Markov Decision Processes (MDPs) and Reinforcement Learning (RL)

Alvaro Soto

Computer Science Department (DCC), PUC

Intro

- So far, we have mainly discussed passive inference engines. An agent receives information from the environment and makes some inference.
- In this part of the class, we will consider the case of an active agent. An agent takes/executes decisions/actions that affect the state of the world.
- We will assume a Markovian World, i.e., decisions/actions are independent of the past given knowledge of the current state of the world.

Markovian Models of Perception and Action

	Pasive	Active
Total observability	MC	MDP
Partial observability	HMM	POMDP

Learning a policy: Planning to act in the world

- **Main goal**: learn how to behave in the world, i.e., learn a policy or plan.
- This policy or plan consist of a mapping from states of the world to "suitable" (hopefully optimal) actions.
- Typical operation:
  0. Using knowledge about the world W and current goal G, agent builds a plan π.
  1. Using sensor data, agent accesses full or partial knowledge about the current state of the world S.
  2. Given S, agent executes the action suggested by its learned policy/plan π.
  3. Repeat steps 1 and 2 until reaching goal state G.
- Relevant complication: Agent decisions influence the environment, therefore, agent influences the training experience.

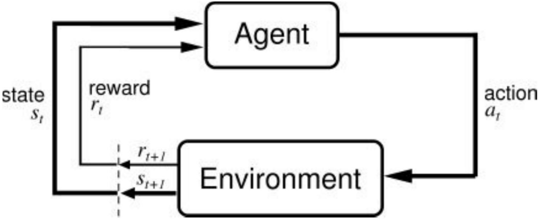
How can we learn an action policy?

Main idea: Reinforcement Learning (RL)

By considering the result of the interaction with the environment in terms of reward/cost, an agent can learn how to act in the world to maximize/minimize this reward/cost

- Inspired by biological behavior: carrot-and-stick strategy.
- Is it RL the same that supervised learning ?

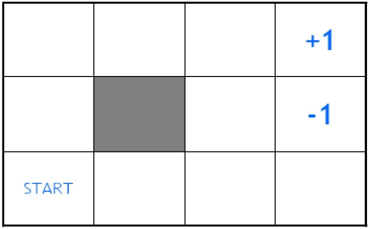
Reinforcement Learning (RL)



- Training experience consists of a sequence of reward values.
- Specifically, training experience is given by a sequence of reward $r_t(s_i, a_i)$, i.e., a reward value at time t when the agent/robot is at state s_i and takes action a_i.
- Usually, the agent has to take decisions considering delayed rewards. Ex. *you should study hard now, so you will be very proud at the end of the semester*. This makes the problem hard.

Ex. Robot Navigation

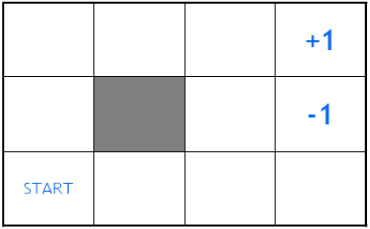
- Actions: up, down, left and right. One cell each time.
- Uncertainty in action execution: 70% success, 10% chances for each of the wrong directions.
- $$\text{Reward} = \begin{cases} +1 & \text{at [3,4]} \\ -1 & \text{at [2,4]} \\ -0.04 & \text{each time step.} \end{cases}$$

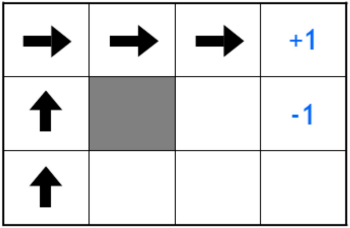


Ex. Robot Navigation

- Actions: up, down, left and right. One cell each time.
- Uncertainty in action execution: 70% success, 10% chances for each of the wrong directions.
- $$\text{Reward} = \begin{cases} +1 & \text{at [3,4]} \\ -1 & \text{at [2,4]} \\ -0.04 & \text{each time step.} \end{cases}$$

Is this an optimal path?

