IT00DP82-3007 Artificial Intelligence and Machine Learning

peter.hjort@metropolia.fi

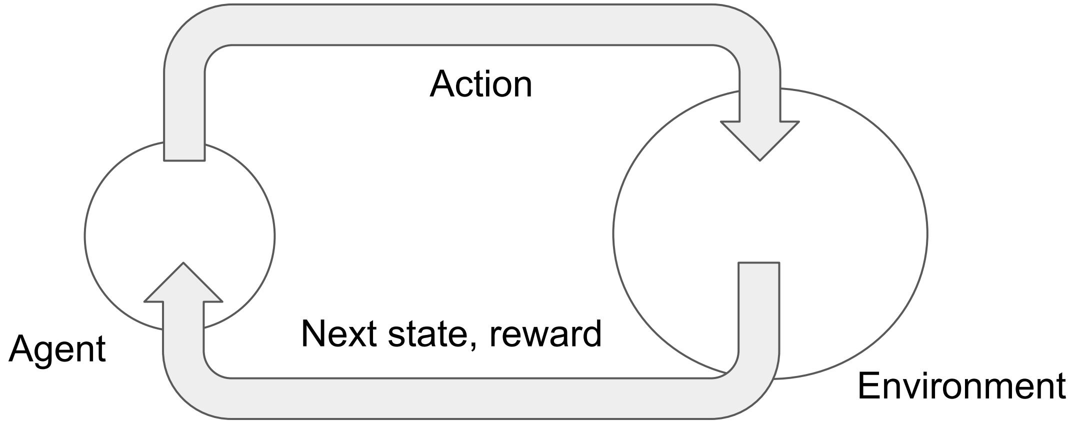
Note: please include string IT00DP82 in the subject field of any email correspondence

Reinforcement learning

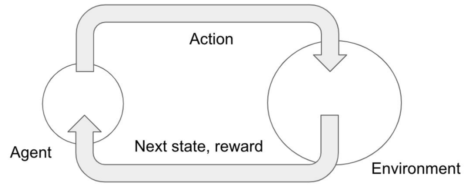
In reinforcement learning (RL) the set-up is more general than in supervised / unsupervised learning. Key concepts in RL are:

- a stateful **agent** (the system being developed) that takes **actions** to maximise long-term **return** (discounted sum of all rewards the agent collects during its operation),
- an **environment** (abstraction of the surroundings of the agent) that determines the **next state** of the agent and **reward** the agent gets.

Rewards are the only guide for the agent to learn to perform the right actions to eventually reach the goal (in episodic case).



Concepts in reinforcement learning



- Agent can influence its environment by taking **actions**. After taking an action the agent
  - Gets a **reward**, a real number, that helps to guide the agent towards desired behaviour. There is no other guidance, so the agent has to learn by interacting with the environment without knowing how the environment operates, using **trial-and-error**.
  - Moves to next state, in general in a stochastic (random) manner.
- Process is **sequential**, ie. proceeds step by step. In one timestep the agent performs an action, receives reward and moves to next state. Often the term **sequential decision making** is used.
- (The observed data (states) are not **i.i.d** (independent and identically distributed) but are correlated (state depends on previous state). This is in contrast with supervised learning).

Target of the agent

The agent is driven by the **return** (discounted sum of all collected rewards), ie. its behaviour is for example not necessarily to reach a designated end state (like with search-based systems).

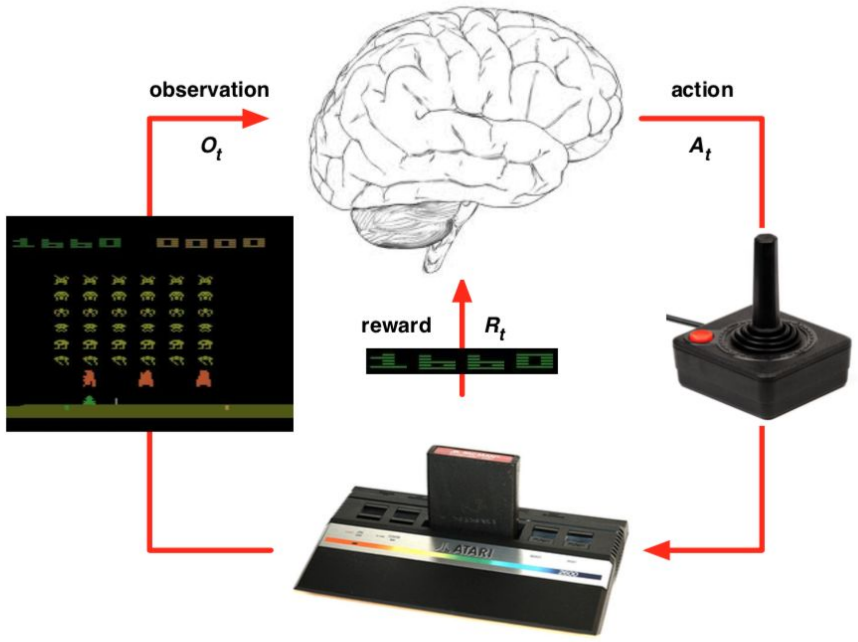
The agent does not maximise its **immediate reward** but rather the **discounted sum of rewards**.

This means that the agent should be able to find strategies where taking an action might immediately give lower reward than other actions but over a long term leads to a good result.

Examples of reinforcement learning

- A chess player makes a move.
- An adaptive controller adjusts parameters of a petroleum refinery operation in real time.
- A gazelle calf struggles to its feet minutes after being born.
- A mobile robot decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station.
- An advert management system deciding which ad to place to a page next.

Atari Example: Reinforcement Learning



- Rules of the game are unknown
- Learn directly from interactive game-play
- Pick actions on joystick, see pixels and scores

From Reinforcement Learning course slides by David Silver (https://www.davidsilver.uk/teaching/)

Examples

Helicopter & Atari games

copter and atari (15:40 - 17:29 and 17:49 - 21:56; note training time discussion was 2015)

Training for Atari Breakout

Breakout

Robots

Robots et al (3:00 - 4:45)

Markov Decision Process - MDP

Modeling the environment - Markov state

We say that the state model is **Markovian** iff (if and only if) the probability function for entering the next state S_{t+1} from state S_t satisfies

$$p(S_{t+1} \mid S_t) = p(S_{t+1} \mid S_1, S_2, \dots , S_t)$$

In other words, **probability of entering the next state depends only on current state**, not on states before the current state.

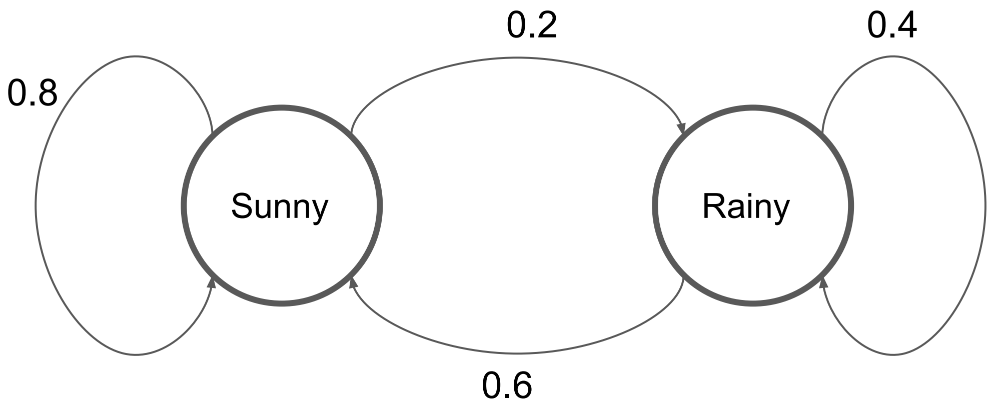
So, the current state captures everything there is to know - we need not care about the history.

The environment state in reinforcement learning is often/usually assumed to be Markovian.

Markov process

Markov process is a tuple <S, P>, where

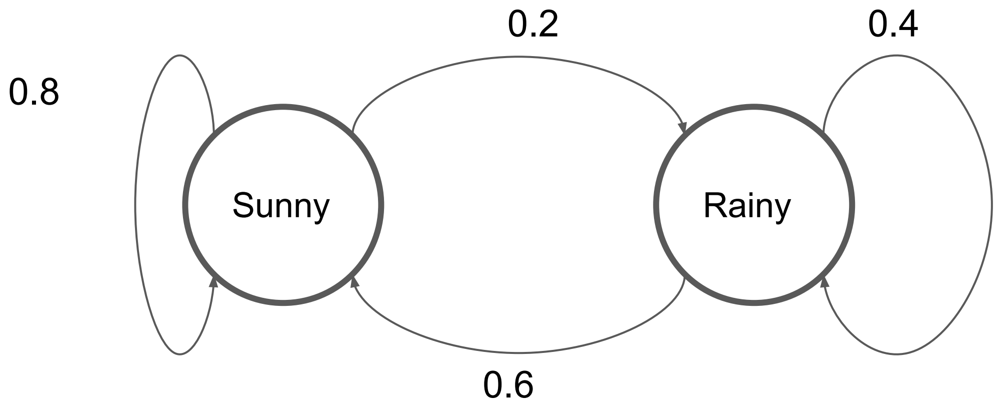
- S is the set of **states**
- P is the **(transition) probability function** $p(s, s') = p(S_{t+1} = s' \mid S_t = s)$



S = { Sunny, Rainy }

p(Sunny, Sunny) = 0.8
p(Sunny, Rainy) = 0.2
p(Rainy, Rainy) = 0.4
p(Rainy, Sunny) = 0.6

Transition matrix representation



S = { Sunny, Rainy }

p(Sunny, Sunny) = 0.8
p(Sunny, Rainy) = 0.2
p(Rainy, Rainy) = 0.4
p(Rainy, Sunny) = 0.6

Transition matrix P consists of the transition probabilities where rows denote the current state and columns denote the next state. Row sums are = 1.

