Policy Networks for Non-Markovian Reinforcement Learning Rewards

ANTONELLA ANGRISANI, Sapienza, University of Rome
ANDREA FANTI, Sapienza, University of Rome
LUCA LOBEFARO, Sapienza, University of Rome

1 INTRODUCTION

A possible technique to produce an agent that can achieve temporal goals specified through LTL_f/LDL_f formulas is to use Reinforcement Learning (RL). However, the theoretical framework of most RL algorithms expects that the task can be modeled with a Markov Decision Process (MDP), a key assumption of which is that the next state and reward at each step only depend on the current state–action pair. This is clearly not true in general for temporal goals. Nonetheless, RL algorithms are often applied to "slightly" non-Markovian environments more or less successfully, by using deep learning models and simple workarounds for specific tasks. These workarounds, however, do not easily scale or adapt to other environments.

A more general approach for environments in which only the reward is non–Markovian is to produce an extended MDP. More specifically, a very convenient way to integrate a temporal goal in an MDP is to compute a DFA from the temporal formula for the non–Markovian goal, and then augment the original MDP state with the state of this automaton.

In practice, however, simply appending the automaton state to the environment observation which is fed to the agent can be problematic. For example, an RL agent learning to play a videogame would most likely observe the pixels on the screen, and thus employ Convolutional Neural Networks model(s); extending this input with an automaton state would have the unwanted effect of making the agent treat this information as "just another pixel".

The most obvious method to overcome this is to have a collection of separate "expert" networks, one for each automaton state, which are still fed the original environment observation, and are selected based on the current state of the automaton. The main downside of this approach, besides memory consumption, is training time: since all networks are separate, the steps needed to make them properly converge increases at least linearly with the dimension of the automaton state. Moreover, due to the nature of the goal, some automaton states may be more or less common, meaning that some networks may be trained more often while others are trained rarely.

Here we compared this baseline approach with sharing all non–output layers of the expert networks, by using the Advantage Actor Critic (A2C) algorithm on the `SapientinoCase` environment. In this case, the non–Markovian task for the agent, a planar unicycle robot, is to visit the colored cells of a continuous 2D grid in a specific order. This environment has a low–dimensional observation space, which allows to use simple Feed–Forward Neural Networks; moreover, since the task depends on a continuous 2D grid map, it allows to fine–tune the difficulty of the task as needed. In fact, we also compared results obtained by initializing the weights of the expert networks for harder maps by using weights previously trained on a simpler map (still exposing a non–Markovian reward).

All three authors contributed equally to this project.

Authors' addresses: Antonella Angrisani, Sapienza, University of Rome, angrisani.1703081@studenti.uniroma1.it; Andrea Fanti, Sapienza, University of Rome, fanti.1650746@studenti.uniroma1.it; Luca Lobefaro, Sapienza, University of Rome, lobefaro.1890195@studenti.uniroma1.it.

The full code is available on the github repository [1].

2 BACKGROUND

2.1 LTL_f/LDL_f for Non-Markovian Rewards

Typically to model and solve problems such as learning to perform well in unknown environments, Markov Decision Processes (MDPs) are used. We are talking about sequential decision-making model, based on the fact that the effects of an action depend only on the state in which it was performed and the reward of that state depends only on the previous action and state. An MDP is represented as a tuple $\mathcal{M} = \langle S, A, Tr, R\rangle$, where: S is a set of states; A a set of actions; Tr a transition function $Tr : S \times A \rightarrow Prob(S)$ that, for every state and action, returns a distribution over the next state; R the reward function $R : S \times A \longrightarrow \mathbb{R}$ that, when an action is performed in a state, specifies the real-valued reward received. A solution to an MDP is a policy that assigns an action to each state, taking into account past states and actions, and the value of policy is the expected sum of the discounted rewards. If a policy maximizes the expected sum of rewards for every starting state, it is the optimal one.

However, it is not always enough that the reward functions depend on the last state only: therefore, we need the non-Markovian rewards. To deal with this type of rewards and to introduce an implicit specification capable of expressing the infinite number of possible histories or futures of a state, a new formalism is illustrated in [2]: LDL_f, the Linear Dynamic Logic over finite traces, based on a standard temporal logic of the future. A traces is a sequence of states and actions.

