Efficient generalization in Partially Observable Markov Decision Processes: a case study

Gianvito Losapio

October 26, 2022

Contents

1 Introduction

1.1 Markov Decision Processes

Markov Decision Processes (MDPs) represent a mathematically idealized form of a specific class of machine learning problems called Reinforcement Learning (RL), which consist of learning from interaction to achieve a goal.

Thanks to MDPs, precise theoretical statements can be made about such a class of problems. The learner and decision maker is called the *agent*. The thing it interacts with, comprising everything outside the agent, is called the *environment*. These interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations to the agent. The environment also gives rise to *rewards*, special numerical values that the agent seeks to maximize over time through its choice of actions.

At each time step t, the agent receives some representation of the environment's state, $s_t \in \mathcal{S}$, and on that basis selects an action, $a_t \in \mathcal{A}(s)$. One time step later, in part as a consequence of its action, the agent receives a numerical reward, $r_{t+1} \in \mathbb{R}$ and finds itself in a new state, s_{t+1} (the next reward and next state, r_{t+1} and s_{t+1}, being jointly determined).

The MDP and agent together thereby give rise to a sequence or trajectory that begins like this:

$$\tau = \{s_0, a_0, r_1, s_1, a_1, r_2, \cdots\} \tag{1}$$

An MDP can be formalized as a tuple

$$\text{MDP} = \langle \mathcal{S}, \mathcal{A}, T, R \rangle \tag{2}$$

where:

- $\mathcal{S}$ is the set of environment states. To be Markovian, each state must encompass all the relevant features for making correct decisions. States can be completely determined by low-level sensations, such as direct sensor readings, or they can be more high-level and abstract, such as symbolic descriptions of objects in a room.

- $\mathcal{A}$ is the set of actions that the agent can execute. The actions can be low-level controls, such as the voltages applied to the motors of a robot arm, or high-level decisions, such as whether or not to have lunch or to go to graduate school. In general, actions can be any decisions we want to learn how to make, and the states can be anything we can know that might be useful in making them.

- T is the stochastic transition function, that is the probability of executing action a from state s at time $t-1$ and reaching state s' at time t

$$T(s, a, s') = \Pr(s_t = s' \,|\, s_{t-1} = s, a_{t-1} = a) \tag{3}$$

The transition function captures the probability of ending up in any possible successor state starting from a given state.

- R is the reward function, modeling both the utility of the current state, as well as the cost of action execution. $R(s, a, s')$ is the reward for executing action a in state s ending up in state s'. The reward signal is your way of communicating to the agent what you want it to achieve, not how you want it achieved. The agent's goal is to maximize the total amount of reward it receives. This means maximizing not immediate reward, but cumulative reward in the long run. This is defined in terms of return and a value function (described later).

The solution to an MDP is defined in terms of a *policy* that maps each state to a desirable action $\pi : \mathcal{S} \to \mathcal{A}$. Formally, a policy can be generally defined in stochastic terms, mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $a_t = a$ if $s_t = s$. The "|" in the middle of $(a|s)$ merely reminds that it defines a probability distribution over a $a \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$.

Discounted return The agent's goal is to maximize the cumulative reward it receives in the long run. More precisely, we seek to maximize the expected return, where the return, denoted G_t, is defined as some specific function of the reward sequence. The *discounted return* is the most common choice:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4}$$

where $0 \leq \gamma \leq 1$ is a parameter called discount rate. The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards. As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted.

Value function Almost all RL algorithms involve estimating value functions, i.e. functions of states that estimate how good it is for the agent to be in a given state. (with the notion of "how good" being defined in terms of discounted return). Once you have a good estimate of the value function, you can easily compute a policy (see section 6 for more details).

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s] \tag{5}$$

$$= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| s_t = s\right] \tag{6}$$

A fundamental property of the value function is that it satisfies the central dynamic programming recursive relation known as the *Bellman equation*:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \tag{7}$$

It states that for any policy π and any state s, the value function $v_\pi(s)$ equals the reward at the state s plus the discounted value function at the successor state s' (weighted by the corresponding probability).

Think of looking ahead from a state to its possible successor states, as suggested by the diagram of Figure 1(a). Each open circle represents a state and each solid circle represents a state–action pair. Starting from state s, the root node at the top, the agent could take any of some set of actions- three are shown in the diagram- based on its policy π. From each of these, the environment could respond with one of several next states, s' (two are shown in the figure), along with a reward, r, depending on its dynamics (given by a probabilistic function p). The Bellman equation (7) averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.

Optimal policy Finding a good solution to an RL problem means, roughly, finding a policy that achieves a lot of discounted return over the long run. Value functions define a partial ordering over policies. As a consequence, for finite MDPs we can precisely define an optimal policy in the following way.

A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy*. Although there may be more than one, we denote all the optimal policies by π_*. They share the same value function, called the *optimal value function*, denoted v_*, and defined as

$$v_*(s) = \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S} \tag{8}$$

The optimal value function assigns to each state the largest expected return achievable by any policy.

Because v_* is the value function for a policy, it must also satisfy the the Bellman equation. This is the Bellman equation for v_*, or the *Bellman optimality equation*:

$$v_*(s) = \max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_*(s')] \tag{9}$$

Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state. Figure 1(b) shows the corresponding diagram.

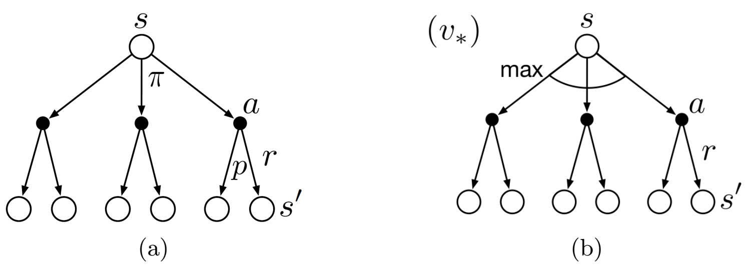


Figure 1: Backup diagram for (a) v_π, (b) v_*. Credits: [1]

For finite MDPs, the Bellman optimality equation for has a unique solution. The Bellman optimality equation is actually a system of equations, one for each state, so if there are $|\mathcal{S}|$ states, then there are $|\mathcal{S}|$ equations in $|\mathcal{S}|$ unknowns. If the dynamics of the environment are known, then in principle one can solve this system of equations for v_* using any one of a variety of methods for solving systems of nonlinear equations.

However, this solution is rarely directly useful since it relies on at least three assumptions that are rarely true in practice: (1) we accurately know the dynamics of the environment; (2) we have enough computational resources to complete the computation of the solution; and (3) we know that the Markov property holds for the states of the problem. For the kinds of tasks in which we are interested, one is generally not able to implement this solution exactly because various combinations of these assumptions are violated. The core of many reinforcement learning methods is trying to approximately solve the Bellman optimality equation.

1.2 Partial observability

In many cases of interest, and certainly in the lives of all natural intelligences, the sensory input gives only partial information about the state of the world. The environment would emit not its states, but only *observations*—signals that depend on its state but, like a robot's sensors, provide only partial information about it. The environmental interaction (introduced in Eq. 1) would then have no explicit states or rewards in this case, but would simply be an alternating sequence of actions $a_t \in \mathcal{A}$ and observations $o_t \in \Omega$:

$$\tau = \{a_0, o_1, a_1, o_2, a_2, o_3, a_3, o_4 \dots\} \tag{10}$$

The framework of MDPs can be generalized to take partial observability into account. Partially Observable Markov Decision Processes (POMDPs) are MDPs in which the environment is assumed to have a well defined latent state s_t that underlies and produces the environment's observations, but is never available to the agent.

A POMDP can be formally described as a tuple

$$\text{POMDP} = \langle \mathcal{S}, \mathcal{A}, T, R, \Omega, O \rangle \tag{11}$$

where:

- $\mathcal{S}, \mathcal{A}, T, R$ is the tuple which defines the underlying MDP.
- Ω is a finite set of possible observations
- O is an observation function, with $O(a, s', o) = \Pr(o|a_t = a, s_{t+1} = s')$ being the probability of observing o given that the agent has executed action a, reaching state s'.

The natural Markov state for a POMDP is the distribution over the latent states, called the *belief.* For concreteness, assume the usual case in which there are a finite number of hidden states, defined in the set $\mathcal{S}$. Then, the belief at time t is the array $b_t \in \mathbb{R}^{|\mathcal{S}|}$ with components:

$$b_t[\,i\,] = \Pr(s_t = i|H_t) \quad \text{for all possible latent states } i \in \mathcal{S} \tag{12}$$

Notice that:

- $\sum_i b_t[\,i\,] = 1$, b_t being a probability distribution over latent states at time t
- the probability distribution is conditioned on H_t, which is the history of observations τ truncated at time t. This is important because the belief at the next time step $t+1$ is updated according to a specific function, given the previous belief b_t, an action a_t and an observation o_t:

  $$b_{t+1} = u(b_t, a_t, o_{t+1}) \tag{13}$$

  with the first belief b_0 given. Typically, in POMDPs every time the agent takes an action a and observes o, the *belief update* is done according to Bayes' rule

  $$b^{a,o}(s') = \Pr(s'|b, a, o) \tag{14}$$

  $$= \frac{O(a, s', o) \sum_{s \in \mathcal{S}} T(s, a, s') b(s)}{\sum_{s \in \mathcal{S}} b(s) \sum_{s' \in \mathcal{S}} T(s, a, s') O(a, s', o)} \tag{15}$$

  The update has to be computed for all states $s' \in \mathcal{S}$.

All beliefs are contained in a $|\mathcal{S}|$-dimensional simplex $\Delta(\mathcal{S})$. For instance, if $|\mathcal{S}| = 3$, the belief space is the standard 3-simplex (tetrahedron or triangular pyramid) shown in Figure 2.

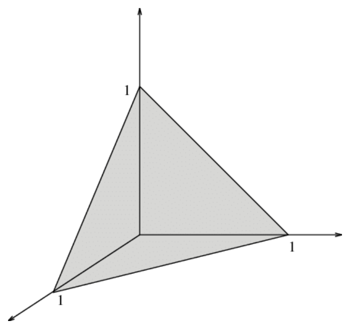


Figure 2: Belief space $\Delta(\mathcal{S})$ with $|\mathcal{S}| = 3$

Value function Analogously to the standard MDP setting, we can define a value function $v_\pi : \Delta(\mathcal{S}) \to \mathbb{R}$ which is defined as the expected discounted return the agent can gather by following π starting from belief b:

$$v_\pi(b) = \mathbb{E}_\pi[G_t | b_t = b] \tag{16}$$

$$= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| b_t = b\right] \tag{17}$$

In this case, the value function is defined over a continuous set of probability distributions, i.e., the belief space $\Delta(\mathcal{S})$, instead of the state space $\mathcal{S}$. It preserves a similar meaning: how good it is for the agent to have a given belief (in terms of discounted return).