	Sunny	Rainy
Sunny	0.8	0.2
Rainy	0.6	0.4

Running the Markov process

If the weather today is Rainy, how is the weather going to tomorrow? How about day after? How about in average?

```
import numpy as np
import numpy.linalg as LA

P = np.array(([0.2, 0.8], [0.6, 0.4]))
t0 = np.array([0.0, 1.0])

t1 = np.dot(t0, P)
print(t1)
t2 = np.dot(t1, P)
print(t2)
print(np.dot(t0, LA.matrix_power(P, 2)))
print(np.dot(t0, LA.matrix_power(P, 10)))
print(np.dot(t0, LA.matrix_power(P, 30)))
print(np.dot(t0, LA.matrix_power(P, 50)))
```

```
[0.6 0.4]

[0.36 0.64]
[0.36 0.64]
[0.42852649 0.57147351]
[0.42857143 0.57142857]
[0.42857143 0.57142857]
```

Markov reward process

Markov reward process is a tuple <S, P, R, γ>, where

- S is the set of **states**
- P is the **transition probability function** $p(s, s') = p(S_{t+1} = s' \mid S_t = s)$
- R is the **reward function** $R(s) = \mathbb{E}(R_{t+1} \mid S_t = s)$
- γ is the **discount factor** in range 0..1

Note: reward function is defined as an expectation, ie. we consider all possible rewards in a state weighted by their probabilities.

Reward examples

- One positive reward when the target has been achieved

- Intermediate rewards and a reward in the end - need to potentially balance between intermediate and end but gives the agent some direction

- Negative reward per step - give incentive to reaching the target fast

Return and discount factor

Return G_t at timestep t is the sum of all rewards after step t:

$$G_t \stackrel{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

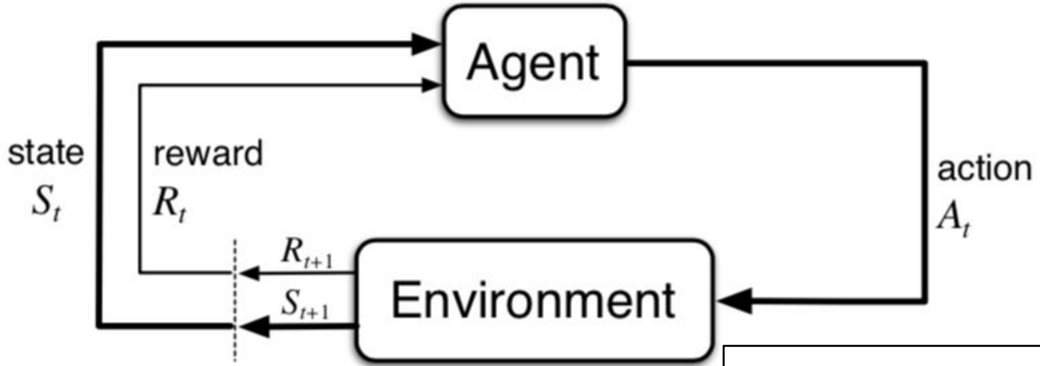
Where γ is in range 0..1 (inclusive) is the discount factor.

Discount factor ($\gamma < 1$) is used for reducing the weight of future rewards in the return. This reflects the idea that future rewards are risky (compare with net present value calculation in finance).

Markov decision process

A tuple $<S, A, p, \gamma>$ is a Markov decision process where

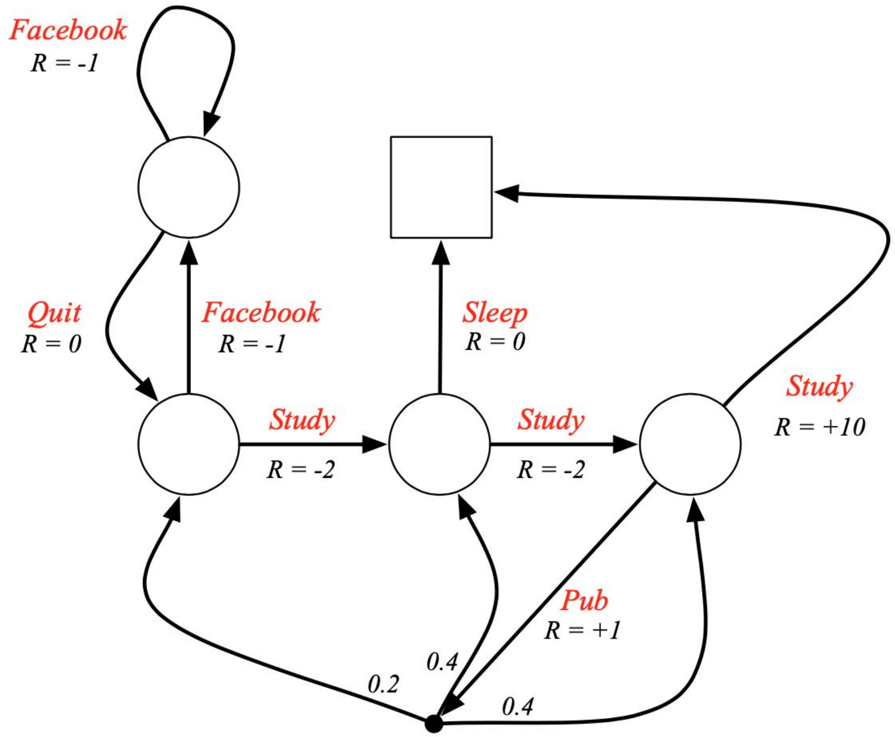
- S is the set of **states**, $S = \{ S_t , t = 0, 1, 2, ... \}$
- A is the set of **actions** $A = \{ A_t, t = 0, 1, 2, … \}$
- p is the **transition function** $p(s', r \mid s, a) \overset{def}{=} Pr\{ s' = S_t , r = R_t \mid s = S_{t-1} , a = A_t \}$ where R_t is the return (a real number) at time step t
- γ is the discount factor for computing **return**: $G_t \overset{def}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$



In reinforcement learning the environment is usually assumed (either explicitly or implicitly) to operate like a Markov decision process.

From Barto & Sutton: Introduction to Reinforcement Learning, 2nd Edition

MDP example: Student life



(Here reward depends only on the action taken, not on the outcome of the action. This simplifies specification and speeds up learning; in the end it is a modeling choice.)

From Reinforcement Learning course slides by David Silver (https://www.davidsilver.uk/teaching/)

Some properties of the transition function

For given state *s* and action *a* the probabilities of next states *s'* and rewards *r* sum up to 1:

$\sum_{s' \text{ in } S} \sum_{r \text{ in } R} p(s', r \mid s, a) = 1$

State-transition probabilities can be computed like this:

$p(s' \mid s, a) = \sum_{r \text{ in } R} p(s', r \mid s, a)$

Expected rewards for state-action pairs:

$r(s, a) = \sum_{r \text{ in } R} r \sum_{s' \text{ in } S} p(s', r \mid s, a)$

Policy,
state-value function and
action-value function

Policy

Policy π gives the action distribution (probability for each action) for all states:

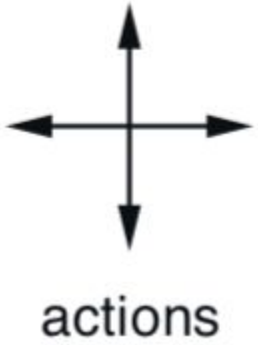
$$\pi(a \mid s) = \Pr \{ a = A_t \mid s = S_t \}$$

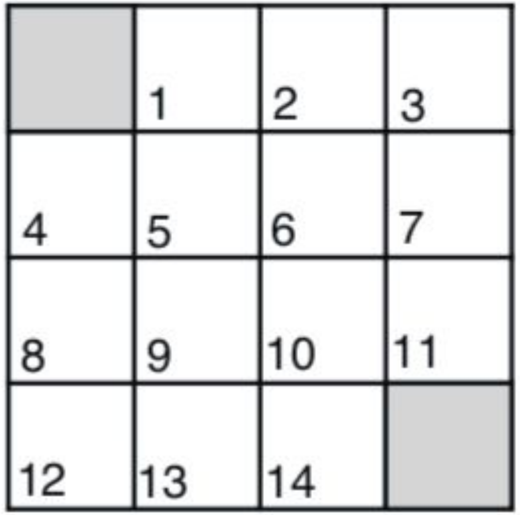
A policy gives the “recipe” for the agent to follow (or to sample from).

One simple policy is random uniform - each action in each state is chosen with the same probability (and probabilities sum up to 1.0).

If the policy is not **stochastic**, ie. doesn’t involve probabilities, it is said to be **deterministic**. Note that the environment may still operate in stochastic way (the result of taking an action is not always the same).