2.1.1 Linear Dynamic Logic over finite traces. LDL_f is an extension of LTL_f, the Linear-time Temporal Logic over finite traces. The latter is merged with Regular Expressions (*RE*), and the resulting LDL_f turns out to be built as follows:

$$\varphi ::= tt|\neg\varphi|\varphi_1 \wedge \varphi_2|\langle\varrho\rangle\varphi$$
$$\varrho ::= \phi|\varphi?|\varrho_1 + \varrho_2|\varrho_1;\varrho_2|\varrho^*$$

where tt stands for logical true; ϕ is a propositional formula; ϱ denotes path expressions, which are *RE* over ϕ with the addition of the test construct $\varphi?$; $\langle\varrho\rangle\varphi$ states that, from the current step in the trace, there exists an execution satisfying the RE ϱ such that its last step satisfies φ.

LDL_f is more expressive than LTL_f, as it captures Monadic Second-Order Logic (MSO) (instead of First-Order Logic (FOL)), keeping the same computational characteristics. The encoding is in polynomial time, and the procedural constraints (for instance, sequencing constraints) are represented in a more natural way. Moreover, we can generate a minimal equivalent extended MDP and construct an automaton for tracking the satisfiability of a formula in a simple and compositional way: if a new reward formula is added, we optimize its automaton and add it to the current extended MDP; if the current MDP was minimal, the resulting MDP is minimal too. Finally, the forward construction of the automaton via progression ensures, on the base of the initial state, the generation of reachable states only, by pruning the others.

Each LDL_f formula can be associated with a Nondeterministic Finite Automata (*NFA*): the latter is represented as a tuple $A = \langle\Sigma, Q, q_0, \delta, F\rangle$, where: Σ is a finite nonempty alphabet; Q is a finite nonempty set of states; $q_0 \in Q$ is the initial states; δ is a transition relation $\delta \subseteq Q \times \Sigma \times Q$; $F \subseteq Q$ is the set of final states. The association is guaranteed if the NFA accepts exactly the traces satisfying the formula (correctness).
Let's see the algorithm LDL_f2NFA [2] (Figure 1) to compute the *NFA* given the LDL_f formula:

Assume that the LDL_f formula is in Negation Normal Form (*NNF*); ∂ is the auxiliary function, which takes as input an LDL_f formula φ in *NNF* (extended with auxiliary constructs F_ψ and T_ψ)

[1]https://github.com/lucalobefaro/reasoning_agent_project.git

```
 1: algorithm LDL$_f$2NFA
 2: input LDL$_f$ formula $\varphi$
 3: output NFA $\mathcal{A}_\varphi = (2^{\mathcal{P}}, Q, q_0, \delta, F)$
 4: $q_0 \leftarrow \{\varphi\}$
 5: $F \leftarrow \{\varnothing\}$
 6: if ($\partial(\varphi, \epsilon) = true$) then
 7:     $F \leftarrow F \cup \{q_0\}$
 8: $Q \leftarrow \{q_0, \varnothing\}$, $\delta \leftarrow \varnothing$
 9: while ($Q$ or $\delta$ change) do
10:     for ($q \in Q$) do
11:         if ($q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Theta)$) then
12:             $Q \leftarrow Q \cup \{q'\}$
13:             $\delta \leftarrow \delta \cup \{(q, \Theta, q')\}$
14:             if ($\bigwedge_{(\psi \in q')} \partial(\psi, \epsilon) = true$) then
15:                 $F \leftarrow F \cup \{q'\}$
```