¡

Optimal policy In POMDPs, a policy π maps beliefs to actions. Thus, analogously to the value function, also the policy is now a function over the belief space $\Delta(\mathcal{S})$ rather than the state space $\mathcal{S}$. As in the fully observable MDP setting, the goal of the agent is to learn an optimal policy. Optimal policies share the same optimal value function

$$v_*(b) = \max_\pi v_\pi(b) \quad \forall b \in \Delta(\mathcal{S}) \tag{18}$$

The Bellman optimality equation reads as:

$$v_*(b) = \max_{a \in \mathcal{A}} \left[\sum_{s \in \mathcal{S}} b(s) R(s,a) + \gamma \sum_{o \in \Omega} p(o|b,a) v_*(b^{a,o})\right] \tag{19}$$

where:

- $b^{a,o}$ is computed according to the belief update function (Eq. 15)

- $p(o|b,a)$ is the denominator of the belief update function (15) and corresponds to the complete knolwdge of the environment, namely **(check this formula)**

$$\begin{aligned} p(o|b,a) &= \mathbb{E}_s\, p(s', o|s,a) \\ &= \sum_{s\in\mathcal{S}} b(s) \sum_{s'\in\mathcal{S}} T(s,a,s')O(a,s',o) \end{aligned}$$

When Eq. (19) holds for every $b \in \Delta(\mathcal{S})$ we are ensured that the solution is optimal.

The optimal policy $\pi_* : \mathcal{B} \to \mathcal{A}$ mapping beliefs to actions can be directly computed from the Bellman optimality equation

$$\pi_*(b) = \arg\max_{a\in\mathcal{A}} \left[\sum_{s\in\mathcal{S}} b(s)R(s,a) + \gamma \sum_{o\in\Omega} p(o|b,a) v_*(b^{a,o}) \right] \tag{20}$$

Alpha vectors In POMDPs the optimal value function (Eq. (19)) exhibits particular structure - i.e., it is piecewise linear and convex - that one can exploit in order to facilitate computing the solution. In the following, some considerations on how to simplify the value iteration algorithm (presented in section 6.2) are reported being at the core of the algorithm used in our experiments.

If the agent has only one time step left to act, we only have to consider the immediate reward for the particular belief b, and can ignore any future outcome of the value function. The iterative update of the value iteration algorithm (Eq. (34)) reduces to:

$$v_k(b) = \max_{a\in\mathcal{A}} \sum_{s\in\mathcal{S}} b(s)R(s,a) \tag{21}$$

We can view the immediate reward function R as a set of n vectors, each one composed of $|\mathcal{S}|$ elements, representing the reward for taking action a in every possible state s

$$R = \left\{ \begin{bmatrix} \\ \alpha^1 \\ \\ \end{bmatrix}, \begin{bmatrix} \\ \alpha^2 \\ \\ \end{bmatrix}, \dots, \begin{bmatrix} \\ \alpha^n \\ \\ \end{bmatrix} \right\} \tag{22}$$

These are called *alpha vectors.* They are initialized with the reward function, then updated across the iterations of the POMDP solver algorithm.

Therefore, given a set of alpha vectors $\{\alpha_k^i\}_{i=1}^n$ at time step k, the value function v_k can be written as

$$v_k(b) = \max_{\{\alpha_k^i\}_i} b \cdot \alpha_k^i \tag{23}$$

Additionally, with each vector an action $a(\alpha_k^i) \in \mathcal{A}$ is associated, which is the optimal one to take in the current step.

Geometrically, we can say that the value function of POMDPs can be parametrized by a finite set of hyperplanes $\{\alpha_k^i\}_{i=1}^n$, thus resulting in a piece-wise linear, convex shape. Each vector defines a region in the belief space for which this vector is the maximizing element of v_k. These regions form a partition of the belief space, induced by the piecewise linearity of the value function. Examples of a value function for a two state POMDP ($|\mathcal{S}| = 2$) are shown in Figure 3. In this case the number of alpha vectors is $n = 4$.

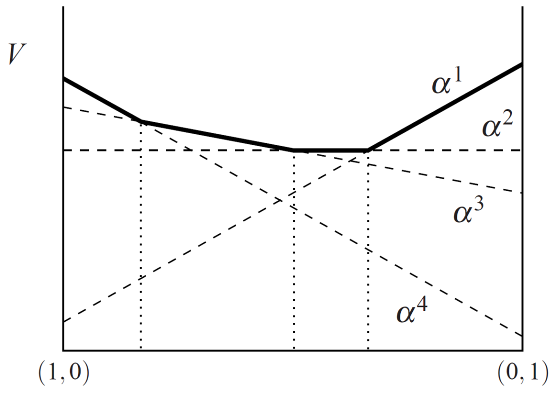


Figure 3: Detailed example of a POMDP value function when $|\mathcal{S}| = 2$. The y-axis shows the value of each belief, and the x-axis depicts the belief space $\Delta(\mathcal{S})$, ranging from (1,0) to (0,1).The value function is indicated by the solid black line, and in this case consists of four alpha vectors, indicated by dashed lines. The induced partitioning of the belief space into four regions is indicated by the vertical dotted lines. Credits: [2]

Notice that the convexity of the value function implies that the value of a belief close to one of the corners of the belief simplex $\Delta(\mathcal{S})$ will be high. A belief located exactly at a particular corner of $\Delta(\mathcal{S})$, i.e., $b(s) = 1$ for a particular s, defines with full certainty the state of the agent. In this way, the convex shape of v_π can be intuitively explained.

1.3 Point-Based Value Iteration (PBVI)

An important contribution to POMDP research in the past decade was the introduction of Point-Based Value Iteration (PBVI), a class of algorithms that allows us to approximately solve large POMDPs rapidly.

Instead of computing the value function on the entire belief space $\Delta(\mathcal{S})$ (which is generally intractable as discussed in section 6.2), a subset of the simplex is considered in order to update the alpha vectors and gradually build a good approximation of v_*. A pseudocode of a generic PBVI algorithm is shown in Figure 4.

The single iteration of a generic PBVI algorithm is composed of two stages:

- *Collect.* A subset of the belief space $B \subset \Delta(\mathcal{S})$ is considered. Drawing samples from the belief space can happen at random or according to some efficient heuristics (several have been proposed throughout the literature **(add ref?)**.

Algorithm 2 Generic point-based value iteration

```
1: while Stopping criterion not reached do
2:     Collect belief subset $B$
3:     Update $V$ over $B$
```

Figure 4: Pseudocode of a generic point-based value iteration algorithm. Credits: [3]

- *Backup.* The approximation of the value function is improved by updating the alpha vectors across the selected belief points. Each belief point $b \in B$ produces a new potential alpha vector

$$\text{backup}(b) = \underset{\{g_a^b\}_{a \in \mathcal{A}}}{\arg\max}\; b \cdot g_a^b \quad \text{where} \tag{24}$$

$$g_a^b = r_a + \gamma \sum_o \underset{\{g_{a,o}^i\}_i}{\arg\max}\; b \cdot g_{a,o}^i \tag{25}$$

$$g_{a,o}^i(s) = \sum_{s'} O(a, s', o) T(s, a, s') \alpha_k^i(s') \tag{26}$$

  Notice that Eq. (25) is directly derived from the Bellman optimality equation. After computing backup(b) for each belief $b \in B$, only the vectors making the value function increase are included in the new set of alpha vectors. Eventually, some pruning criteria can also be applied afterwards.

PBVI algorithms differ for the different criteria adopted to perform the two stages. Details about the PBVI algorithm used in our experiments are presented in section 2.2.

Although computing the vector backup(b) for a given b is straightforward, it is clearly intractable to compute it for every possible belief point in the simplex $\Delta(\mathcal{S})$.

This is why only a finite set of belief points is considered, being a natural way to sidestep this intractability. The backup stage reduces to applying Eq. (24) a fixed number of times, resulting in a small number of vectors (bounded by the size of the belief set B).

The general assumption underlying these so-called point-based methods is that by updating not only the value but also its gradient (the alpha vectors) at each $b \in B$, the resulting policy will generalize well and be effective for beliefs outside the set B. Whether or not this assumption is realistic depends on the POMDP's structure and the contents of B, but the intuition is that in many problems the set of 'reachable' beliefs (reachable by following an arbitrary policy starting from b_0) forms a low dimensional manifold in the belief simplex, and thus can be covered densely enough by a relatively small number of belief points. **("gradient update" needs to be clarified)**.

Clearly, of paramount importance is to locate a good set B of points in the belief space $\Delta(\mathcal{S})$, in the sense that it can provide a good approximation of the optimal value function. Clearly, the capacity of the algorithm to find a policy which generalizes well heavily depends on the belief points used to solve the Bellman equation. This is the key observation at the core of our experiments.

Intuitively, some heuristics for the *collect* phase can work properly on a set of problems, whereas other problems may require a different algorithm to efficiently sample the belief space.

2 Methods

In this section, we describe how the framework of POMDPs is used to formalize our specific use case: an olfactory navigation problem. Afterwards, we present the algorithm which has been used to solve it.

2.1 Olfactory navigation as a POMDP

Consider a food source located outdoors which exudes odor at a constant rate. An agent is given the opportunity to freely navigate the environment and at the same time sniff the air with the goal of detecting the odor to finally reach the food source. Figure 5 shows the 2D environment used throughout the experiments.

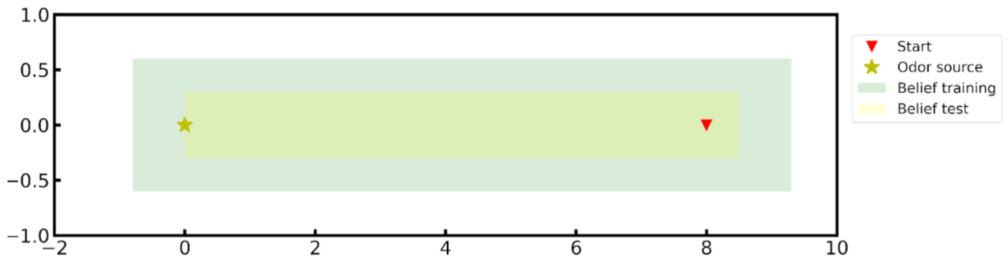


Figure 5: 2D environment used throughout the experiments. The red triangle indicates the starting position of the agent. The star denotes where the food source is located. The green and yellow rectangles in the background represent a uniform probability corresponding to the training and the test priors.

Olfactory navigation (or search) is thus the process of looking for the food source by exploiting odor cues as a guide along the path.

The agent performing the olfactory search is given two possible choices: (i) moving while sniffing on the ground or (ii) stopping and sniffing in the air.

Those two possible actions yield two different observations. The statistics of odors on the ground is profoundly different from the statistics of odors in the air (see figure 6 for snapshots of odor plume obtained from direct numerical simulations). As a consequence, detecting an odor in the air is much more likely than detecting an odor on the ground. Therefore, sniffing in the air might be more informative to guide the search, especially at larger distances from the source. A strategy is required to efficiently navigate the odors towards the source in the shortest amount of time.

Table 1 shows how the problem can be formalized as a POMDP. Notice that the agent's state is represented by its own position inside the environment. As a consequence, introducing partial observability in the MDP in this case means making the agent unaware of its actual position as it navigates across the environment.