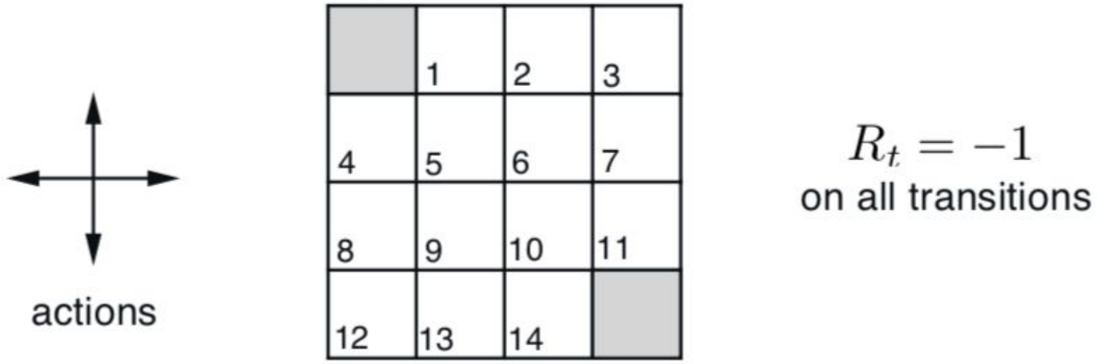
MDP example: Gridworld





$R_t = -1$
on all transitions

Example: Gridworld policy



Random walk policy:

$\pi(\uparrow \mid s) = 0.25$

$\pi(\rightarrow \mid s) = 0.25$

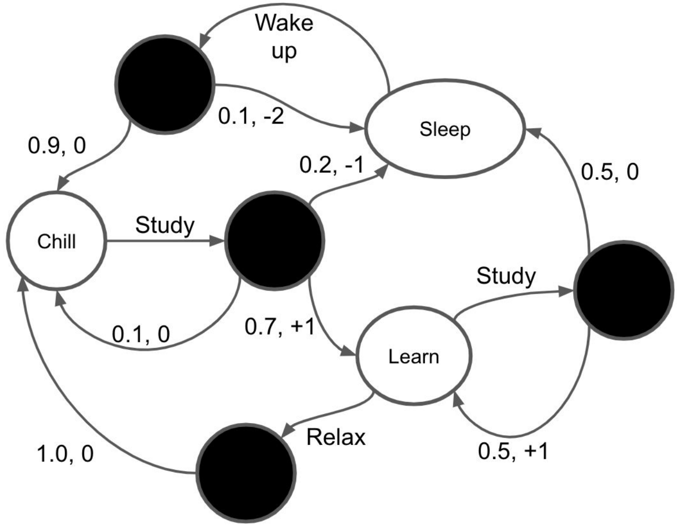
$\pi(\downarrow \mid s) = 0.25$

$\pi(\leftarrow \mid s) = 0.25$

For all $s \in \{ 1, \dots , 14 \}$

Note that the policy is stochastic, but in this case the MDP is not: movements are deterministic, the environment does not disturb the movements.

Example: another student life policy



Agent has only one state where more than one action is available, example policy:

π(Study | Learn) = 0.6

π(Relax | Learn) = 0.4

π(Study | Chill) = 1.0 etc

Value functions

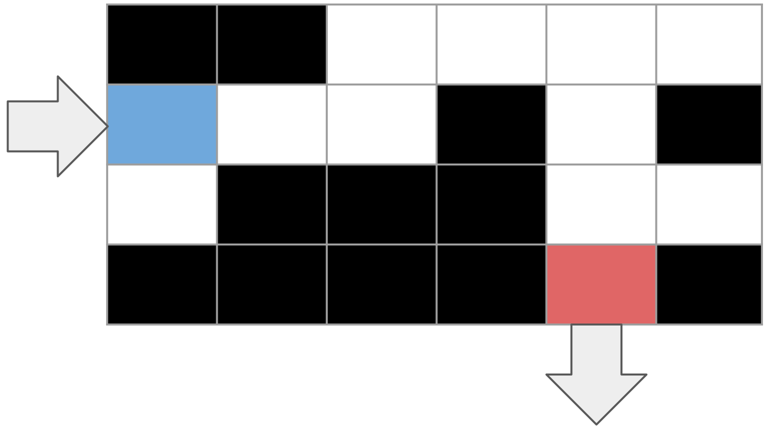
State-value function $v_\pi(s)$ $\overset{\text{def}}{=} \mathbb{E}(G_t \mid s = S_t)$ is the expected return when starting from state s and following policy π.

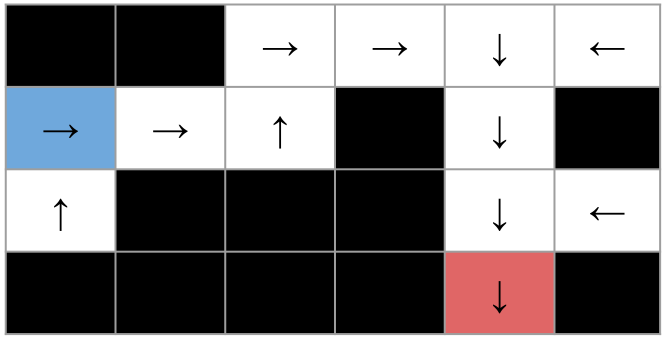
State-value function gives the answer to the question “how good is this state under policy π”.

Action-value function $q_\pi(s, a)$ $\overset{\text{def}}{=} \mathbb{E}(G_t \mid s = S_t, a = A_t)$ is the expected return when starting from state *s*, taking action *a*, and after that following policy π.

Action-value function gives the answer to the question “how good is it to take action *a* in state *s*, and then follow policy π”.

Example: Maze





States: all non-black squares

Actions: N, E, S, W. Action that would lead to black square or out of maze leaves state unchanged.

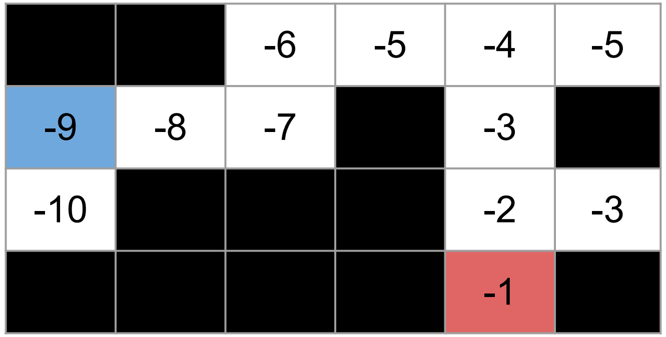
Rewards: -1

(Optimal and deterministic) policy π

Note: optimality will be defined later.

Example: Maze

State-value function $v_\pi(s)$



⬛	⬛	-6	-5	-4	-5
-9	-8	-7	⬛	-3	⬛
-10	⬛	⬛	⬛	-2	-3
⬛	⬛	⬛	⬛	-1	⬛

Action-value function $q_\pi()$

$q_\pi((2,1), \rightarrow) = -1 + (-8) = -9$

$q_\pi((2,1), \downarrow) = -1 + (-10) = -11$

...

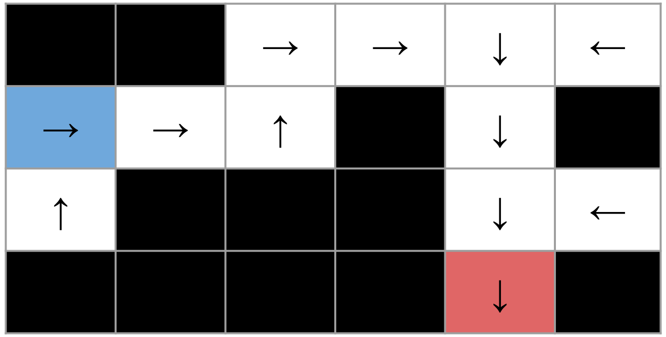
$q_\pi((1,5), \downarrow) = -1 + (-3) = -4$

$q_\pi((1,5), \leftarrow) = -1 + (-5) = -6$

$q_\pi((1,5), \rightarrow) = -1 + (-5) = -6$

$q_\pi((1,5), \uparrow) = -1 + (-4) = -5$

...



⬛	⬛	→	→	↓	←
→	→	↑	⬛	↓	⬛
↑	⬛	⬛	⬛	↓	←
⬛	⬛	⬛	⬛	↓	⬛

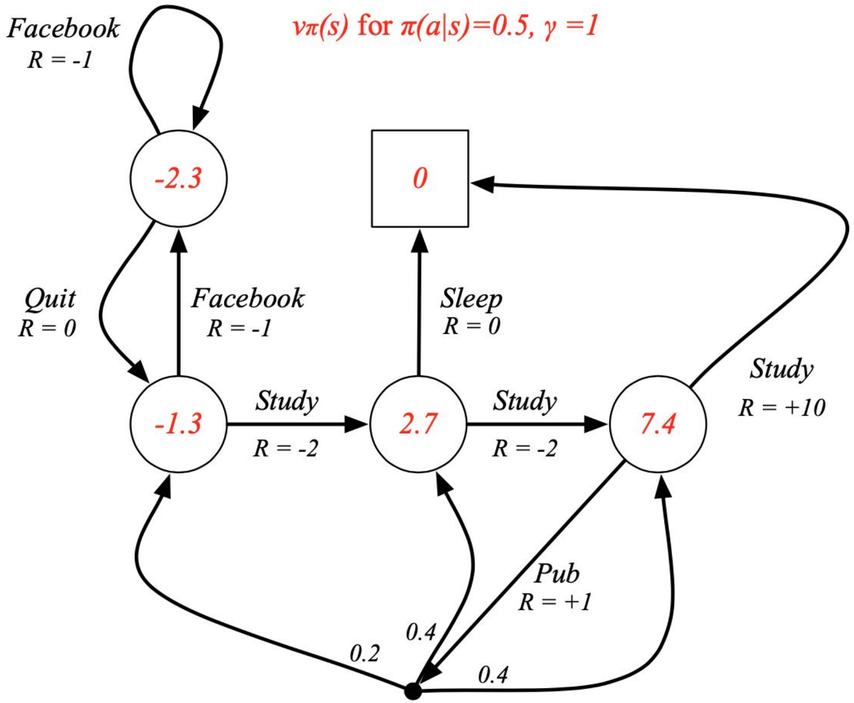
(What is expectation?)