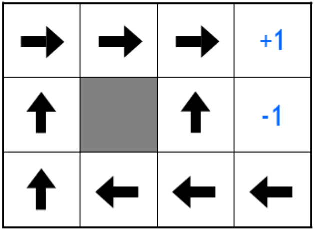




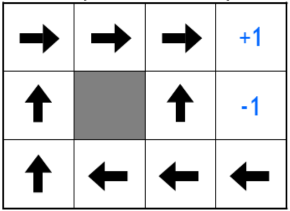
Ex. Robot Navigation

- Actions: up, down, left and right. One cell each time.
- Uncertainty in action execution: 70% success, 10% chances for each of the wrong directions.
- $$\text{Reward} = \begin{cases} +1 & \text{at [3,4]} \\ -1 & \text{at [2,4]} \\ -0.04 & \text{each time step.} \end{cases}$$

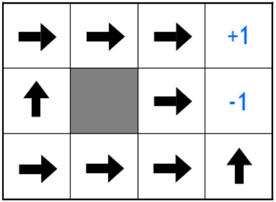
Universal Planner



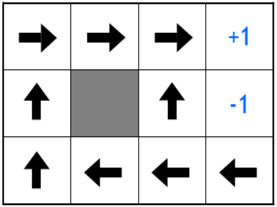
Optimal Policy



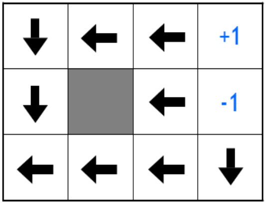
Optimal Policy, if step -2



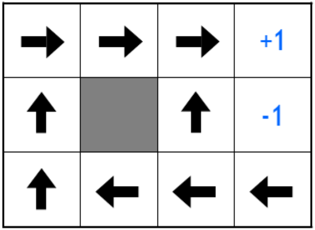
Optimal Policy, if step -0.04



Optimal Policy, if step 0.01



Universal Planner



- We need to find a policy.
- Policy: a suitable/optimal way to act in any state of the world.

How can I find the optimal policy?

How can I find the "optimal" policy?

Big result: if we can model our problem as a MDP, we can indeed find an optimal policy.

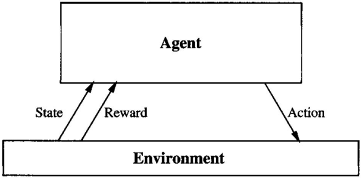
- A MDP is composed of:
  - A finite set of states : $s_1, \ldots, s_n$
  - A set of rewards: $r_1, \ldots, r_m$.
  - A set of actions: $a_1, \ldots, a_l$.
  - A set of transition probabilities between states: $P_{ij}^k = P(s_j | s_i, a_k)$.

- A MDP is composed of:
  - A finite set of states : $s_1, \dots, s_n$
  - A set of rewards: $r_1, \dots, r_m$.
  - A set of actions: $a_1, \dots, a_l$.
  - A set of transition probabilities between states: $P_{ij}^k = P(s_j | s_i, a_k)$.

Environment:	You are in state 65. You have 4 possible actions.
Agent:	I'll take action 2.
Environment:	You received a reinforcement of 7 units. You are now in state 15. You have 2 possible actions.
Agent:	I'll take action 1.
Environment:	You received a reinforcement of -4 units. You are now in state 65. You have 4 possible actions.
Agent:	I'll take action 2.
Environment:	You received a reinforcement of 5 units. You are now in state 44. You have 5 possible actions.
$\vdots$	$\vdots$

Reinforcement Learning (RL)

To account for the uncertainty about future rewards, the model introduces a discount factor.



$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \ldots$$

Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots, \text{ where } 0 \leq \gamma < 1$$

Discount factor γ: takes care of the increasing uncertainty with respect to rewards that are distant in the future.

Value function: $V(s)$

- Let's define the value function $V(s)$ for a state s, as the total reward that the agent will receive starting at s and following policy π thereafter.

$$V(s) = E\left\{\sum_{t=0}^{\infty} \gamma^t r_t^{\pi}\right\}$$

- r_t^{π} denotes reward following polici π.
- Notice that we need to consider an expected value to account for the uncertatinty in action execution. Due to similar reason, we need to include a discount factor γ.

How can I find an optimal policy?

- We need to find policy π that maximizes the value function:

$$V^*(s) = \max_{\pi} E\left\{\sum_{t=0}^{\infty} \gamma^t r_t^{\pi}\right\}$$

- We can optimizes this equation using dynamic programming. Specifically, we maximize Bellman's equation:

$$V^*(s) = \max_{a}\left\{R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')\right\}, \quad \forall s \in S$$

- We are really interested in the actions that maximize the value function, and therefore lead to the optimal policy:

$$V^*(s) = \arg\max_{a}\left\{R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')\right\}, \quad \forall s \in S$$

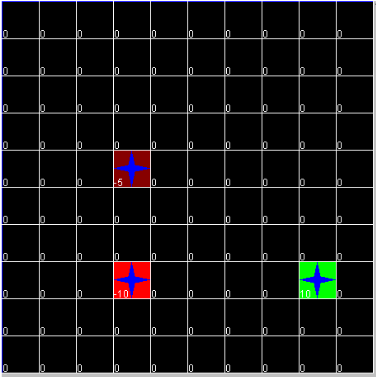
Value Iteration

- In case of finite state and action spaces, we can optimize Bellman's equation using an iterative optimization method known as the Value Iteration algorithm.
- Starting from random values of the value function, at each iteration this method improves the current estimation of these values $\hat{V}^*(s), \ \forall s \in S$.
- This procedure leads to convergence to the optimal values $\hat{V}^*(s)$ and the related optimal policy.