Solving the POMDP yields a policy taken in response to a history of odor stimuli. Starting from the initial belief, the policy is used to compute the next action. The belief is updated according to the next observation, then the policy is again used to compute

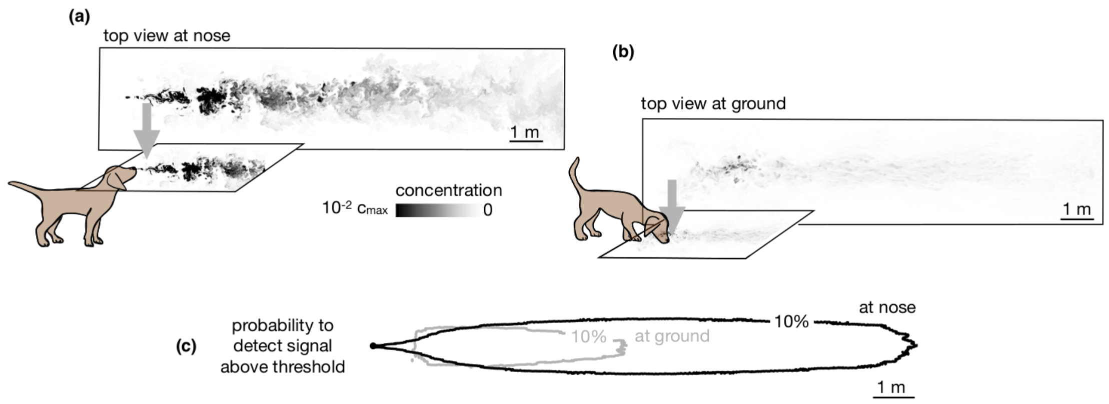


Figure 6: Different odor statistics. Top view of the odor plume (a) at nose height and (b) at ground. (c) 10% isoline of the probability to detect the odor (defined as the probability that odor is above a fixed threshold with respect to the maximum concentration at the source) at the ground (grey) and at the nose height (black). Credits: [5]

the next further action corresponding to the updated belief. The process continues until the source is reached or a stop criterion is met.

2.2 Forward Search Value Iteration (FSVI)

Forward Search Value Iteration (FSVI) [4] is one of the state-of-the-art algorithms for solving POMDPs. It has been used in our experiments.

FSVI constructs trajectories in the belief space by choosing actions according to a given policy (independently learned on the underlying MDP). All the beliefs encountered during such trajectories are collected and used to update the set of alpha vectors.

FSVI is an instance of PBVI. As such, two main phases can be distinguished:

- *Collect.* The initial state s_0 is sampled from the initial belief. Starting from s_0, a trajectory is computed based on a given policy, and the beliefs encountered along the trajectory are collected.
- *Backup.* A set of alpha vectors is computed on the set of collected beliefs. The value function is updated accordingly.

The trajectories considered with FSVI typically focus the belief search towards areas of high expected reward, and do so quickly by ignoring partial observability. An obvious limitation of this heuristic is that it does not select actions that can reveal information about the true state of the environment. For example, if collecting information requires a sequence of actions that moves the agent away from the goal/rewards, FSVI will not attempt these trajectories, and will not learn how the information can affect the policy. Still, FSVI will evaluate these actions during the point-based backup operation. Thus,

Table 1: Olfactory navigation as a POMDP

Main ingredients	
Environment	$W \times H$ grid world (circular)
Agent	single
States	position of the agent ($W \times H$ possible states)
Actions	sniff on the ground (still or move up, down, left, right), sniff in the air
Observations	no detection, detected odor, reached source
Belief	p.d.f. on the position of the agent ($W \times H$ probability map)
Reward	1 if the agent has reached the source, 0 otherwise
Transition probability	given a pair (state, action) emit 1 in correspondence of the next correct state
Observation probability	given a pair (state, action) emit the corresponding probability (0/1 for reached source, odor statistics for the rest)
Parameters	
Environment size	bounds of the rectangle $\{x_{\min}, x_{\max}, y_{\min}, y_{\max}\}$
Num. of states per length	number of cells λ per unit length of the environment
Odor source location	coordinates of the environment (x_g, y_g)
Odor statistics	$W \times H$ probability maps $o_{\text{ground}}, o_{\text{air}}$
Starting position	coordinates of the environment (x_s, y_s)
Detection radius	radius r_d of the target area around the source
Training prior	initial belief at training time b_{train}
Test prior	initial belief at test time b_{test}
Max. steps	maximum number $T_{\max}$ of steps allowed in a simulation

if information can be obtained using some action in the current belief state, this action will be evaluated and may become a part of the policy.

A pseudocode of FSVI is shown in Figure 7. Table 2 reports its parameters.

Algorithm 7 FSVI

```
Function FSVI
1: Initialize $V$
2: while $V$ has not converged do
3:     Sample $s_0$ from the $b_0$ distribution
4:     // Compute a trajectory assuming that $s_0$ is the initial state
5:     MDPExplore($b_0$, $s_0$)

Function MDPExplore($b$, $s$)
1: if $s$ is not a goal state then
2:     // Choose an action following the MDP policy for the current sampled state $s$
3:     $a^* \leftarrow \text{argmax}_a\, Q^{MDP}(s, a)$
4:     // Sample the next state and belief state
5:     Sample $s'$ from $T(s, a^*, *)$
6:     Sample $o$ from $O(a^*, s', *)$
7:     MDPExplore($b^{a^*,o}$, $s'$)
8: $V \leftarrow V \cup backup(b, V)$
```

Figure 7: Pseudocode of FSVI. Credits: [3]

Table 2: Parameters of FSVI

MDP policy	policy used to construct trajectories
Number of iterations	number n_{iter} of trajectories to be considered
Discount factor	γ used in the Bellman equation
Seed	used to pick the starting points of each trajectory

3 Experiments and results

In this section, we first introduce some preliminary results obtained as part of a recent work. Thereafter, our new experiments are presented with a description of the chosen parameters and a discussion of the results.

3.1 Preliminary results

In the work by Rigolli et. al. [5] the FSVI algorithm has been applied to the olfactory navigation problem as described in the section 2.1. An optimal navigation strategy has been formulated in accordance to the experimental results.

While the searcher could a priori reach the target using ground cues only, the experiments have demonstrated that alternation between sniffing on the ground and sniffing in the air emerges as a multi-modal navigation strategy.

More specifically, starting from the initial position the agent learns to move towards the odor source by alternating between two different behaviours:

- *Cast*: movement orthogonal to the wind direction
- *Surge*: movement parallel to the wind direction (upwind)

Such a strategy resembles the one utilized in nature by male moths trying to follow pheromone plumes of female moths. Figure 8 provides a visual example.

When the male moth detects the pheromone plume, it starts flying upwind, tracing the pheromone molecules in the plume (surge). However, as the structure of the plume is quite complex and unpredictable, the male moth looses track of the pheromone plume often during the surge behavior. For this reason, the male moths have developed a behavior that allows them to re-discover the pheromone plume again (cast). The casting frequency increases and the speed decreases when close to the source.

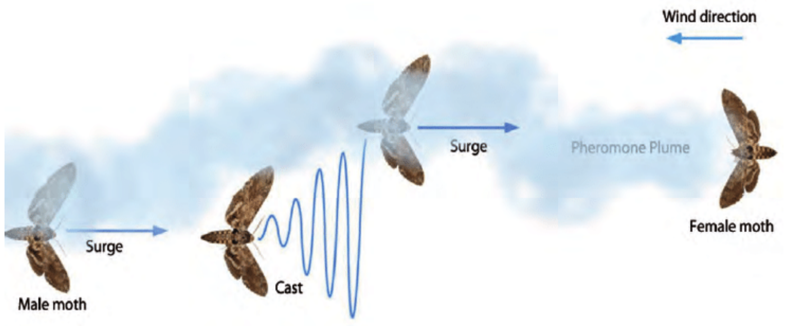


Figure 8: Illustration of the cast and surge behaviour performed by the male moth to follow the female pheromone plume. Credits: [6]

The policy learned by FSVI can be summarized as follows:

- *Initial condition.* See Figure 9 (top). The test prior is represented by a uniform probability distribution of size $L_x \times L_y$ (grey rectangle) surrounded by zero values (white background). It means that agent knows that its own positon is equally likely inside that area. The contour around the source (green diamond) shows the 25% isoline of probability of odor detection (size $x_{\text{thr}} \times y_{\text{thr}}$).

- *Search.* The search process is decomposed into $N \sim L_x/x_{\text{thr}}$ distinct episodes. In each episode the agent explores a region of size $L_y \times x_{\text{thr}}$ approximately (cast of width $\sim L_y$ + surge of length $\sim x_{\text{thr}}$). While casting the agent sniffs many times in the air (blue dots along the green trajectory). Since no odor is detected, the belief is updated accordingly, by "whitenining" a portion of the belief close to the source. On the other hand, as the agent surges upwind its belief about its own position translates forward with it.

- *Detection.* When odor is detected the non-zero values of the belief reduce to a narrow area. The agent is likely reaching the odor source in a few steps.

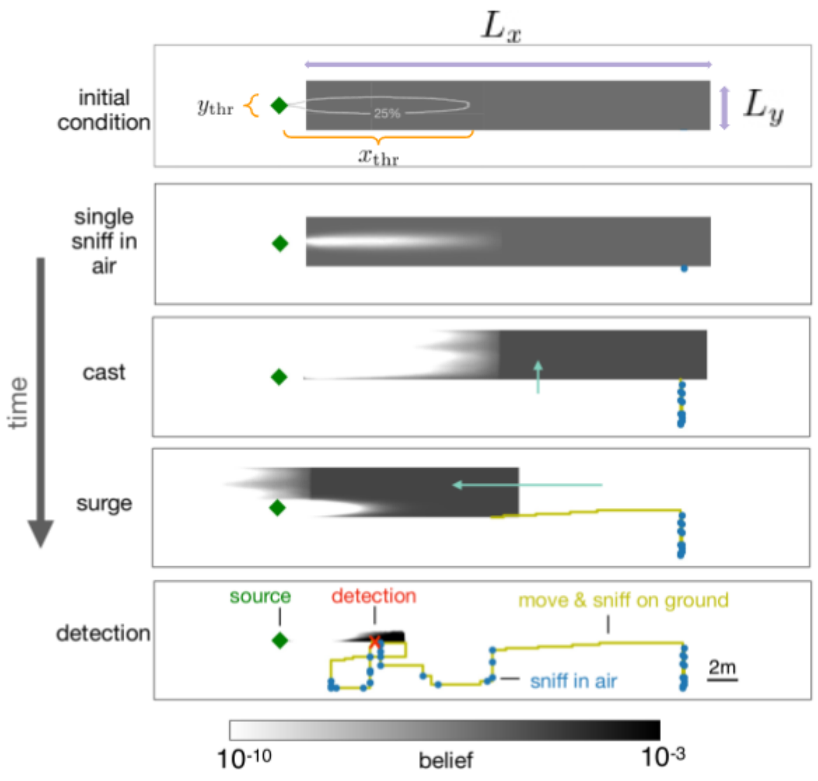


Figure 9: Key steps of the navigation strategy learned as a policy of the POMDP. Credits: [5]

The whole navigation strategy consists in trying to refine the agent's knowledge about its own position. Once the agent is confident enough about its own position, the task of reaching the source is straightforward because the location of the source is known to the agent. More generally, the less uncertainty the agent has over its true state, the better it can predict the future, and as such take better decisions.

Although such a problem formulation might appear counter-intuitive, it is equivalent to the opposite one in which the position of the agent is known but the position of the source is unknown. As a matter of fact, it is the relative position between the source and the agent which plays a key role in solving the problem. If this information is transparent, then the problem poses no challenges to derive a shortest-path policy like the one presented in Figure 12.

It is also worth pointing out that the agent is allowed to have complete knowledge about the statistics of the odor. Indeed, the belief update incorporates a precise mathematical description about it **(likelihood)**. When no odor is detected, the agent knows that it is likely to be located outside the range of high probability of detection close to the source. In this way, during the belief update the agent is capable of "whitening" a specific portion of the belief (refer to the second line of Figure 9).

Rigolli et. al. [5] have demonstrated that the final experimental strategy developed by the agent is supported by Marginal Value Theory (MVT).