Expectation of a set of random variables is the sum of variable values weighted with their probabilities:

E(1,2,3), where p(1)=0.5, p(2)=0.25, p(3)=0.25 is 0.5*1 + 0.25*2 + 0.25*3 = 1.75

If all probabilities are equal, expectation = mean.

State-value function example: student life

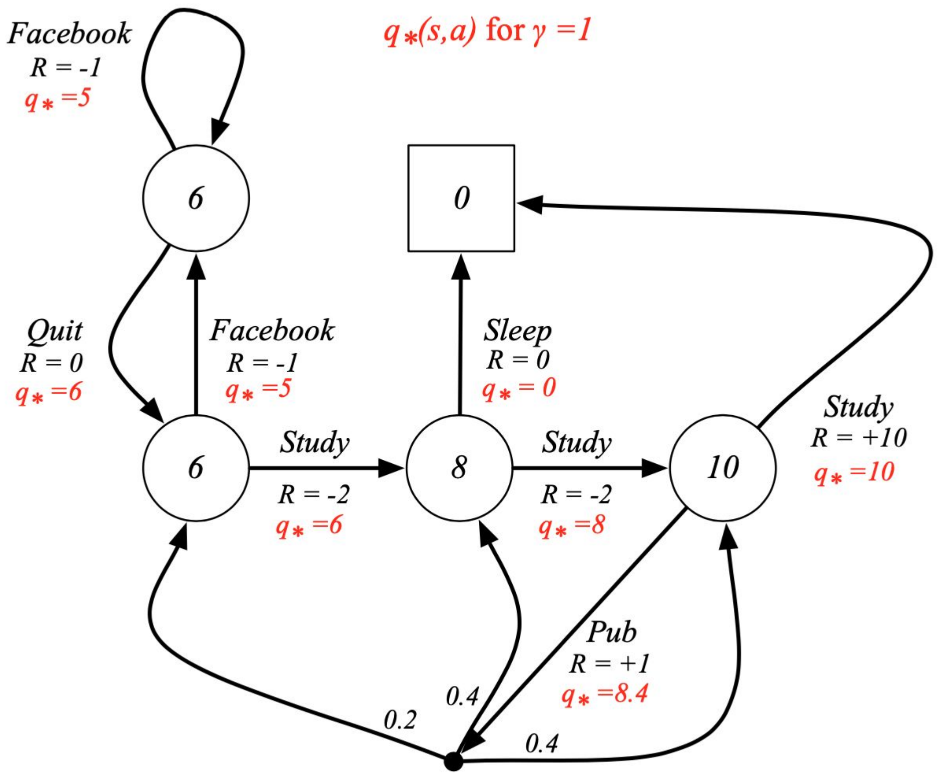


Policy in this example is random uniform: in each state there are two possible actions, and probability for both of them to be selected is 0.5.

Each of the states (open circles and box) have a value that gives a measure of how good it is to be in that state.

From Reinforcement Learning course slides by David Silver (https://www.davidsilver.uk/teaching/)

Action-value function example: student life



In this example the action-value function q is optimal (more on optimality later).

Each of the arrows leaving a state (actions) have value for function q.

From Reinforcement Learning course slides by David Silver (https://www.davidsilver.uk/teaching/)

Model-based learning and Bellman equations

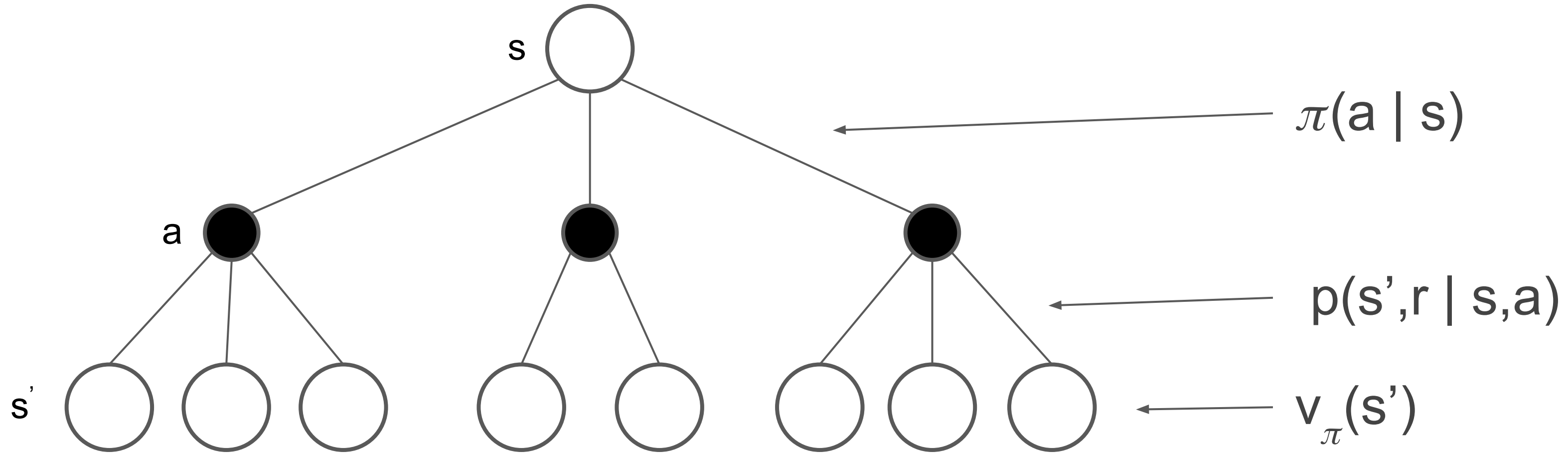
Model-based learning

Let’s take a look at how to evaluate and optimise a known MDP. This is, of course, an unrealistic setting from general reinforcement learning point of view. It turns out, however, that the concepts and techniques in model-based learning are useful in more realistic settings.

The problems we’ll take a look at are:

- We know MDP (S, A, P, γ) and policy π; what is the **value function $v_\pi(s)$**?
- We know MDP (S, A, P, γ); what is the **optimal value function $v_*(s)$** and corresponding **optimal policy $\pi_*(a \mid s)$**?

Backup diagram for state-value function



$$v_\pi(s) = \textstyle\sum_a [\pi(a \mid s) \sum_{s',r} p(s',r \mid s,a)(r + \gamma\, v_\pi(s'))]$$

(Bellman equation for state-value function)

(Write the expectation as a sum)

$$v_\pi(s) = \mathbb{E}_\pi(R_{t+1} + \gamma\, v_\pi(S_{t+1}) \mid s = S_t) =$$

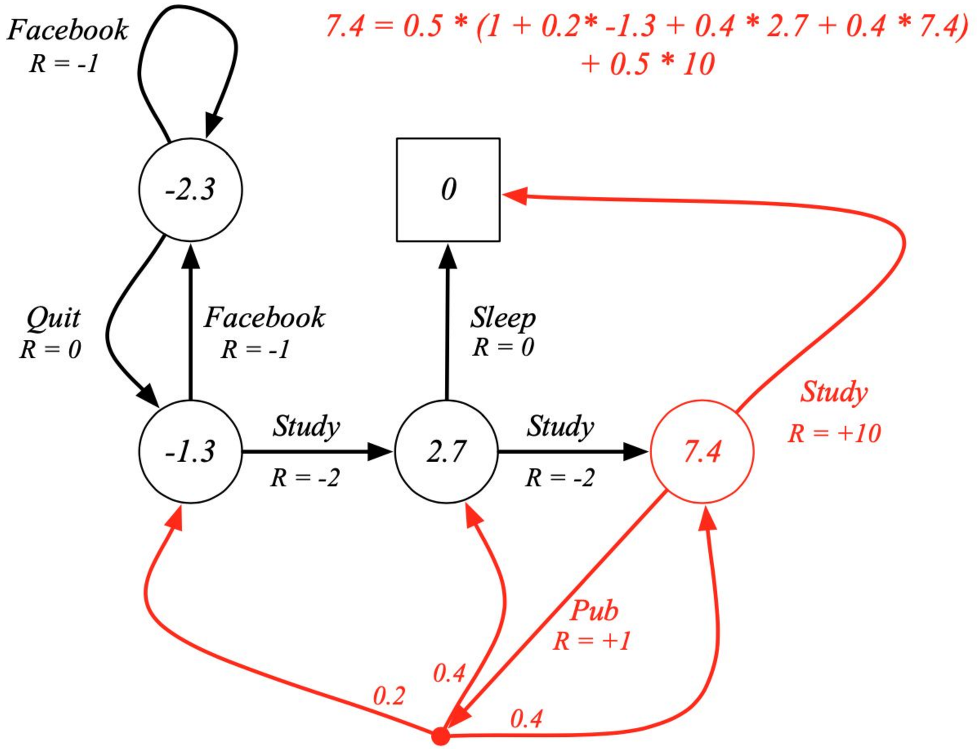
$$\textstyle\sum_{a \in A} \pi(a \mid s)\, [r(a, s) + \gamma \sum_{s' \in S} (p(a, s, s')\, v_\pi(s'))]$$

First multiplier in the sum: probability of taking action *a* in state *s* when following policy π. To compute expectation we multiply this with the value term.

Reward for taking action *a* in state *s*.

Probability of ending up in state *s’* when taking action a in state *s*. When multiplied with the value of state *s’* and summed up, this gives the expected value.