Value Iteration Algorithm

```
Initialize V(s) arbitrarily
loop until policy good enough
     loop for s ∈ S
           loop for a ∈ A
                Q(s,a) := R(s,a) + γ Σ_{s'∈S} T(s,a,s') V̂(s')
           end loop
           V̂(s) := max_a Q(s,a)
     end loop
end loop
return{V̂(s)}
```

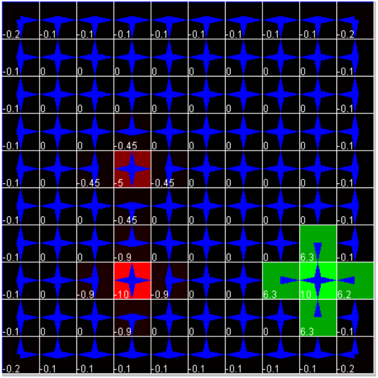
Example: Value Iteration



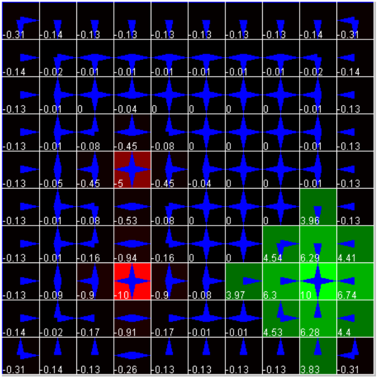
$$V^*(s) = \max_a \left\{ R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right\}, \quad \forall s \in S$$

Example: Value Iteration

Iteration 1



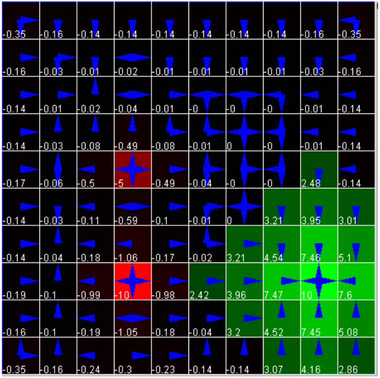
Iteration 2



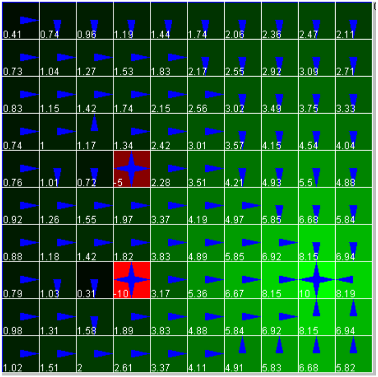
$$V^*(s) = \max_a \left\{ R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right\}, \quad \forall s \in S$$

Example: Value Iteration

Iteration 3



Iteration 10



$$V^*(s) = \max_a \left\{ R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right\}, \quad \forall s \in S$$

Policy Iteration

- As an alternative to the Value Iteration algorithm, we can optimize Bellman's equation using the Policy Iteration algorithm, an iterative optimization method that improves the policy directly.
- Starting from a random policy, at each iteration this method improves the current estimation of this policy by solving a linear set of equations:

$$V_\pi(s) := R(s,\pi(s)) + \gamma \sum_{s' \in S} T(s,\pi(s),s') V_\pi(s')$$

- Notice that this is a set of $|S|$ linear equations in $|S|$ unknown variables.

Policy Iteration Algorithm

```
Choose an arbitray policy π'
Loop
        π := π'
        Compute value function of policy π:
              #solve linear equations
              V_π(s) := R(s,π(s)) + γ Σ_{s'∈S} T(s,π(s),s') V_π(s')
        Improve the policy at each state
              π'(s) := argmax_a (R(s,a) + γ Σ_{s'∈S} T(s,a,s') V_π(s'))
until π = π'
```

Model Free Learning

Q-Learning

- In some cases, there is not model information, i.e., the agent does not know the transition function $P(s_{t+1}|s_t, a_t)$.
- Still, the agent can observe its current state s_t, it has knowledge about its executed action a_t, and it receives a reward r_t from the environment.
- In other words, agent still can collect training data consisting of experiences: (s_t, a_t, s_{t+1}, r_t), where we can also obtain reward value $r_t(s_t, a_t)$ and info about valid transition (s_t, s_{t+1}).
- However, given lack of knowledge about the underlying MDP model, we can't use value or policy iteration to find an optimal policy.
- Instead, we can learn the so-called Q-function that is defined as: $Q(s,a) = r(s,a) + \gamma V^*(succ(s,a))$, where $succ(s,a)$ returns the next state after executing action a at state s.
- So $Q(s,a)$ represents the maximum discounted cumulative rewards that the agent can be achieve by starting at state s, execute action a, and follow the optimal policy thereafter (maximizing value function).