3.2 Studying generalization

The objective of the experiments is to characterize generalization properties of the FSVI algorithm with respect to the points in the belief space which have been used to solve the Bellman equation. The analysis is twofold:

- Studying generalization as a function of the number of beliefs used to evaluate the Bellman equation (an increasing number of training iterations n_{iter} corresponds to an increasing number of beliefs used at training time to update the alpha vectors)
- Studying generalization as a function of the distance in the belief simplex between the distribution of beliefs at training time and the distribution of beliefs at test time. The Jensen-Shannon (JS) distance is used being a measure of the similarity between two probability distributions. It is based on the Kullback–Leibler divergence, with some notable differences, including that it is symmetric and it always has a finite value. The JS distance between two probability vectors p and q is defined as

$$\mathrm{JS}(p,q) = \sqrt{\frac{D(p \parallel m) + D(q \parallel m)}{2}} \tag{27}$$

where m is the pointwise mean of p and q and $D(\cdot \parallel \cdot)$ is the Kullback-Leibler divergence, namely

$$D(p \parallel q) = \sum_x p(x) \log\left(\frac{p(x)}{q(x)}\right) \tag{28}$$

The evaluation metrics used to quantify generalization are:

- *Additional time to target.* Additional number of steps A_T used to reach the source compared to the shortest path.

$$A_T = T - T_s \tag{29}$$

  where T_s is number of steps required for the shortest path (computed pretending that the environment is not circular), T is the actual number of steps used by the agent at the end of the simulation. Recall that by choice the cumulative reward can be either zero or one, indicating whether the agent has reached the source or not. Thus, the actual number T of steps can be indirectly computed as

$$T = \begin{cases} \left\lfloor \dfrac{\log(r_*)}{\log(\gamma)} \right\rfloor & \text{if } r_* > 0 \\ T_{\max} & \text{otherwise} \end{cases} \tag{30}$$

  where r_* is the cumulative reward, γ is the discount factor used in the Bellman equation, $T_{\max}$ is the maximum number of steps allowed in a simulation.

- *Cast width.* Maximum number of steps C_W defining a casting phase along the test trajectory. Given the trajectory array at the end of the simulation

$$\text{traj} = \Big[(x_1, y_1), (x_2, y_2), \ldots, (x_T, y_T)\Big] \tag{31}$$

  a cast c_i is defined as

$$\text{traj}[i : i + \alpha] \text{ is a cast } c_i \iff d_x(i) \leq \Delta * \epsilon \quad \text{AND} \quad d_y(i) \geq \Delta * \epsilon \tag{32}$$

  where $d_x(i) = \sum_{k=0}^{\alpha} x_{\text{diff}}(i + k)$, $d_y(i) = \sum_{k=0}^{\alpha} y_{\text{diff}}(i + k)$ measure the cumulative shifts over the x-axis and y-axis, respectively, over a time window of length α (x_{diff} and y_{diff} being the first-order absolute discrete differences of the trajectory). Notice that $\Delta = 1/\lambda$ is a datum of the problem (inverse of num. of states per length), whereas α, ϵ are two arbitrary parameters needed to fix the definition of a cast. In our experiments, $\alpha = 4$ and $\epsilon = 3$.

  Consecutive casts c_i are considered to be part of a single casting phase C_i (e.g., a casting phase C_i composed of three casts $\{c_i, c_{i+1}, c_{i+2}\}$ has length 3α). At the end, the cast width is given by the maximum length of all the casting phases

$$C_W = \max_i C_i \tag{33}$$

  An example of cast computation is shown in Figure 10.

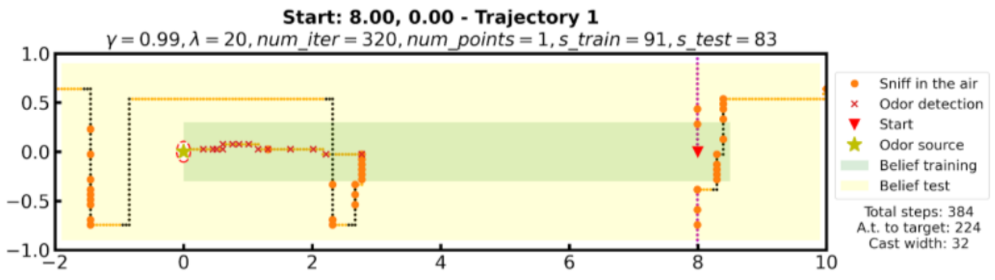


Figure 10: Examples of casting phases (black dots) computed along a given trajectory (yellow dots). Purple dots denote the casting phase corresponding to the final cast width.

The choice of such metrics is motivated as follows. The additional time to target quantifies the overall performance of the agent, without taking into account the shape of the trajectory. Depending on the parameters of the simulation (e.g., source location, agent's starting position) it is expected to be either close to or not too much larger than the shortest path in the case of a good performance. On the other hand, the cast width represents a valuable detail about the trajectory, in light of the optimal navigational strategy described in section 3.1. The theory prescribes that the cast width should be approximately equal to the test prior width. As a consequence, it represents a direct measure of how much the chosen trajectory looks like the optimal one.

Parameters setting Here are the POMDP parameters used throughout all the experiments:

- *Environment size.* The bounds of the environment are $\{x_{\min} = -2, x_{\max} = 10, y_{\min} = -1, y_{\max} = 1\}$.
- *Num. of states per length.* The number of cells per unit length is $\lambda = 20$. Thus, the grid world has width $W = 12 * \lambda = 240$, height $H = 2 * \lambda = 40$ with a total number of cells equals to 9600.
- *Odor source location.* The odor source is located at the origin $(x_g, y_g) = (0, 0)$.
- *Odor statistics.* Two probability maps (one for odor in the air and one for odor at the ground) are obtained from direct numerical simulations of odor transport. Figure 11 shows the probability $o_{\text{air}}, o_{\text{ground}}$ for each cell of the grid world.
- *Starting position.* The agent starts at the position $(x_s, y_s) = (8, 0)$
- *Detection radius.* The target area around the source is defined by the radius $r_d = 2.1 * \Delta x$, with $\Delta x = (x_{\max} - x_{\min}) / \Big((x_{\max} - x_{\min}) * \lambda - 1 \Big)$

- *Training prior.* The initial belief at training time is zero everywhere except a uniform probability distribution inside a rectangle with variable bounds $\left\{x^{\text{tr}}_{\text{min}}, x^{\text{tr}}_{\text{max}}, y^{\text{tr}}_{\text{min}}, y^{\text{tr}}_{\text{max}}\right\}$
- *Test prior.* The initial belief at test time is zero everywhere except a uniform probability distribution inside a rectangle with variable bounds $\left\{x^{\text{te}}_{\text{min}}, x^{\text{te}}_{\text{max}}, y^{\text{te}}_{\text{min}}, y^{\text{te}}_{\text{max}}\right\}$
- *Max. steps.* The maximum number of steps allowed in a simulation is $T_{\text{max}} = 500$.

Here are the FSVI parameters used throughout all the experiments:

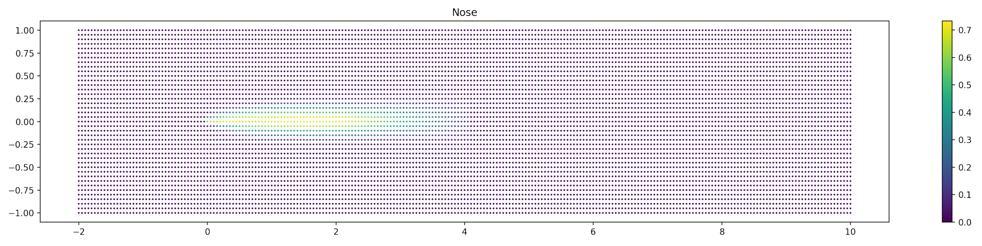
- *MDP policy.* The policy used to construct trajectories is computed once through 1000 iterations of the standard Value Iteration algorithm (described in section 6.1). Then, for each experiment it is loaded from memory. Figure 12 shows the learned policy and the corresponding value function.
- *Number of iterations.* The number of trajectories n_{iter} is variable. Figure 13 shows examples of starting points drawn uniformly at random from the training prior. Given a number of iterations, all the trajectories considered at training time start from the blue dots and follow the MDP policy shown in Figure 12.
- *Discount factor.* $\gamma = 0.99$.
- *Seed.* Every experiment has a different seed. By changing seed the starting points of the trajectories would be different from those in Figure 13.

Additional time to target Two main batches of experiments have been run. In the first batch, the test prior has been fixed with bounds $\left\{x^{\text{te}}_{\text{min}} = 0.0, x^{\text{te}}_{\text{max}} = 8.5, y^{\text{te}}_{\text{min}} = -0.3, y^{\text{te}}_{\text{max}} = 0.3\right\}$, whereas the training prior has been varied (with increasing area compared to the test prior). In the second batch, the opposite has happened.

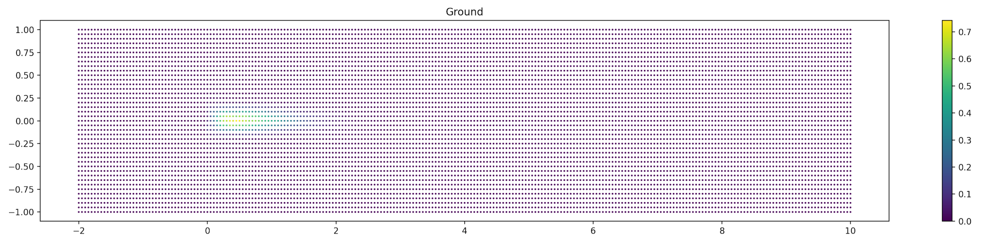
Here we show the results of the first batch, the other ones being almost identical. Figure 14 and Figure 15 report the additional time to target as a function of the number of iterations and of the average JS distance, respectively. Eight different simulations have been considered for each configuration of the parameters. The average value is reported with the error bars representing a single standard deviation.

The simulations have been run in parallel on a farm of CPUs with average training times reported in Table 3.

Two clear trends can be observed in the figures. In Figure 14 the additional time to target decreases as the number of iteration increases. This means that the more beliefs upon which the alpha vectors are updated the faster is the agent to reach the source. Thus, we can infer that the approximation of the value function improves with the number of iterations. In some plots, the trend is not strictly decreasing yet we can observe that a minimum value is approached at some point, with the algorithm reaching a convergence.



(a)



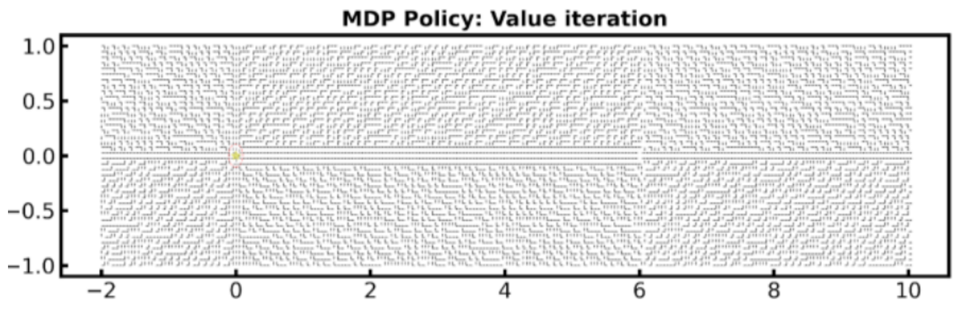
(b)

Figure 11: The probability maps of detecting an odor signal used to train the POMDP: (a) in the air o_{air} and (b) at the ground o_{ground}.

Furthermore, we can observe that the maximum and the minimum values of the last plot (bottom right) are larger than the maximum and the minimum value of the first plot (top left). In the last plot the training prior is much larger compared to the test prior, whereas in the first plot the training prior is equal to the test prior. This means that when the priors are quite different, a larger number of iterations is required to reach a similar additional time to target (i.e., a better approximation of the value function is required to achieve a similar performance).