Bellman expectation equation example



Policy evaluation

Target: compute the state-value function for a given policy.

Idea: let’s turn the Bellman equation for state-values into iteration:

Compute new state values (for all states) using state-values from previous iteration in the right-hand side of Bellman expectation equation. Start iteration from an initial function, for example v(s) = 0 for all s:

$$\mathbf{v}_{\mathbf{k+1}}(\mathbf{s}) = \sum_{a \in A} \pi(a \mid s)\, [r(a, s) + \gamma \sum_{s' \in S} (p(a, s, s')\, \mathbf{v}_{\mathbf{k}}(\mathbf{s'}))]$$

Here v is a function from state-vector to state-vector.

Policy evaluation

0.25 * (-1 + (-2)) +
0.25 * (- 1 + (-2)) +
0.25 * (- 1 + (-2)) +
0.25 * (- 1 + (-1.75)) =
-2.9375

$k = 0$

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

Limited precision:
1.7 in (1,2) is
actually 1.75

$k = 2$

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

$k = \infty$

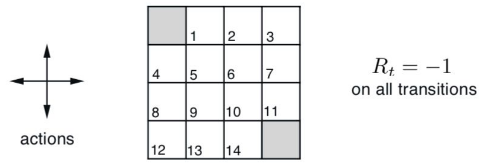
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Upper left corner and lower right corner are the end states (no actions taken there).

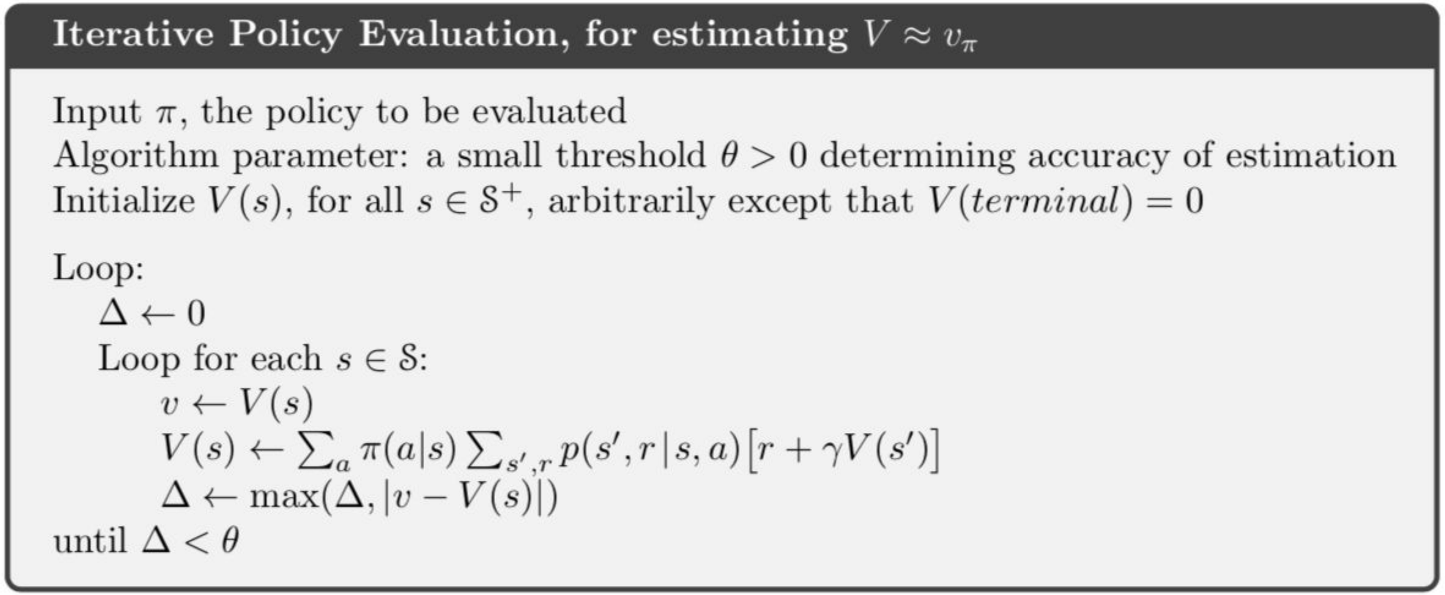
Actions are movements to N,E,S,W - if not possible to move, will stay in place.

Reward is -1 for all actions, and discount factor = 1. All actions are deterministic.

Policy is random movement N,E,S,W with equal probabilities.



Policy evaluation



Iterative Policy Evaluation, for estimating $V \approx v_\pi$

```
Input $\pi$, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
    $\Delta \leftarrow 0$
    Loop for each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) \big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$
```

Policy iteration

Optimal policy

We define (partial) ordering of policies with help of state-value function (partial) ordering:

$\pi \geq \pi'$ if for all s $\in$ S: $v_{\pi}(s) \geq v_{\pi'}(s)$

For **any MDP** there exists an optimal policy π_* (not necessarily unique) such that

$\pi_* \geq \pi$ for all π

For all optimal policies π_* the corresponding state-value and action-value functions are equal.

Optimal value functions

The state-value function always depends on policy, thus notation $v_\pi(s)$. When we are looking for the optimal v, we'll let π vary and pick the maximum v:

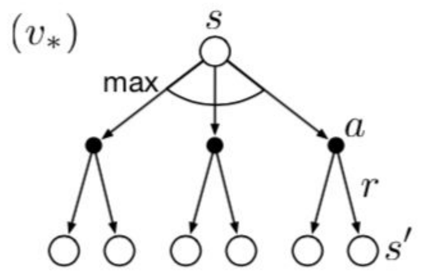
$$v_*(s) = \max_\pi v_\pi(s)$$

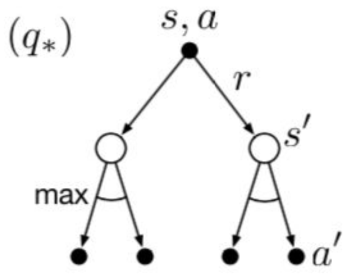
In the same way, the optimal action-value function is defined:

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

When we know v_* or q_* we say that the MDP is solved. By knowing v we can look ahead one step and select the best action (the one leading to highest-valued state); if we know q, we can just select the action with highest future return.

Bellman optimality equations





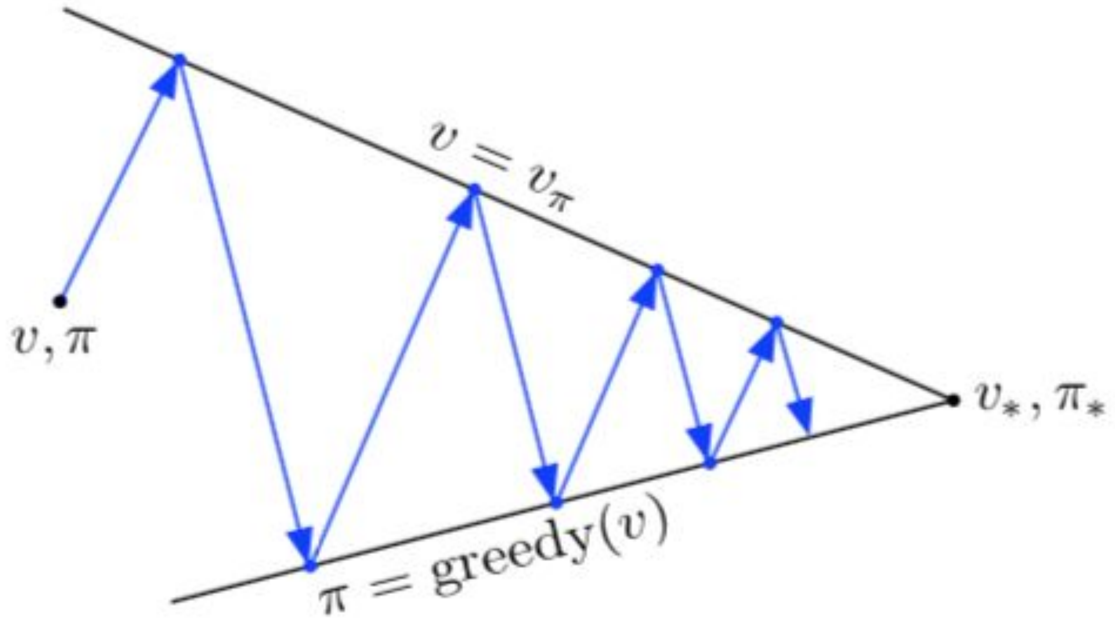
$v_*(s)$ $= \max_{a \text{ in } A(s)} q_*(s,a)$

$q_*(s,a)$ $= r + \gamma \sum_{s' \in S}(p(a, s, s') \max_{a' \text{ in } A(s')} q_*(s',a'))$

Bellman optimality equations are non-linear - there are no general (analytical) solutions.

Iterative methods must be used, we'll take a look at **policy iteration**. (There are more efficient methods like Q learning and Sarsa).