How can we learn the Q-function ?

Update Q(s,a) every time an experience (s,a,s',r) is observed:

$$Q(s,a) = r(s,a) + \gamma V^*(succ(s,a))$$

At each state, optimal policy takes action that maximizes the Q-function, then:

$$Q(s,a) = r(s,a) + \gamma \arg\max_{a'} Q(succ(s,a),a')$$

Each update is independent of the policy being followed. The only requirement is to keep updating each pair (s,a). This suggests the following algorithm to infer the Q-function:

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s,a)$ to zero.
Observe the current state s
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s,a)$ as follows:

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s \leftarrow s'$

Smoothing the Q-function

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state s
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

$$Q(s, a) = r(s, a) + \gamma \arg\max_{a'} Q(succ(s, a), a')$$

Selection of action a can be noisy. Usually, we should add some smoothing to the estimation $\hat{Q}(s, a)$ of the Q-function:

$$\hat{Q}(s, a)^{new} = (1 - \alpha)\hat{Q}(s, a)^{old} + \alpha Q(s, a)$$

How to select action **a**?

Q learning algorithm

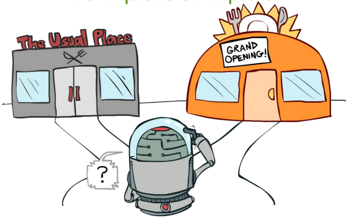
For each s, a initialize the table entry $\hat{Q}(s,a)$ to zero.
Observe the current state s
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s,a)$ as follows:

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s \leftarrow s'$

To Explore or Exploit?



Drawing by Ketrina Yim

How to select action **a**?

- ϵ-greedy exploration policy:

How to select action **a**?

- ϵ-greedy exploration policy:
  - With probability $1 - \epsilon$:

How to select action **a**?

- ϵ-greedy exploration policy:
  - With probability $1 - \epsilon$:
    - Choose the current optimal action: $\arg\max_a \hat{Q}(s, a)$.

How to select action **a**?

- ϵ-greedy exploration policy:
  - With probability $1 - \epsilon$:
    - Choose the current optimal action: $\arg\max_a \hat{Q}(s, a)$.
  - With probability ϵ:

How to select action **a**?

- ϵ-greedy exploration policy:
  - With probability $1 - \epsilon$:
    - Choose the current optimal action: $\arg\max_a \hat{Q}(s, a)$.
  - With probability ϵ:
    - Select a random action.

How to select action **a**?

- ϵ-greedy exploration policy:
  - With probability $1 - \epsilon$:
    - Choose the current optimal action: $\arg\max_a \hat{Q}(s, a)$.
  - With probability ϵ:
    - Select a random action.

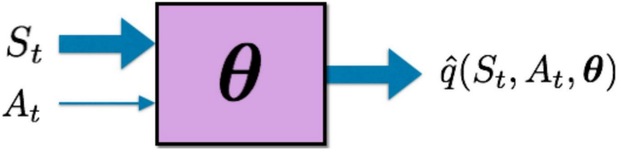
- Guaranteed to compute optimal policy.

How to select action **a**?

- ϵ-greedy exploration policy:
  - With probability $1-\epsilon$:
    - Choose the current optimal action: $\arg\max_a \hat{Q}(s,a)$.
  - With probability ϵ:
    - Select a random action.
- Guaranteed to compute optimal policy.
- Usually, we can use a scheme to decay ϵ over time (why?).

Continuous State Space

- So far we estimate V or Q using lookup tables.
  - Every state s has an entry $V(s)$, or
  - Every state-action pair (s, a) has an entry $Q(s, a)$.
- This is possible in a world with a finite number of states.
- However, in case of a large, possible infinite state space, we need to consider an alternative approach to estimate the Q-function.
- In particular, we need to replace the lookup table by a continuous function approximation, such as a parameterized estimation θ over a continuous time t.



Q-Learning: Continuous case

There are many possible options to obtain a continuous estimation of the Q-function, e.g.:

- Neural networks
- Decision tree
- Nearest neighbor
- Fourier / wavelet bases
- etc.

Today, one of the most popular approaches to estimate the Q-function is to use Deep Learning, ex.:

Deep Q-Networks (DQN),
Mnih et al. NIPS, 2014; Mnih et al. Nature 2015.