In Figure 15 the additional time to target increases along with the JS distance. In this case, a JS score close to zero means that some beliefs at training time are almost identical to some other beliefs at test time, whereas a JS score close to 1 means that the closest beliefs between training and test time are completely distant from each other. As a consequence, we can say that the closer the beliefs at training time to the beliefs at test time the faster is the agent to reach the source. Hence, we can infer that the approximation of the value function improves as the distribution of beliefs at training time is closer to the distribution of beliefs at test time (i.e., intuitively when beliefs encountered at test time occupy a portion of the belief space which is quite overlapped with the one explored at training time).

Additionally, we can observe that the range of the JS score changes across the plots. In the first plot (top left) the JS score ranges approximately between 0.30 and 0.41, whereas in the last plot (bottom right) the JS score ranges between 0.57 and 0.70. Therefore, when the training prior is equal to the test prior the distribution of beliefs at training time is more similar to the distribution of beliefs at test time compared to the case when



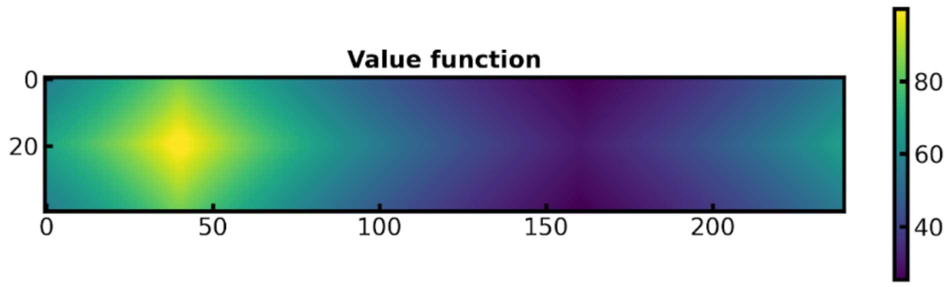


Figure 12: (Top) MDP Policy learned through Value Iteration. Each cell of the grid world is associated with an arrow indicating the direction of the agent's movement. Three main "central corridors" can be identified around $y = 0$. From $x = -2$ to $x = 0$ and from $x = 6$ to $x = 10$ the direction of the movement is rightward (due to the circular environment); from $x = 0$ to $x = 6$ the movement is leftward. All the other cells point towards the central corridor. The red circle around the source denotes the target area defined by the detection radius. (Bottom) The corresponding value function.

the training prior is much larger than the test prior.

Overall, we can say that given a training prior b_{train} and a test prior b_{test} a good generalization in terms of additional time to target can be achieved by requiring a specific number of iterations and a specific JS score.

Cast width The cast width has been measured on the trajectories obtained by the same batches of experiments described in the previous paragraph.

Figure 16(a) shows that the cast width increases with the test prior width. Thus, regardless of the fact that the training prior is fixed and smaller compared to the test prior, the casts performed by the agent are approximately equal to the width of the test prior L_y. This is in accordance to the optimal navigational strategy introduced in the section 3.1, in which each episode contains a casting phase of width $\sim L_y$.

On the other hand, Figure 16(b) shows that the cast width stays almost constant when the training prior width is increased. This is again in accordance to the optimal navigation strategy. Despite the fact that the area of the training prior grows, the casts

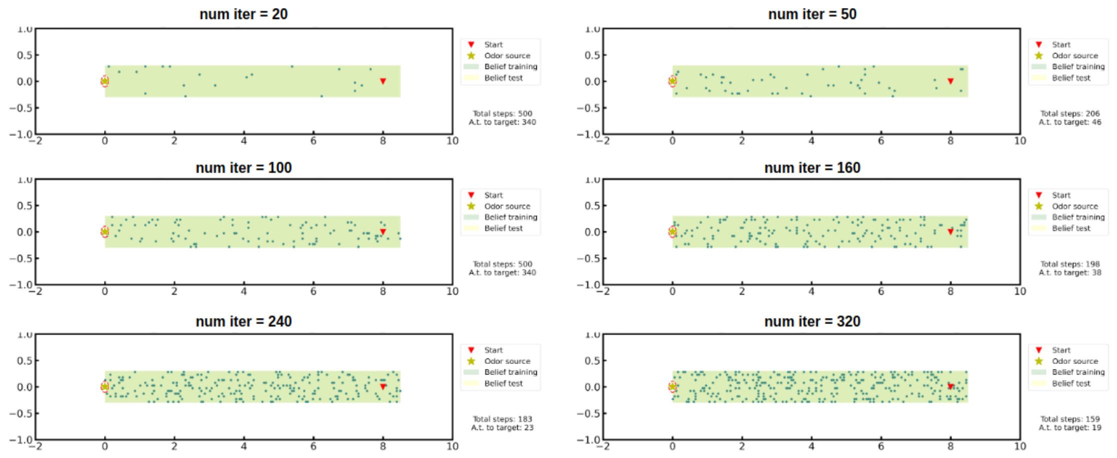


Figure 13: Examples of starting points with an increasing number of iterations.

performed by the agent remain approximately equal to the width of the test prior.

In general terms, we can say that the learned policy generalizes well to different test priors when a specific number of iterations is required at training time. The blue and the yellow curves in the right plot (corresponding to a number of iterations of $20, 50$ respectively) show two cases in which the casting width is not constant, therefore we can conclude that the algorithm has not converged, and the learned policy does not generalize well compared to the other cases.

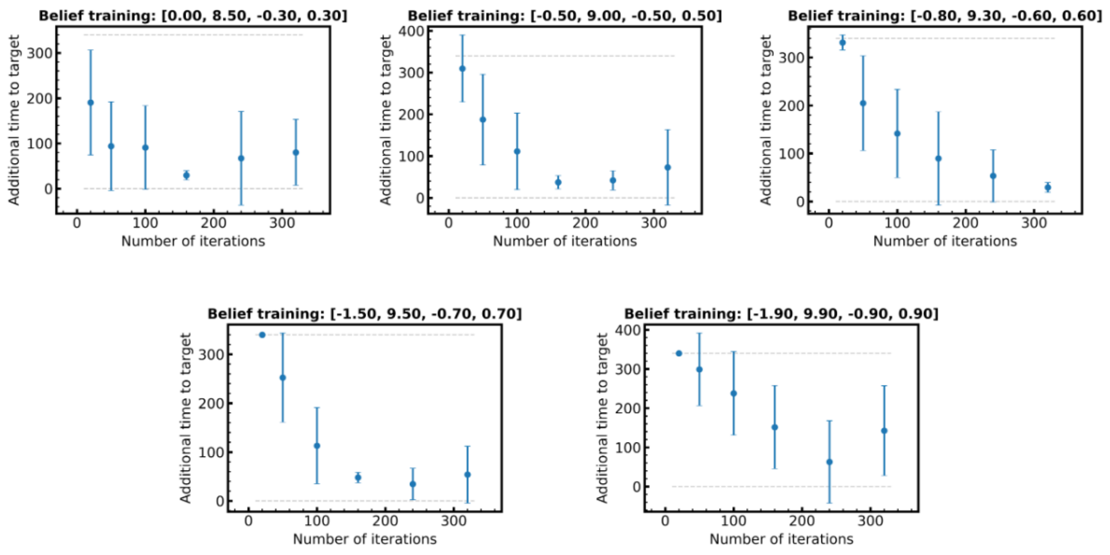


Figure 14: Additional time to target as a function of the number of iterations. The title of the plots contains the bounds of the training priors. The grey dotted lines represent two extreme cases. Crossing the lower one means that in some experiments the agent has exploited the circularity of the environment to reach the source in a time shorter than T_s. Instead, the upper one simply denotes the maximum number of additional steps $T_{\max} - T_s$, being an upper bound on the result of a single experiment.

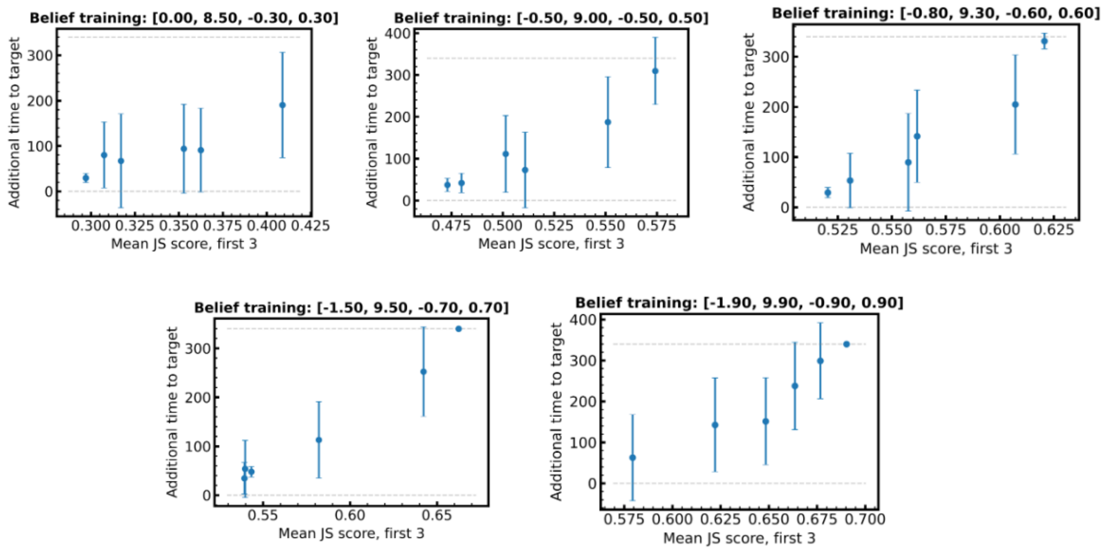
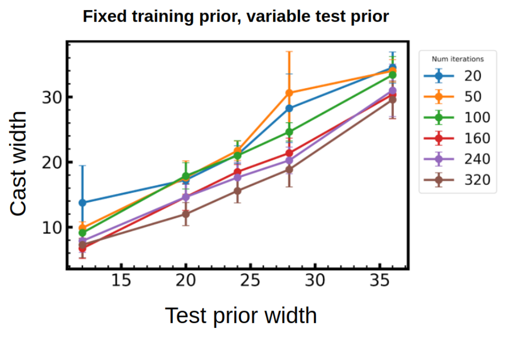
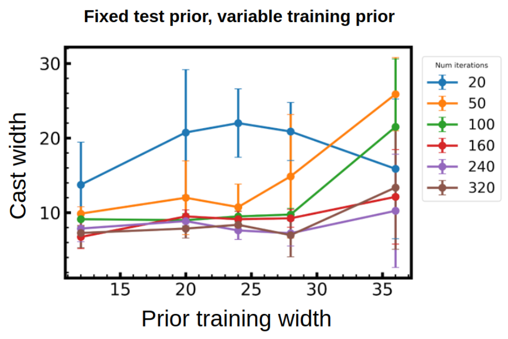


Figure 15: Additional time to target as a function of the average JS distance (computed on the first three neighbours). The title of the plots contains the bounds of the training priors.



(a)



(b)

Figure 16: Cast width as a function of (a) the test prior width, (b) the training prior width.

4 Code

4.1 Serial implementation

The code used to run experiments has been implemented from scratch. The language is C++. It makes use of the GSL library.