Policy iteration



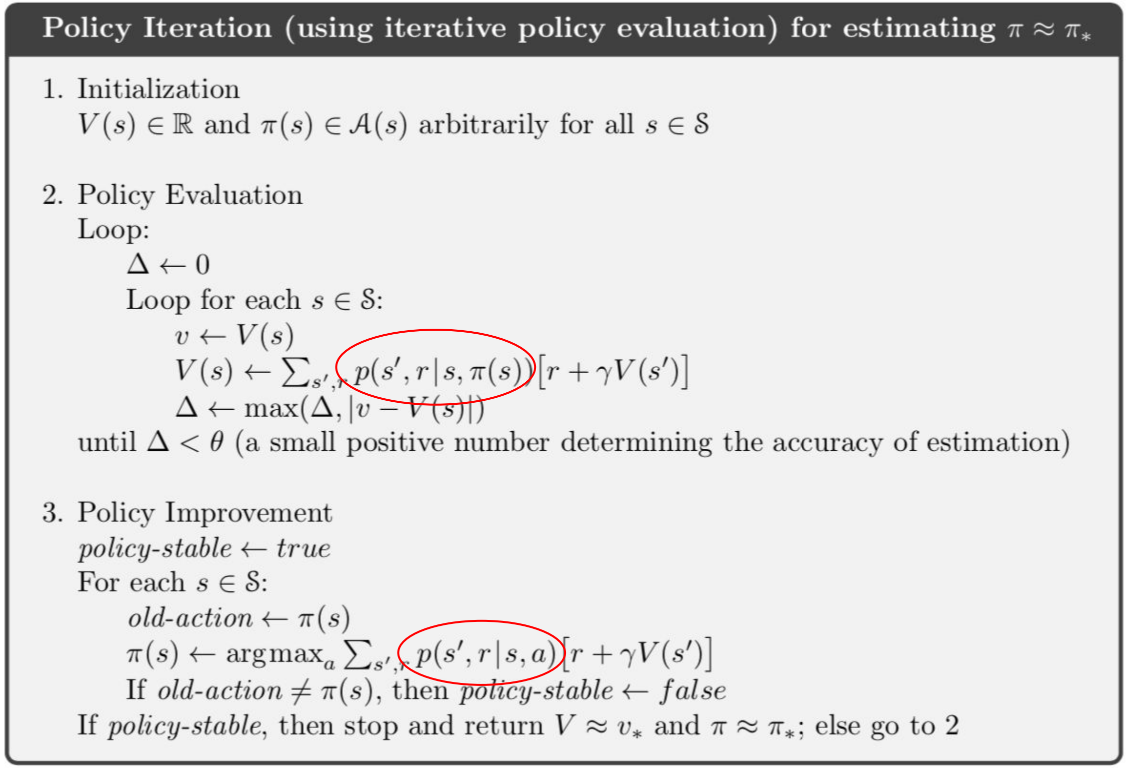
Idea of policy iteration:

Starting point: policy π (can be for example random uniform)

Repeat:

- **Evaluate** policy: compute the v corresponding to policy π
- Define new policy π: act **greedily** on v, ie. at each state s select the action with highest expected return

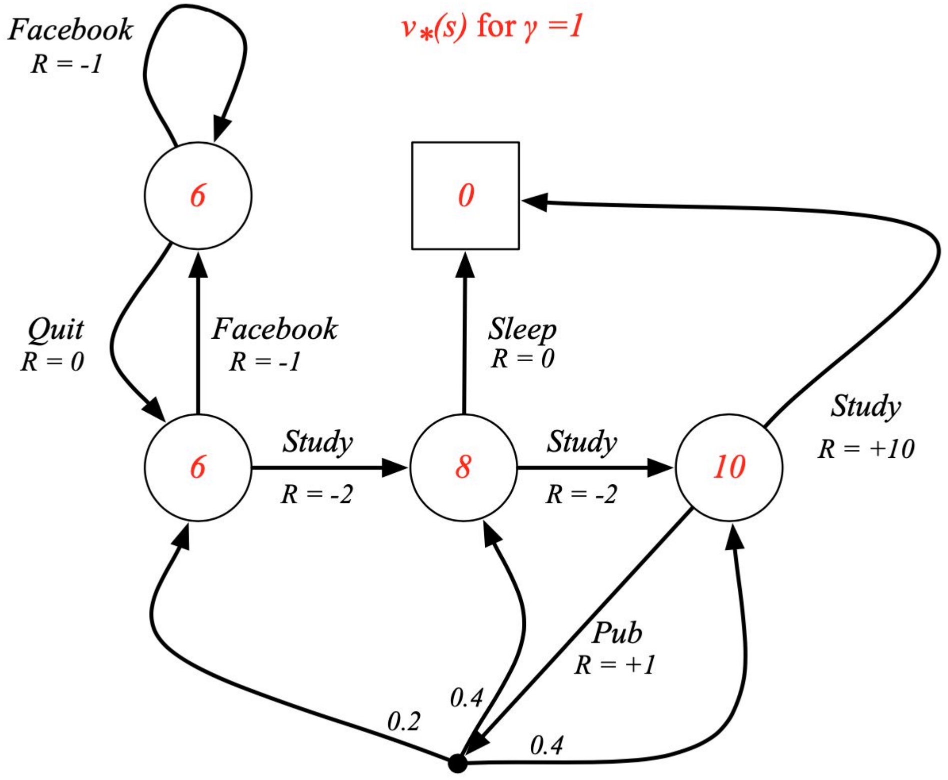
Policy iteration



Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

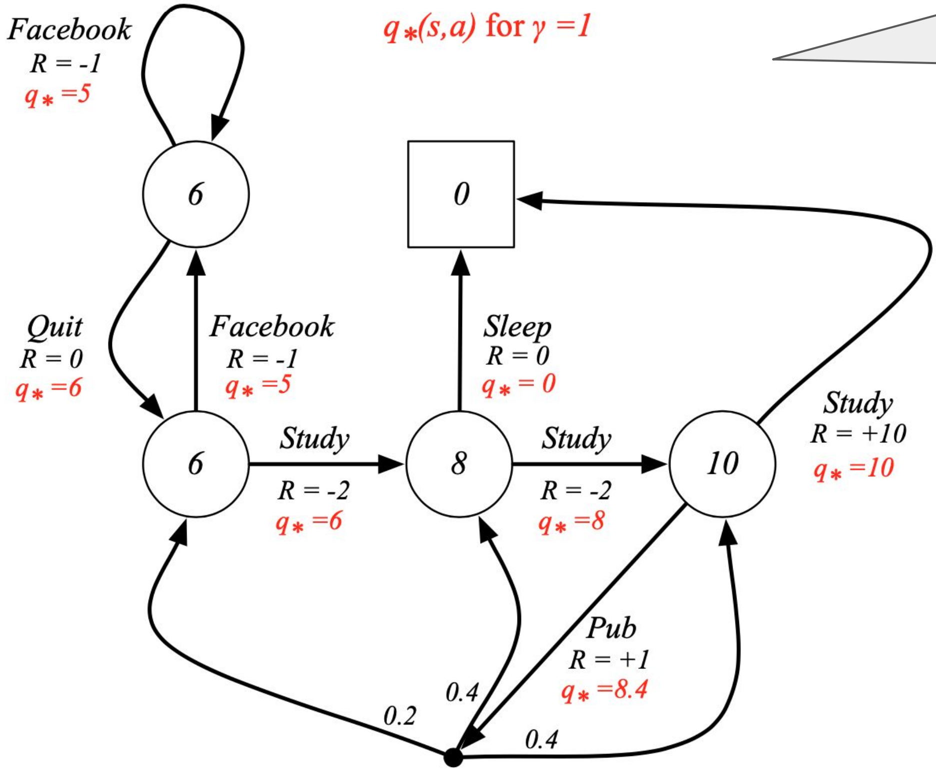
```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} \boxed{p(s',r|s,\pi(s))}[r + \gamma V(s')]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
       $old\text{-}action \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} \boxed{p(s',r|s,a)}[r + \gamma V(s')]$
       If $old\text{-}action \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

Optimal state-value function example

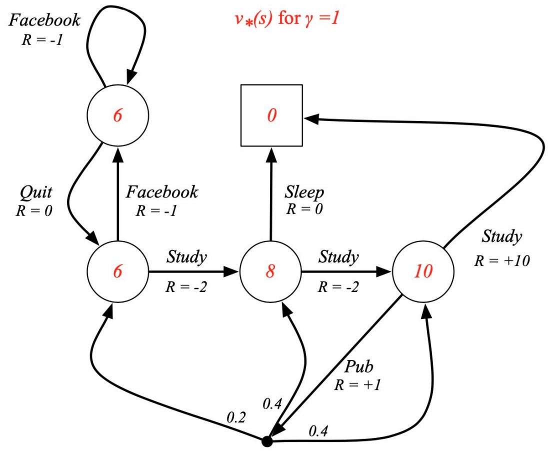


State-value function when acting according to optimal policy.

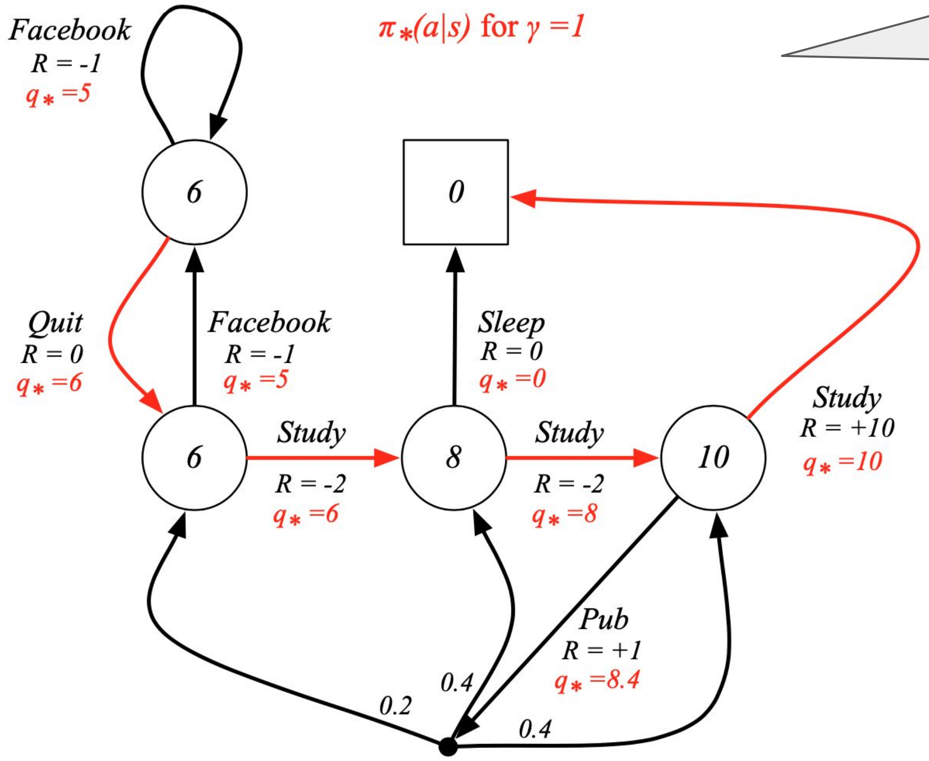
Optimal action-value function



At each state compute over all actions sum of reward and the optimal state-value function.



Optimal policy



At each state select the action with the highest action-value function.

Monte Carlo model-free learning

Model-free learning

So far we have assumed that the MDP is known, ie. we can consult the state-transition and reward functions when needed.

This is, however, usually an unrealistic assumption. Instead, we don’t have knowledge of the environment. To be able to operate in the environment in a way that maximises the cumulative reward, we need to learn.

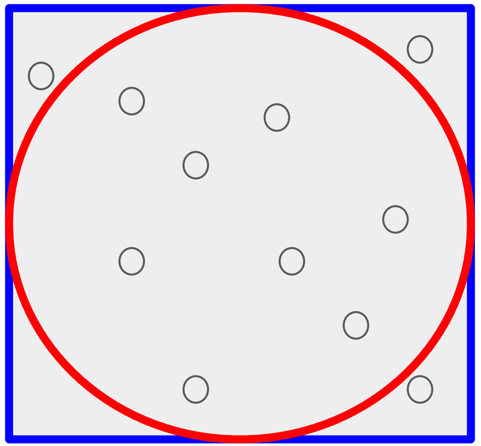
We could aim to learn the model, ie. understand exactly how the environment operates, or we can only learn state-value function, or action-value function, or policy. Note that the assumption still is that there is an underlying MDP.

We’ll take a (brief) look at Monte Carlo. There is much much more!

Monte Carlo methods

Many (computationally hard) problems can be approximately solved by using randomisation. In case of Monte Carlo, random sampling is often used and the result improves as more samples are gathered.

An example: approximation of π (the real pi, not policy). Random samples are generated to approximate the area of square and circle, and π = 4 * A(circle) / A(square)



$\pi \approx 4 * 8/11 = 2{,}91$

