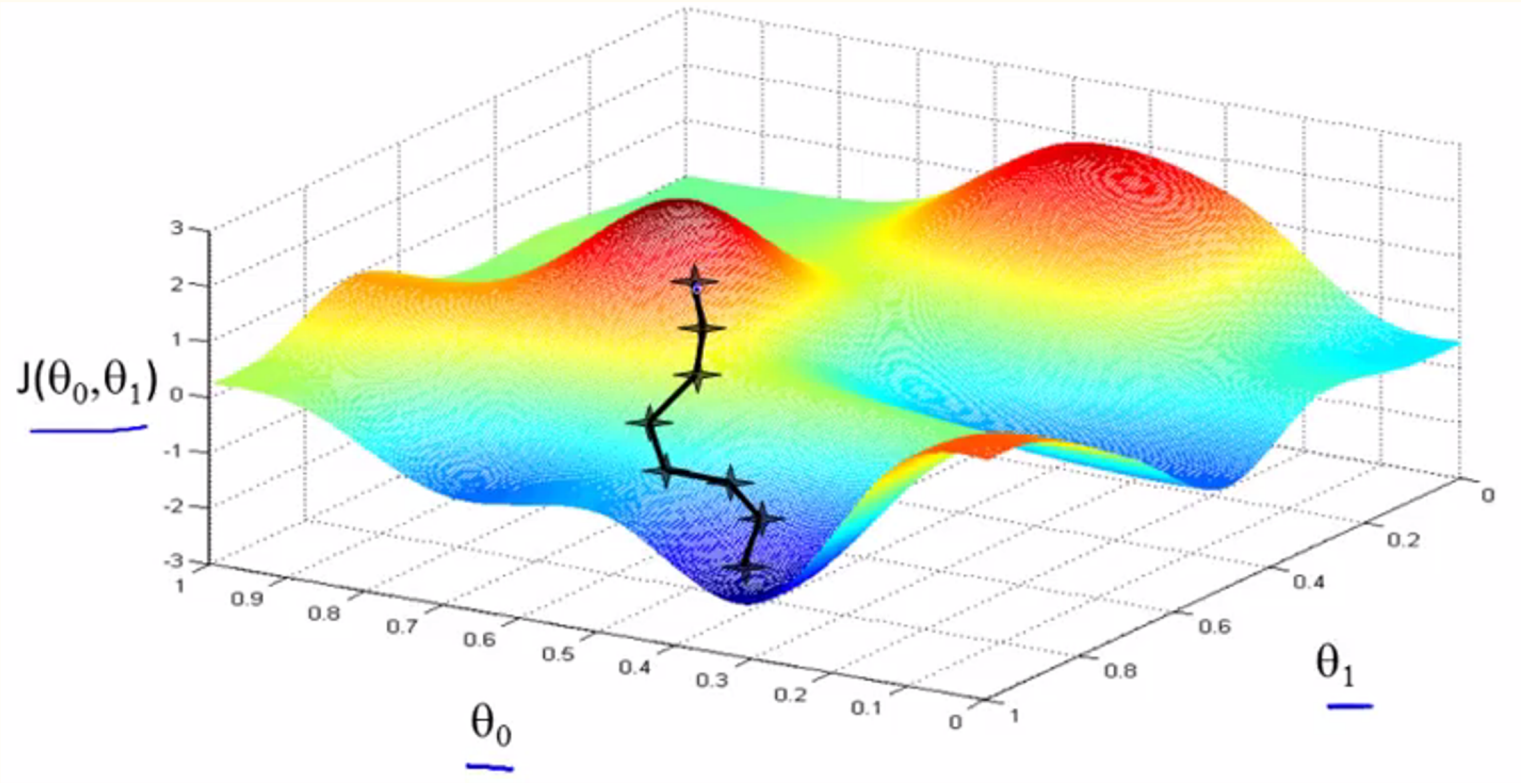
$$\theta_0 \leftarrow \theta_0 - \alpha * \partial(J(\theta_0, \theta_1))/\partial \theta_0$$

$$\theta_0 \leftarrow \theta_0 - \alpha * \partial(J(\theta_0, \theta_1))/\partial \theta_0$$