Table 3: Average training time

	Number of iterations					
	20	50	100	160	240	320
Single experiment	4m	26m	1.6h	4.6h	11.3h	20.4h

Instructions on how to use the code to reproduce the experiments on the UNice cluster (maybe on a dedicated, private GitHub page).

4.2 Moving to GPU

Preliminary FLOPS analysis Have a look at the Table 4 (legend in Table 5).

Table 4: Number of operations

Full backup	Approx. flops
$\text{backup}(b) = \arg\max_{\{g_a^b\}_{a\in\mathcal{A}}} b \cdot g_a^b$	$\lvert B\rvert\lvert\mathcal{S}\rvert\lvert\mathcal{A}\rvert$
$g_a^b = r_a + \gamma \sum_o \arg\max_{\{g_{a,o}^i\}_i} b \cdot g_{a,o}^i$	$\lvert\mathcal{A}\rvert\lvert B\rvert\ n\lvert\mathcal{S}\rvert\lvert\Omega\rvert$
$g_{a,o}^i(s) = \sum_{s'} O(a,s',o)\, T(s,a,s')\, \alpha_k^i(s')$	$3n\ \lvert\mathcal{A}\rvert\lvert\Omega\rvert\lvert\mathcal{S}\rvert$
$r_a(s) = \sum_{s'} T(s,a,s')\, R(s,a,s')$	–
	$\lvert\mathcal{A}\rvert\lvert\mathcal{S}\rvert \Big(\lvert B\rvert + \lvert B\rvert\lvert\Omega\rvert n + 3n\lvert\Omega\rvert\Big)$

Policy evaluation	Approx. flops
$\pi_*(b) = \arg\max_{a\in\mathcal{A}} \left[\sum_{s\in\mathcal{S}} b(s)R(s,a) + \gamma \sum_{o\in\Omega} p(o\mid b,a)v_*(b^{a,o})\right]$	$\lvert\mathcal{A}\rvert\ (4+n)\lvert\mathcal{S}\rvert$

Table 5: Legend of Table 4

Symbol	Description
	MDP
$\mathcal{S}$	Set of states
$\mathcal{A}$	Set of actions
	POMDP
B	Set of beliefs considered at training time
Ω	Set of observations
n	Number of alpha vectors

Short intro to CUDA CUDA (Compute Unified Device Architecture) is a parallel computing platform and an application programming interface (API) model created by Nvidia. It allows software developers to use a graphics processing unit (GPU) for general purpose processing – an approach termed GP-GPU (general-purpose computing on graphics processing units).

CUDA C is an extension of the C language incorporating special qualifiers, types and variables that gives direct access to the GPU's virtual instruction set and its computational elements.

Parallelization on the GPU can be implemented by mapping computational tasks to a virtual, multi-dimensional grid like the one shown in Figure 17.

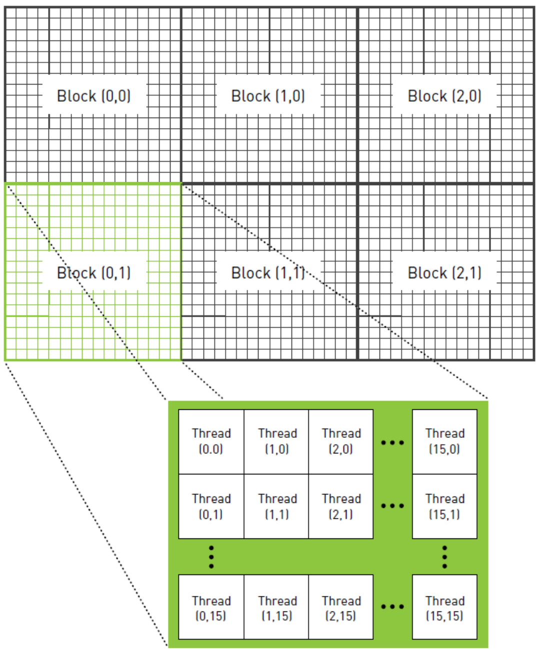


Figure 17: Example of a parallel computational grid on CUDA. Credits: [7]

The grid is made of two elements: blocks and threads. Depending on the desired level of parallelization, developers have to specify the size and the composition of the grid and write some code to efficiently split tasks across blocks and threads.

Blocks can be arranged in up to three dimensions, whereas threads can be arranged in up to two dimensions. The number of blocks is by default limited to 65,535, the maximum number of threads per block is a device property, typically 512.

A shared memory can be used to create a copy of the variable for each block that you launch on the GPU. Every thread in that block shares the memory, but threads cannot see or modify the copy of this variable that is seen within other blocks. This provides an excellent means by which threads within a block can communicate and collaborate on computations. Furthermore, shared memory buffers reside physically on the GPU as opposed to residing in off-chip DRAM. Because of this, the latency to access shared memory tends to be far lower than typical buffers.

Parallelizing the operations The two main operations being parallelized are *Full Backup* and *Policy evaluation*, with the corresponding operations reported in Table 4.

A pseudocode of the CUDA version of such operations is reported in Algorithm 1 and Algorithm 2. A descriptions of the CUDA kernels is reported in the supplementary material "Understanding kernel computations".

Algorithm 1: Full backup on CUDA

Input

- B : belief set (size $|B| \times |S|$)
- γ : discount factor
- R : reward matrix (size $|A| \times |S|$)
- T : transition matrix (size $|A| \times |S|$)
- O : observation matrix (size $|\Omega| \times |A| \times |S|$)
- Γ : set of alpha vector (size $n \times |S|$)

Output

- B_{KP} : the set of new alpha vectors for each belief in the belief set B (size $|B| \times |S|$)

Pseudo-code

Each step corresponds to a CUDA kernel

1. Compute the matrix G_{AO} (size $|A||\Omega||S|n$) element-wise

$$g^i_{a,o}(s) = \sum_{s'} O(a,s',o)T(s,a,s')\alpha_i(s')$$

2. Compute the matrix $\hat{G}_{AO}$ (size $|A|n|\Omega||B|$) element-wise

$$\hat{G}_{AO}\underbrace{[\,j\,]}_{\text{4D}}\underbrace{[\,o\,]}_{\text{3D}}\underbrace{[\,a\,][\,i\,]}_{\text{2D}} = b_j \cdot g^i_{a,o}$$

3. Compute the matrix $\hat{G}^*_{AO}$ (size $|A||\Omega||B|$) element-wise

$$\hat{G}^*_{AO}\underbrace{[\,j\,]}_{\text{3D}}\underbrace{[\,a\,][\,o\,]}_{\text{2D}} = i^*_{a,o,j} = \arg\max_i\ b_j \cdot g^i_{a,o}$$

4. Compute the matrix G_{AB} (size $|A||S||B|$) element-wise

$$G^B_A\underbrace{[\,j\,]}_{\text{3D}}\underbrace{[\,a\,][\,:\,]}_{\text{2D}} = r_a + \gamma \sum_o \operatorname*{arg\,max}_{\{g^i_{a,o}\}_i} b \cdot g^i_{a,o}$$

5. Compute the matrix $\hat{G}_{AB}$ (size $|B||A|$) element-wise

$$\hat{G}^B_A[\,j\,][\,a\,] = b_j \cdot g^{b_j}_a$$

6. Compute the vector $\hat{G}^*_{AB}$ (size $|B|$) on CPU element-wise:

$$\hat{G}^*_{AB}[\,j\,] = a^*_{b_j} = \arg\max_a b_j \cdot g^{b_j}_a$$

7. Compute the matrix B_{KP} (size $|B||S|$) element-wise

$$B_{KP}[\,j\,][\,s\,] = \operatorname*{arg\,max}_{\{g^b_a\}_{a\in\mathcal{A}}}\ b_j \cdot g^b_a = G^B_A[\,j\,][\,a^*_{b_j}\,][\,s\,]$$

Algorithm 2: Policy evaluation on CUDA

Input
- b : current belief (size $|S|$)
- γ : discount factor
- R : reward matrix (size $|A| \times |S|$)
- T : transition matrix (size $|A| \times |S|$)
- T_{inv} : inverse transition matrix (size $|A| \times |S|$)
- O : observation matrix (size $|\Omega| \times |A| \times |S|$)
- Γ : set of alpha vector (size $n \times |S|$)

Output
- a^* : best action given the current belief

Pseudo-code

Split sum:

$$\pi_*(b) = \arg\max_{a \in \mathcal{A}} \underbrace{\sum_{s \in \mathcal{S}} b(s)R(s,a)}_{c_1} + \gamma \underbrace{\sum_{o \in \Omega} p(o|b,a)v_*(b^{a,o})}_{c_2}$$

```
foreach action $a \in \mathcal{A}$ do
    1. Compute $\sum_{s \in \mathcal{S}} b(s)R(s,a)$ as a dot product on GPU $\longrightarrow c_1 = b \cdot R[\,a\,][\,:\,]$
    2. Initialize $c_2 = 0$
    3. foreach observation $o \in \Omega$ do
        4. Compute $b_{a,o}$ on GPU $(b, O, T_{\text{inv}}, a, o)$
        5. Compute $v_*(b_{a,o}) = \max_{\{\alpha_i\}_i} b_{a,o} \cdot \alpha_i \longrightarrow$ each dot product $b_{a,o} \cdot \alpha_i$ on GPU
        6. Compute $p(o \,|\, b, a) \longrightarrow$ dot product $b \cdot t_{a,o}$ on GPU
        7. $c_2 += p(o \,|\, b, a)\, v_*(b_{a,o})$
    end
    $\text{sum}_a = c_1 + \gamma\, c_2$
end
return  $a^* = \arg\max_a \text{sum}_a$
```

5 Conclusion and future work

POMDP is a valuable problem formulation for the use case at hand. In the formulation of the problem as a POMDP there are two key assumptions:

- the location of the source is known
- the agent incorporates a model of the odor statistics in the belief updates

Thanks to them, the solution to the POMDP provides an *explainable policy* supported by Marginal Value Theory (see [5]). In the case of deep-Q-learning (see Appendix), all the information has been incapsulated into the reward function, providing a solution which is not as valuable as the one obtained with the FSVI algorithm.

Future work

- Improve JS score. Find a universal measure which has a direct correlation with generalization properties.
- *Does FSVI provide a good heuristics for collecting the belief points for the problem at hand?* Need to have comparisons of the same evaluation metrics measured with other algorithms (e.g., add results with Perseus, PBVI, ..)
- Benchmark with other Q-learning algorithms.

6 Appendix

6.1 Value Iteration for MDPs

The pseudocode is reported in Figure 18.

```
Value Iteration, for estimating $\pi \approx \pi_*$

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

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
```

Figure 18: Pseudocode of the value iteration algorithm. Credits: [1]

6.2 Exact Value Iteration for POMDPs

Analogously to MDPs, value iteration is a method for solving POMDPs that builds a sequence of value-function estimates which converge to the optimal value function for the current task.

Value iteration consists of converting the Bellman optimality equation for POMDPs (19) into un update rule.

$$v_{k+1}(b) = \max_{a \in \mathcal{A}} \left[\sum_{s \in \mathcal{S}} b(s) R(s,a) + \gamma \sum_{o \in \Omega} p(o|b,a) v_k(b^{a,o}) \right] \tag{34}$$

In the process knwon as *exact value iteration*, in each iteration the value function is updated across the entire belief space. The Bellman backup can be written in terms of vector operations, and operations on sets of vectors

$$J_P v_k = \bigcup_{a \in \mathcal{A}} G_a \tag{35}$$

$$G_a = \bigoplus_{o \in \Omega} \left\{ \frac{1}{|\Omega|} r_a + \gamma g^i_{a,o} \right\}_i \tag{36}$$

$$g^i_{a,o} = \sum_{s'} O(a, s', o) T(s, a, s') \alpha^i_k(s') \tag{37}$$

where $r_a(s) = R(s,a)$ and $\bigoplus$ denotes the cross-sum operator[1].