```
100000 :  3.143
200000 :  3.14682
300000 :  3.14852
400000 :  3.1467
500000 :  3.14584
600000 :  3.1452666666666667
700000 :  3.1447885714285713
800000 :  3.14342
900000 :  3.14244
```

Evaluating value function for an unknown MDP

MDP is not known; need to collect knowledge about its behaviour empirically.

Idea: start at some state (or the designated start state) and run episode (we now look at episodic tasks only, so they are known to terminate) following the policy π to be evaluated and observe what happens. Use the collected information to estimate how MDP works. Keep doing this.

What information could be observed? Rewards perhaps, and trying to estimate the value function (expected total return) would surely be useful?

The type of learning where we are following policy π and aim to improve that same policy is called on-policy learning. Off-policy learning is following a policy and learning about another policy at the same time.

Approximating value function

By running multiple episodes we'll get multiple samples of real value function values for each state - replace the expectation in value function definition with the mean over episodes (the empirical mean).

The cost of estimating value function for some state of interest s encountered when following policy π is in general independent of the number of states - it's the number of episodes that counts.

Monte Carlo action value estimation

Reminder: action value function q(s,a) is the value of taking action a plus the expected value of being in state s (under some policy).

To estimate q(s,a) instead of v(s) the immediate reward + future value estimate needs to be computed for each (state, action) pair. If there are n states, and m possible actions in each state, this means that n * m values need to computed.

The computation for a single (s,a) pair is very similar to that for a single state s.

However, we now have even more values to be estimated than in value function estimation - how to make sure we find all of them?

Exploration vs. exploitation

To address the questions in the previous slide, let’s consider how much should one

- **Explore** - look for better courses of action
- **Exploit** - do what is know to work best

Do you always eat at the same restaurant and enjoy predictably good dinner, ie. exploit? Or do you sometimes try out different places, even if it might be that you get disappointed, ie. explore?

Let’s mix exploration and exploitation and call the method ϵ **-greedy**.

ϵ -greedy exploration

Assume *m* actions are possible at state *s*, then *a* = argmax*$_{a \text{ in } Actions(s)}$ *q(s, a)* is the best action, the one that would be selected when acting greedily.

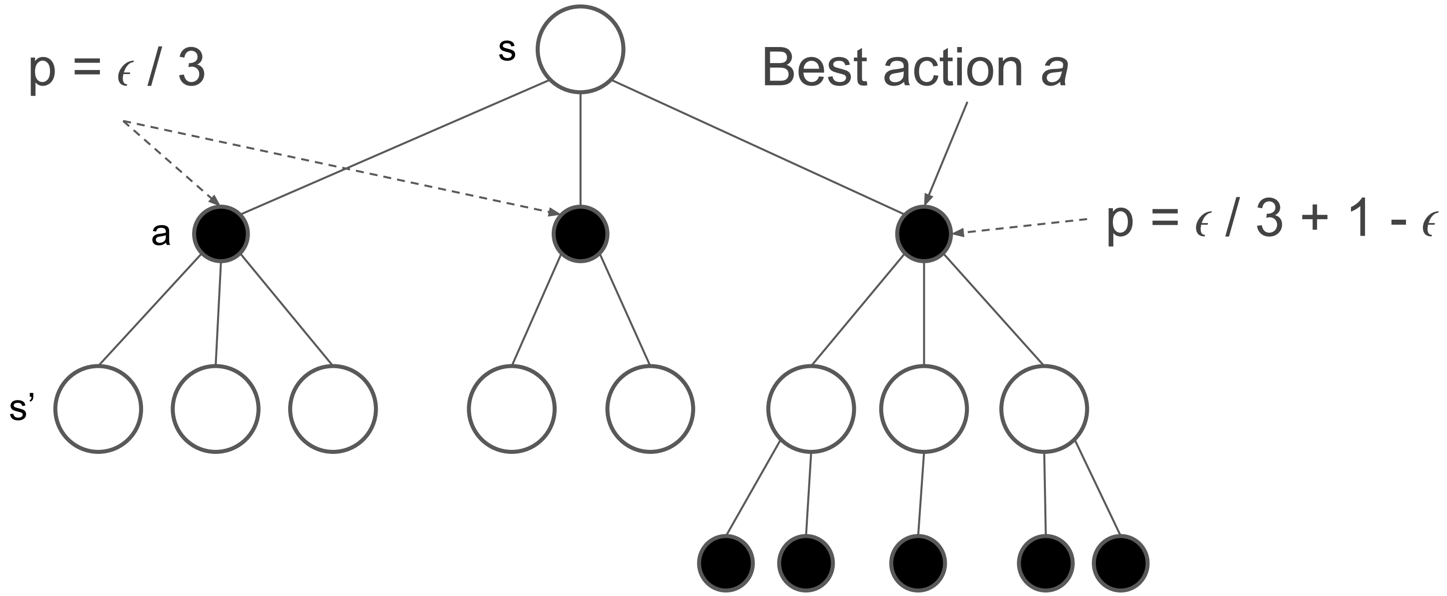
In ϵ -greedy exploration **a random action is selected with probability ϵ and the greedy action *a** with probability 1 - ϵ**. The policy will look like:

π(a | s) = ϵ / m + 1 - ϵ,   if a = a*

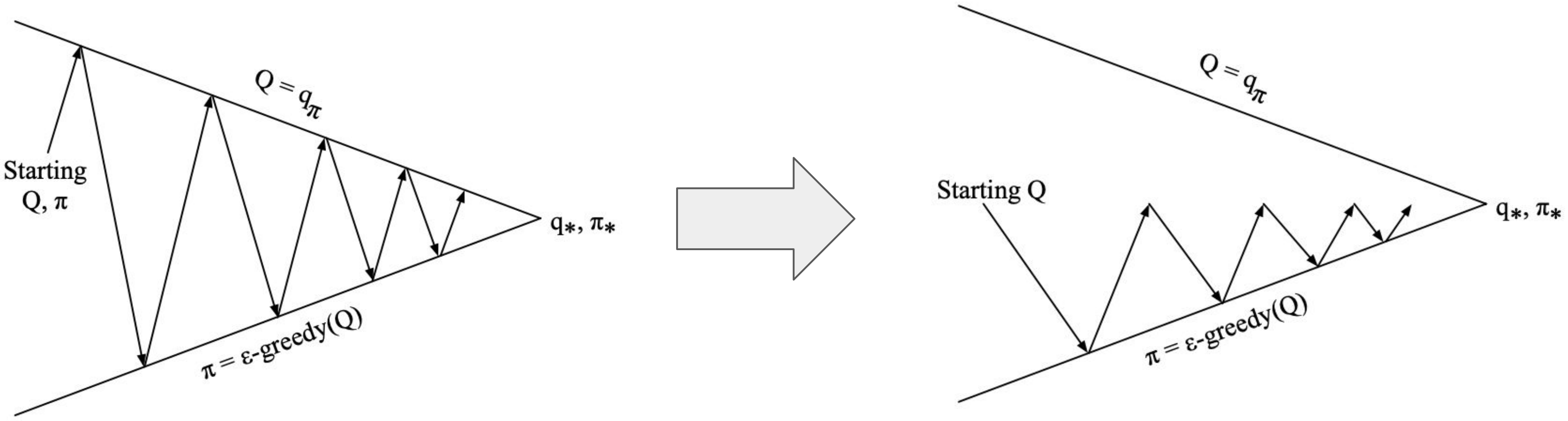
π(a | s) = ϵ / m,   if a ≠ a*

where m is the number of actions available in state s. (Note: often the policy itself is not explicitly needed.)

MC and ϵ -greedy exploration

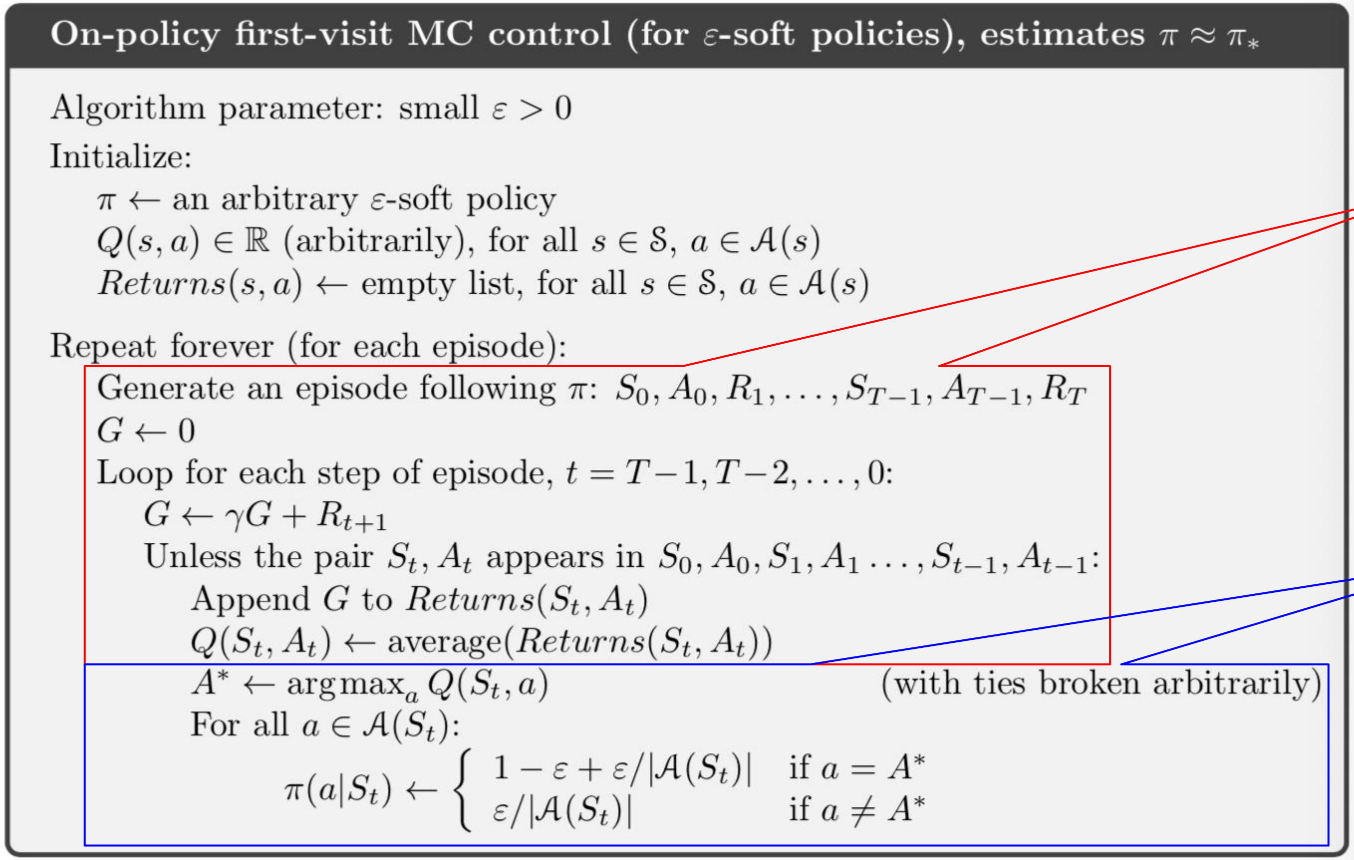


Monte Carlo policy iteration



No need to fully evaluate policy before improving it - improve after each episodic sample.

First-visit MC control



On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