α is the learning rate



Gradient Descent recall



Gradient Descent for Value Function

Let's define the cost function J as :

$$J(w) = -\frac{1}{2} * E[\ (V(S) - V(S,w))^2\ ]$$

Gradient descent is define by :

$$\partial w = \alpha * E[(V(S) - V(S,w)) * \partial(V(S,w))/\partial w]$$

And for each observation we get :

$$\partial w = \alpha * (V(S) - V(S,w)) * \partial(V(S,w))/\partial w$$

Gradient Descent for Action Value Function

Let's define the cost function J as :

$$J(w) = -\frac{1}{2} * E[\ (Q(S,A) - Q(S,A,w))^2\ ]$$

Gradient descent is define by :

$$\partial w = \alpha * E[(Q(S,A) - Q(S,A,w)) * \partial(Q(S,A,w))/\partial w]$$

And for each observation we get :

$$\partial w = \alpha * (Q(S,A) - Q(S,A,w)) * \partial(Q(S,A,w))/\partial w$$

Value Function Approximation with MC

We use the global return G_t as a true value of V(S) :

$$\partial w = \alpha * (G_t - V(S,w)) * \partial(V(S,w))/\partial w$$

And when using a lookup table we get :

$$\partial w = \alpha * (G_t - V(S,w)) * x(S)$$

MC converges to a local optimum !

Value Function Approximation with TD(0)

We use the TD target $R_{t+1} + \gamma V(S_{t+1}, w)$ as a true value of $V(S)$:

$$\partial w = \alpha * (R_{t+1} + \gamma V(S_{t+1}, w) - V(S, w)) * \partial(V(S, w))/\partial w$$

And when using a lookup table we get :

$$\partial w = \alpha * (R_{t+1} + \gamma V(S_{t+1}, w) - V(S, w)) * x(S)$$

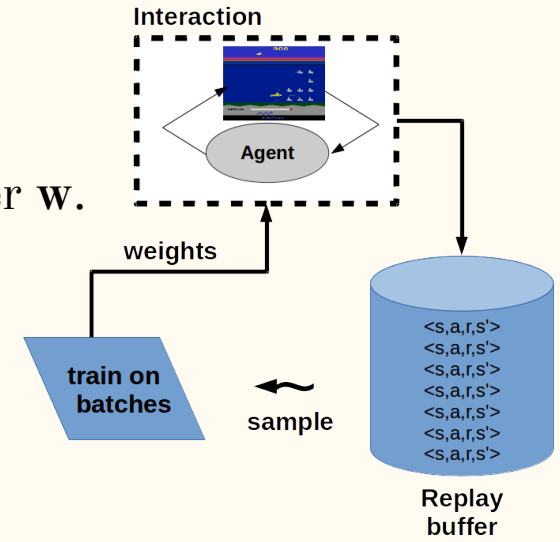
Introducing Experience Replay

Batch method is used because Gradient Descent updates step by step and don't use the "experience".

We create a set D of (state, value) pair to update parameter **w**.
We use the Least Squares algorithm formula :