Notice that we operate independent of a particular belief b now so Eq. (25) can no longer be applied. Instead we have to include all ways of selecting $g^i_{a,o}$ for all o.

In Eq. (36) we create $|v_k|^{|\Omega|}$ new vectors for each action, with a complexity of $|S|$ for each new vector. In Eq. (37) we generate $|v_k||\mathcal{A}||\Omega|$ vectors and computing each of these vectors takes $|\mathcal{S}|^2$ operations. Hence, the overall complexity of a single iteration is:

$$\underbrace{|v_k||\mathcal{A}||\Omega|\,|\mathcal{S}^2|}_{g^i_{a,o}} + \underbrace{|\mathcal{A}||v_k|^{|\Omega|}\,|\mathcal{S}|}_{G_a} \tag{38}$$

In practice, exact value iteration is only feasible for the smallest of problems, since the set of α-vectors grows exponentially in $|\Omega|$ with every iteration. As the computational cost of each iteration depends on the number of vectors in v_k, an exponential growth makes the algorithm prohibitively expensive. To some degree, the sets of α-vectors can be reduced to their minimal form after each stage, resulting in more manageable value functions. But this is not sufficient for scaling to domains with more than a few dozen states, actions, and observations. The regions of many of the generated vectors will be empty and these vectors are useless as such, but identifying and subsequently pruning them requires linear programming which introduces considerable additional cost (e.g., when the state space is large).

6.3 Q-learning

Recall that the value function $v_\pi(s)$ is the expected return when starting in s and following π thereafter (Eq. 5). Similarly, we can define the value of taking action a in state s under a policy π, denoted $q_\pi(s,a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| s_t = s, a_t = a\right] \tag{39}$$

We call q_π the *action-value function* for policy π or alternatively the *q-function*.

The optimal value functions v_* and q_* assign to each state, or state-action pair, the largest expected return achievable by any policy.

The Bellman optimality equation for q_* is

$$q_*(s,a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma \max_{a'} q_*(s_{t+1}, a') \middle| s_t = s, a_t = a\right] \tag{40}$$

$$= \sum_{s'}\sum_{r} p(s', r|s,a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \tag{41}$$

Once one has v_* or q_* it is easy to determine a corresponding optimal policy $\pi_* : \mathcal{S} \to \mathcal{A}$ mapping states to actions

[1]Cross-sum of sets is defined as: $\bigoplus_k R_k = R_1 \oplus R_2 \oplus \cdots \oplus R_k$ with $P \oplus Q = \{p + q | p \in P, q \in Q\}$

$$\pi_*(s) = \arg\max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_*(s')] \tag{42}$$

$$\pi_*(s) = \arg\max_a q_*(s, a) \tag{43}$$

Clearly, deriving π_* from q_* is immediate. On the other hand, a one-step search is required to derive π_* from v_*: the actions that appear best after a one-step search will be optimal actions. Another way of saying this is that any policy that is greedy with respect to the optimal value function v_* is an optimal policy.

The relationship between q_*, v_*, π_* can be easily understood by looking at their matrix representations:

$$\begin{bmatrix} \\ \\ \qquad\qquad q_* \qquad\qquad \\ \\ \\ \end{bmatrix}_{|\mathcal{S}|\times|\mathcal{A}|} \quad \begin{bmatrix} \qquad v_* \qquad \end{bmatrix}_{|\mathcal{S}|} \quad \begin{bmatrix} \qquad \pi_* \qquad \end{bmatrix}_{|\mathcal{S}|} \tag{44}$$

where

$$v_*(s) = \max_a q_*(s, a) \tag{45}$$

$$\pi_*(s) = \arg\max_a q_*(s, a) \tag{46}$$

i.e., v_* is the maximum of the matrix q_* row-wise and π_* contains the corresponding action indexes.

The algorithm known as Q-learning tries to directly approximate the optimal value function q_* through an iterative update

$$q(s, a) \leftarrow q(s, a) + \alpha\Big[r(s, a) + \gamma \max_{a'} q(s', a') - q(s, a)\Big] \tag{47}$$

6.3.1 Deep Q-learning with experience replay

Deep Q-learning is a modification of the standard Q-learning algorithm that can learn successful policies directly from high-dimensional sensory inputs (e.g., images) using end-to-end reinforcement learning. In particular, it has been used to learn from raw pixels how to play many Atari games at human-level performance.

The q-function is approximated using a deep neural network termed Deep Q-Network (DQN). The q-function is represented by $q(s, a; \theta_i)$, in which θ_i are the parameters (that is, weights) of the Q-network at iteration i.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Two modifications are applied in order to make the Q-learning algorithm suitable for training large neural networks without diverging.

- Experience replay. Agent's experience $e_t = (s_t, a_t, r_t, s_{t+1})$ is stored at each time step t in a data set $D_t = \{e_1, \ldots, e_t\}$. During learning, Q-learning updates are applied on minibatches of experience drawn uniformly at random from the pool of stored samples $(s, a, r, s') \sim U(D)$

- Separate target network. A separate DQN is used to generate the targets y_j in the Q-learning update (different from the deep Q-network used for training). More precisely, every C updates the parameters of the training network θ_i at iteration i are cloned to obtain the parameters θ_i^- of the target network.

The DQN can be used to learn the optimal q-function q_* by minimizing the mean squared error in the Bellman equation (41).

$$\min_{\theta_i} \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(\underbrace{r + \gamma \max_{a'} q(s', a'; \theta_i^-)}_{\text{target } y \text{ from (41)}} - \underbrace{q(s, a; \theta_i)}_{\text{prediction } \hat{y}}\right)^2\right] \tag{48}$$

Once the q-function has been learned, the evaluation procedure is typically conducted with an ε-greedy policy with a small ε (e.g., 0.05), in order to minimize the possibility of overfitting during evaluation.

It is worth pointing out the advantages of the DQN algorithm compared to the classical Q-learning algorithm:

- The use of experience replay has three main advantages. First, each step of experience is potentially used in many weight updates, which allows for greater data efficiency. Second, learning directly from consecutive samples is inefficient, owing to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates. Third, when learning onpolicy the current parameters determine the next data sample that the parameters are trained on. Unwanted feedback loops may arise and the parameters could get stuckin a poor localminimum, or even diverge catastrophically. By using experience replay the behaviour distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters.

- The use of a separate target network makes the algorithm more stable compared to standard online Q-learning, where an update that increases $q(s_t, a_t)$ often also increases $q(s_{t+1}, a)$ for all a and hence also increases the target y_j, possibly leading to oscillations or divergence of the policy. Generating the targets using an older set of parameters reduces the correlation between the predicted q values and the target values, and adds a delay between the time an update to q is made and the time the update affects the targets y_j, making divergence or oscillations much more unlikely.

6.3.2 Rainbow

Rainbow combines six extensions to the DQN algorithm (namely: double Q-learning, prioritized replay, dueling networks, multi-step learning, distributional RL and noisy nets). The Deep Q-Network is used to learn the distribution of returns (instead of the expected return)

$$p_\theta^i(s, a) = Pr(G_t = z_i | s_t = s, a_t = a) \tag{49}$$

i.e., the probability that the return at time t (following policy π) equals z_i, where z_i is an atom of a discrete support z

$$\Big[\qquad z \qquad \Big]_{N_{\text{atoms}}}$$

$$z_i = v_{\min} + (i-1)\frac{v_{\max} - v_{\min}}{N_{\text{atoms}} - 1} \tag{50}$$

The network architecture is a dueling network with two streams of computation, the value and advantage streams, sharing an initial encoder. The encoder can be a fully-connected or a convolutional neural network composed of a few layers, whereas the value and the advantage can be for example fully-connected networks with a single hidden layer.

The original computation of the q-function is as follows:

$$q_\theta(s, a) = \underbrace{v_\eta(\phi)}_{\text{value}} + \underbrace{a_\psi(\phi, a)}_{\text{advantage}} - \underbrace{\bar{a}_\psi(\phi)}_{\text{aggregator}} \tag{51}$$

with $\phi = f_\xi(s)$ being the output of the shared encoder, $\bar{a}_\psi(\phi) = \frac{1}{N_{\text{actions}}} \sum_{a'} a_\psi(\phi, a')$.

The architecture is adapted for use with return distributions, hence the output of the value and the aggregator streams are respectively:

$$\Big[\quad v_\eta \quad\Big]_{N_{\text{atoms}}} \qquad \left[\quad a_\psi \quad\right]_{N_{\text{actions}}\times N_{\text{atoms}}}$$

A softmax layer is used to obtain the normalised returns' distribution:

$$p^i_\theta(s,a) = \frac{\exp(v^i_\eta(\phi) + a^i_\psi(\phi,a) - \bar{a}^i_\psi(\phi))}{\sum_{j=1}^{N_{\text{atoms}}} \exp(v^j_\eta(\phi) + a^j_\psi(\phi,a) - \bar{a}^j_\psi(\phi))} \tag{52}$$

Notice that the parameter θ is the concatenation of the parameters of the shared encoder f_ξ, of the value stream v_η, and of the advantage stream a_ψ, $\theta = \{\xi, \eta, \psi\}$.

The learning procedure is (distributional Q-learning)

$$\min_\theta \; D_{\text{KL}}\left(\Phi_z d^{(n)}_t || d_{t;\theta}\right) \tag{53}$$

where

- $d^{(n)}_t$ is the multi-step target distribution

$$\begin{aligned}
d^{(n)}_t &= \Bigg(\underbrace{R^{(n)}_t + \gamma^{(n)}_t z,}_{\text{support}}\; \underbrace{p_{\bar{\theta}}(s_{t+n}, a^*_{t+n})}_{\text{returns' distr}}\Bigg) \\
R^{(n)}_t &= \sum_{k=0}^{n-1} \gamma^{(k)}_t R_{t+k+1} \\
\gamma^{(n)}_t &= \prod_{i=1}^{n} \gamma_{t+i} \\
a^*_{t+n} &= \arg\max_a \underbrace{z^T p_\theta(s_{t+n}, a)}_{q_\theta(s_{t+n},a)}
\end{aligned}$$

  with $\bar{\theta}$ parameters of the target network, θ parameters of the online network. Notice that the action a^*_{t+n} is selected according to the online network and evaluated using the target network (double Q-learning). Notice also that $R^{(n)}_t, \gamma^{(n)}_t$ are the n-step return and the product of discounted return before step n (multi-step learning).