```
Algorithm parameter: small $\varepsilon > 0$
Initialize:
    $\pi \leftarrow$ an arbitrary $\varepsilon$-soft policy
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$
            $A^* \leftarrow \arg\max_a Q(S_t, a)$                    (with ties broken arbitrarily)
            For all $a \in \mathcal{A}(S_t)$:
                $\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
```

Evaluate Q_π from the information gained by running one episode, using first-visit strategy.

Improve π. Note that storing π explicitly is typically not needed

Summary of MC for learning v or q

Run a number of episodes from a state, collect rewards earned, and compute the average return (discount rewards).

Balance exploration with exploitation by acting in ϵ -greedy way. This ensures all actions have a possibility to be selected.

MC only works in terminating, episodic, environments.

MC is not the most advanced way of doing this. TD-learning and SARSA are two methods that are more popular.

Need for function approximator

In the methods described so far the assumption has been that the state-value function or state-action function are fully specified for all states, or state-action pairs. This is unrealistic:

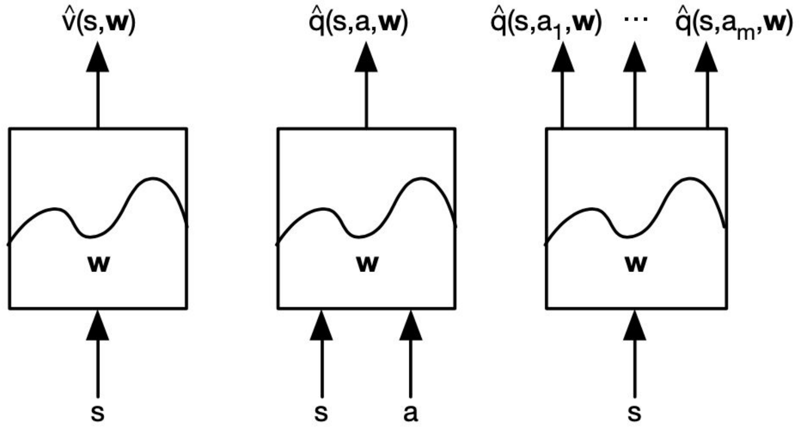
- The number of states / state-action pairs in realistic settings are huge (backgammon 10^{20}, other games much more, continuous value cases uncountable infinite)
- Even if the value could be stored, exploring enough to get v or q function value estimates would take impossibly long time

Function approximators

To overcome the state space size problem, v and q functions can be approximated with

- Linear, or other shallow models
- In more complicated cases by neural networks

The idea is to train a (parametric) model that represents the value function:



$\hat{v}(s,\mathbf{w})$
$\hat{q}(s,a,\mathbf{w})$
$\hat{q}(s,a_1,\mathbf{w})$
$\cdots$
$\hat{q}(s,a_m,\mathbf{w})$
w
w
w
s
s
a
s

Value function approximation - incremental

In incremental methods value function is trained (for example Monte Carlo policy iteration, this can be done with other methods also):

- Use the return of an episode as the “true value” and adjust parameters of function approximator using gradient descent
- The training data for the approximator is collection of pairs (S_i, G_i), ie. returns G_i when an episode is started at S_i
- Same method can be used to learn q function - training data is then (S_i, A_i, G_i)

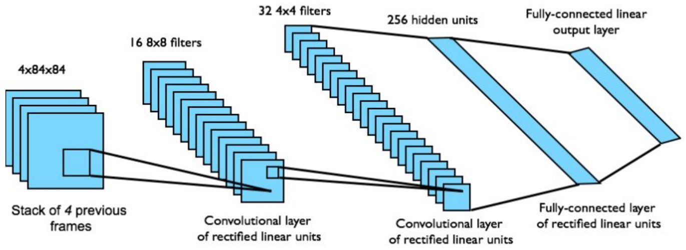
Value function approximation - batch

In iterative methods the samples collected are discarded after they are used.

Store experience in replay memory. Sample a mini-batch from experience and use stochastic gradient descent to adjust the function approximator parameters.

This approach is used in DQN (Atari games system):

- State s is 4 frames of raw pixels (the game screen)
- A convnet is trained to predict q function, ie. value for each action (up,down,shoot, etc)



4x84x84
16 8x8 filters
32 4x4 filters
256 hidden units
Fully-connected linear output layer
Stack of 4 previous frames
Convolutional layer of rectified linear units
Convolutional layer of rectified linear units
Fully-connected layer of rectified linear units

Summary

Agent and environment: interaction with actions, rewards, and observations (states).

MDP for describing the dynamics, state-value function v and action-value function q for the MDP with given policy.

Learning optimal policy when MDP is given.

Learning optimal policy in model-free way with Monte Carlo.

Using deep nets to approximate value functions.

Links

Sutton & Barto: Reinforcement Learning, 2nd edition. Available at http://incompleteideas.net/book/the-book-2nd.html

Videos from a reinforcement learning course given by David Silver. Recommended! https://www.youtube.com/watch?v=2pWv7GOvuf0