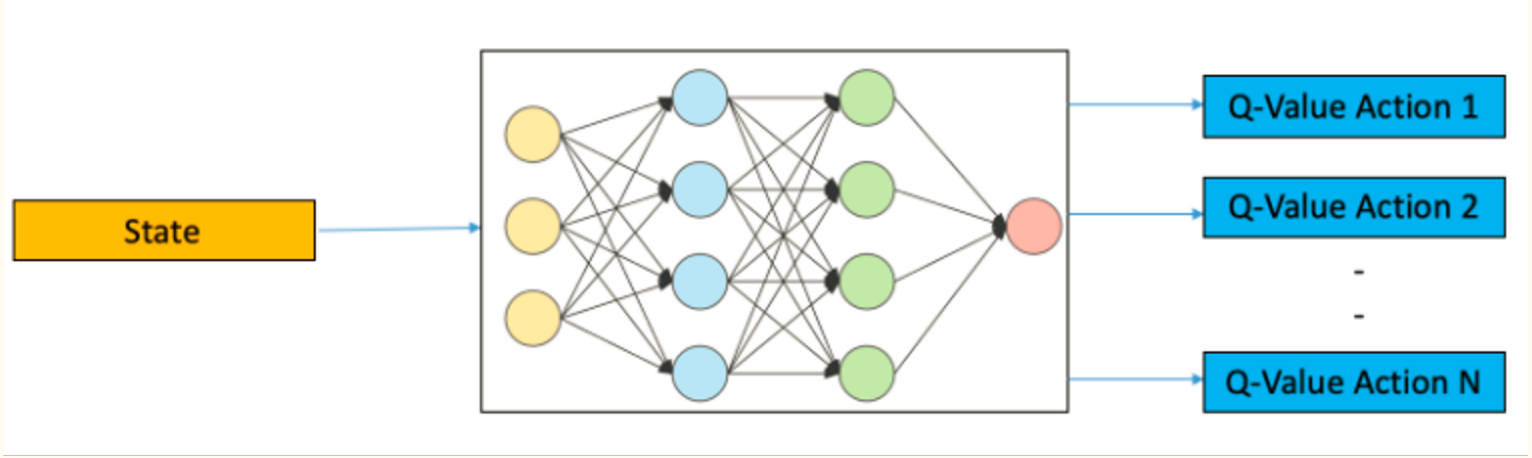
$$LS(w) = -\frac{1}{2} * \sum E[\ (V(S) - V(S,w))^2\ ] \quad \text{over D}$$

Repeat stochastic gradient descent over each (,value) of D converges to LS.



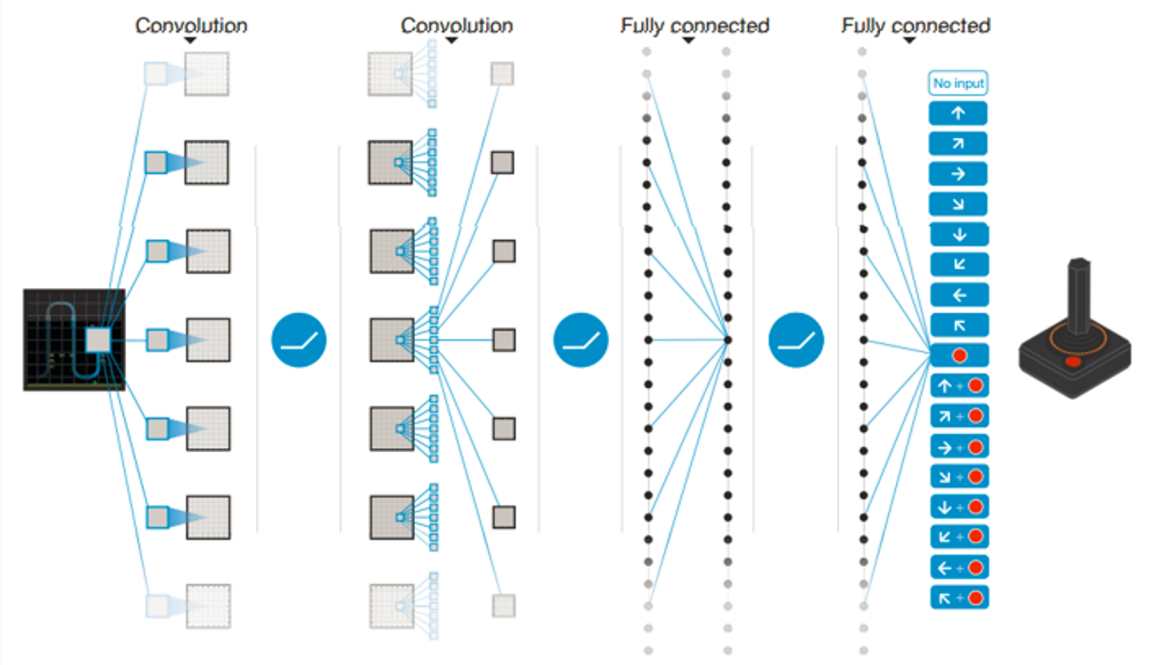
DQN for ATARI game

Modelisation of NN to determine from state, each action value.



DQN for ATARI game

Using CNN, with a fixed architecture !!!



End