- $d_{t;\theta}$ is the predicted distribution with the online network

$$d_{t;\theta} = (z, p_\theta(s_t, a_t))$$

- Φ_z is a L2-projection of the target distribution onto the fixed support z (Cranmer distance)

The equation (53) can be written as:

$$\min_\theta -\frac{1}{B}\sum_j \sum_i \hat{p}^i_{\bar{\theta}}(s_{j+n}, a^*_{j+n}) \log(p^i_\theta(s_j, a_j)) \tag{54}$$

where

- $\left\{ \left(s_j, a_j, R_j^{(n)}, \gamma_j^{(n)}, s_{j+n}\right) \right\}_j$ is the minibatch of n-step transitions. Once every n step a transition is accumulated into the replay buffer, with s_j, a_j being the initial state and action, $R_j^{(n)}, \gamma_j^{(n)}$ computed as before, s_{j+n} being the last state.
- $\hat{p}^i_{\bar{\theta}}(s_{j+n}, a^*_{j+n})$ is the projection onto the atom z_i of the distribution of returns computed with the target network (parameters $\hat{\theta}$) with the state s_{j+n} as input. Only the distribution on the action a^*_{j+n} is considered, selected with the online network (parameters θ) as $= \arg\max_a z^T p_\theta(s_{j+n}, a)$

$$\hat{p}^i_{\bar{\theta}}(s_{j+n}, a^*_{j+n}) = \left[\Phi_z \left(R_j^{(n)} + \gamma_j^{(n)} z, p_{\bar{\theta}}(s_{j+n}, a^*_{j+n}) \right) \right]_i$$

All linear layers of the networks are replaced with their noisy equivalent (noisy nets)

$$y = (b + Wx) + \left(b_{\text{noisy}} \odot \epsilon^b + (W_{\text{noisy}} \odot \epsilon^w) x \right) \tag{55}$$

where ϵ^b, ϵ^w are random variables (factorised Gaussian noise) and $\odot$ denotes the element-wise product.

Instead of sampling uniformly at random from the replay buffer, prioritized experience replay samples transitions with probability

$$p_t \propto \left(D_{\text{KL}} \left(\Phi_z d_t^{(n)} || d_{t;\theta} \right) \right)^\omega \tag{56}$$

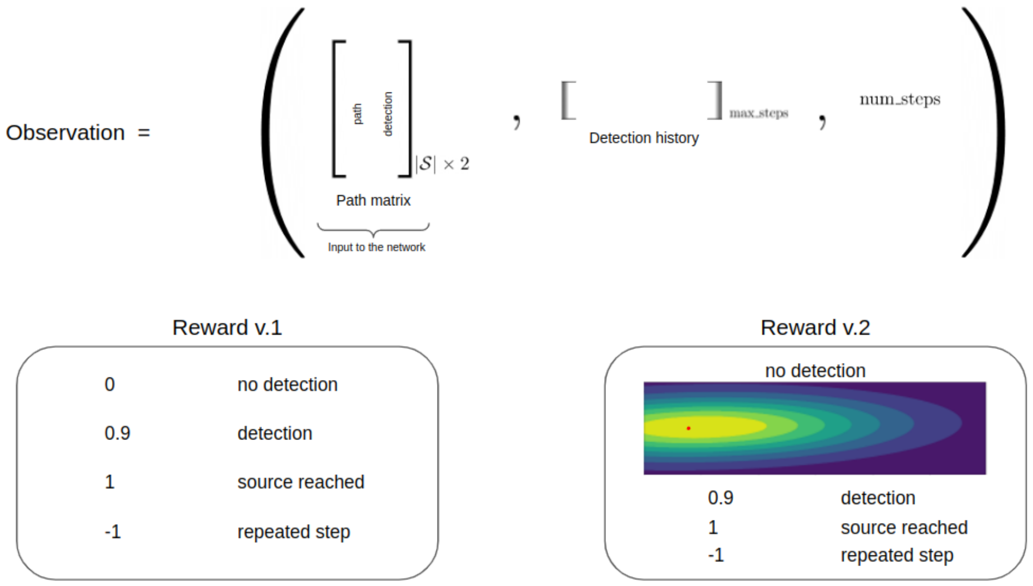
where ω is a hyper-parameter called importance sampling exponent that determines the shape of the distribution. Consider also that new transitions are inserted into the replay buffer with maximum priority, providing in any case a bias towards recent transitions.

Algorithm 3: Rainbow

```
Result: Estimate the optimal policy $\pi \approx \pi_*$
Input
- $N$ = capacity of the replay memory
- $M$ = number of episodes
- $T$ = number of time steps per episode
- $B$ = minibatch size
- $\varepsilon$ = small probability for the $\varepsilon$-greedy policy
- $N_{\text{atoms}}$ = number of atoms in the discrete support $z$
- $[v_{\min}, v_{\max}]$ = interval defining the support $z$
- $n$ = number of steps defining the multi-step transitions
- $L$ = learning period
- $C$ = network update period
- $\omega$ = importance sampling exponent
- Network architecture, optimization parameters
Output
- $q$
Pseudo-code
for episode = $1, \dots, M$ do
    Set the initial state $s_1$, initialize networks parameters $\theta, \bar{\theta}$
    for timestep $t = 1, \dots, T$ do
        Choose $a_t = \begin{cases} \text{random with probability } \varepsilon \\ \arg\max_a q_\theta(s_t, a) \text{ with probability } (1-\varepsilon) \end{cases}$
        Execute $a_t$ and observe reward $r_t$, new state $s_{t+1}$
        Store $n$-step transition $(s_t, a_t, R_t^{(n)}, \gamma_t^{(n)}, s_{t+n})$ in $D$
        Sample minibatch of $B$ transitions $\left\{ \left(s_j, a_j, R_j^{(n)}, \gamma_j^{(n)}, s_{j+n}\right) \right\}_j$ from $D$
        if $t$ mod $L = 0$ then
            Perform a gradient descent step:
            $\theta \leftarrow \min_\theta -\frac{1}{B} \sum_j \sum_i \hat{p}^i_{\bar{\theta}}(s_{j+n}, a^*_{j+n}) \log(p^i_\theta(s_j, a_j))$
        end
        if $t$ mod $C = 0$ then
            Update the target network $\bar{\theta} \leftarrow \theta$
        end
    end
end
```

6.3.3 Experiments

Problem setting Description of the problem setting here: observation, reward, DQN architecture, rainbow parameters, ecc.



Reward v.1

0	no detection
0.9	detection
1	source reached
-1	repeated step

Reward v.2

0.9	detection
1	source reached
-1	repeated step

Figure 19: Setting

Experiments with Reward v.1 Describe Figure 20

Experiments with Reward v.2 Describe Figure 21

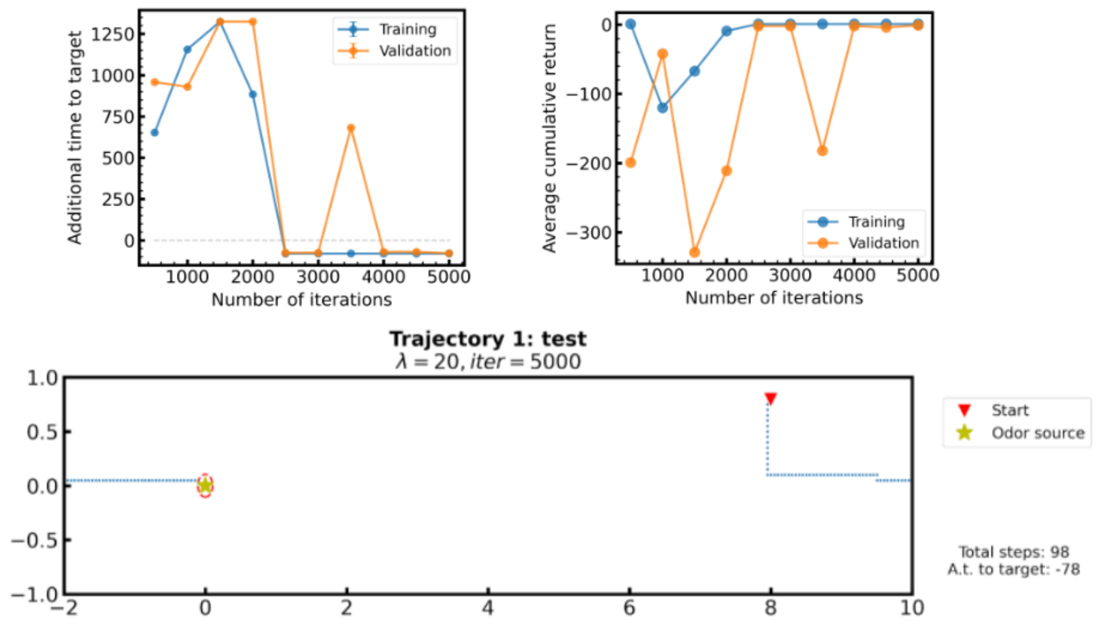


Figure 20: Reward v.1

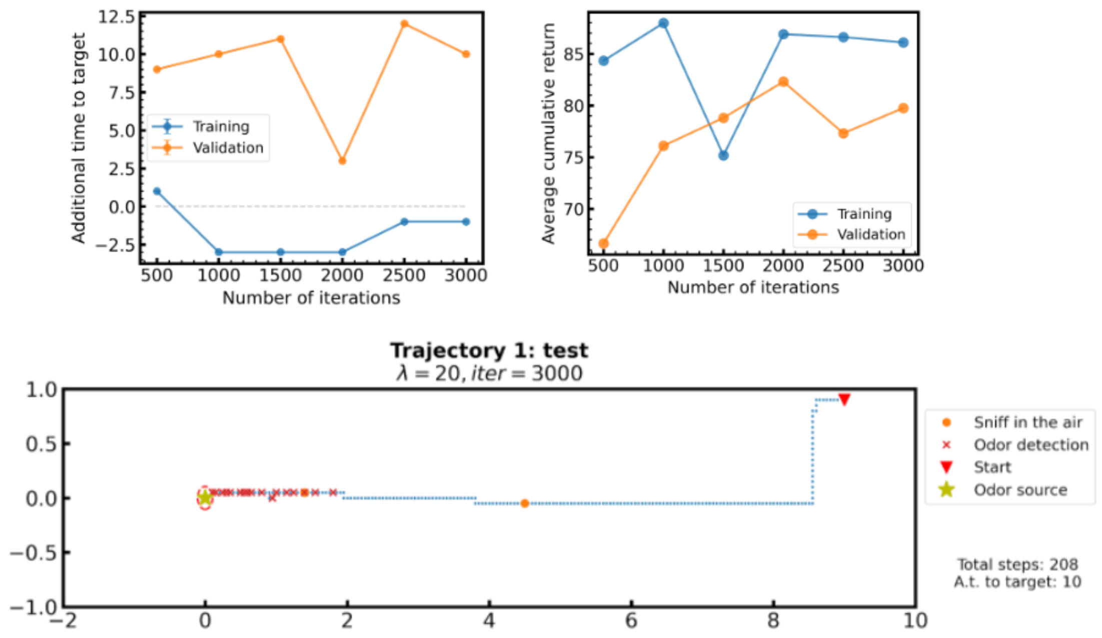


Figure 21: Reward v.2

References

[1] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

[2] Spaan, M. T., & Vlassis, N. (2005). Perseus: Randomized point-based value iteration for POMDPs. Journal of artificial intelligence research, 24, 195-220.

[3] Shani, G., Pineau, J., & Kaplow, R. (2013). A survey of point-based POMDP solvers. Autonomous Agents and Multi-Agent Systems, 27(1), 1-51.

[4] Shani, G., Brafman, R. I., & Shimony, S. E. (2007, January). Forward Search Value Iteration for POMDPs. In IJCAI (pp. 2619-2624).

[5] Rigolli, N., Reddy, G., Seminara, A., & Vergassola, M. (2021). Alternation emerges as a multi-modal strategy for turbulent odor navigation. bioRxiv.

[6] López, L. L., Vouloutsi, V., Chimeno, A. E., Marcos, E., i Badia, S. B., Mathews, Z., ... & i Lluna, A. P. (2011). Moth-like chemo-source localization and classification on an indoor autonomous robot. In On Biomimetics. IntechOpen.

[7] Sanders, J., & Kandrot, E. (2010). CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional.

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529-533.

[9] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... & Silver, D. (2018, April). Rainbow: Combining improvements in deep reinforcement learning. In Thirty-second AAAI conference on artificial intelligence.