Fig. 1. *LDL$_f$2NFA* algorithm

and a propositional interpretation Θ for $\mathcal{P}$, and returns a positive boolean formula whose atoms are φ subformulas:

$$
\begin{array}{rcl}
\partial(tt, \Theta) & = & true \\
\partial(f\!f, \Theta) & = & false \\
\partial(\phi, \Theta) & = & \partial(\langle\phi\rangle tt, \Theta) \quad (\phi \text{ prop.}) \\
\partial(\varphi_1 \wedge \varphi_2, \Theta) & = & \partial(\varphi_1, \Theta) \wedge \partial(\varphi_2, \Theta) \\
\partial(\varphi_1 \vee \varphi_2, \Theta) & = & \partial(\varphi_1, \Theta) \vee \partial(\varphi_2, \Theta) \\
\partial(\langle\phi\rangle\varphi, \Theta) & = & \begin{cases} \boldsymbol{E}(\varphi) \text{ if } \Theta \models \phi \quad (\phi \text{ prop.}) \\ false \text{ if } \Theta \not\models \phi \end{cases} \\
\partial(\langle\varrho?\rangle\varphi, \Theta) & = & \partial(\varrho, \Theta) \wedge \partial(\varphi, \Theta) \\
\partial(\langle\varrho_1 + \varrho_2\rangle\varphi, \Theta) & = & \partial(\langle\varrho_1\rangle\varphi, \Theta) \vee \partial(\langle\varrho_2\rangle\varphi, \Theta) \\
\partial(\langle\varrho_1; \varrho_2\rangle\varphi, \Theta) & = & \partial(\langle\varrho_1\rangle\langle\varrho_2\rangle\varphi, \Theta) \\
\partial(\langle\varrho^*\rangle\varphi, \Theta) & = & \partial(\varphi, \Theta) \vee \partial(\langle\varrho\rangle\boldsymbol{F}_{\langle\varrho^*\rangle\varphi}, \Theta) \\
\partial([\phi]\varphi, \Theta) & = & \begin{cases} \boldsymbol{E}(\varphi) \text{ if } \Theta \models \phi \quad (\phi \text{ prop.}) \\ true \text{ if } \Theta \not\models \phi \end{cases} \\
\partial([\varrho?]\varphi, \Theta) & = & \partial(nnf(\neg\varrho), \Theta) \vee \partial(\varphi, \Theta) \\
\partial([\varrho_1 + \varrho_2]\varphi, \Theta) & = & \partial([\varrho_1]\varphi, \Theta) \wedge \partial([\varrho_2]\varphi, \Theta) \\
\partial([\varrho_1; \varrho_2]\varphi, \Theta) & = & \partial([\varrho_1][\varrho_2]\varphi, \Theta) \\
\partial([\varrho^*]\varphi, \Theta) & = & \partial(\varphi, \Theta) \wedge \partial([\varrho]\boldsymbol{T}_{[\varrho^*]\varphi}, \Theta) \\
\partial(\boldsymbol{F}_\psi, \Theta) & = & false \\
\partial(\boldsymbol{T}_\psi, \Theta) & = & true
\end{array}
$$

where: E(φ) recursively replaces in φ all occurrences of atoms of the form F_ψ and T_ψ by E(ψ). $\partial(\varphi, \epsilon)$ is defined inductively, except for $\partial(\langle\phi\rangle\varphi, \epsilon) = false$ and $\partial([\phi]\,\varphi, \epsilon) = true$.

Algorithm *LDL$_f$2NFA* is correct for every finite trace $\pi : \pi \models \varphi$ *iff* $\pi \in \mathcal{L}(\mathcal{A}_\varphi)$ (**Theorem 1** [2]). It terminates in at most an exponential number of steps, and generates a set of states $\mathcal{S}$ whose size is at most exponential in the size of the formula φ.

The *LDL$_f$* formula can be then transformed into a Deterministic Finite Automata (*DFA*) with an on-the-fly approach, without constructing $\mathcal{A}_\varphi$: it is possible by progressing all states that the *NFA* can be in, after consuming the next trace symbol, and accept the trace only if, once it has been completely read, the set of possible states contains a final state.

2.1.2 Non-Markovian Rewards. A Non-Markovian-Reward decision process (NMRDP) is represented as a tuple $\mathcal{M} = \langle S, A, Tr, R \rangle$, where: S is a set of states; A a set of actions; Tr a transition function $Tr : S \times A \longrightarrow Prob(S)$ that, for every state and action, returns a distribution over the next state; R the reward function $R : (S \times A)^* \longrightarrow \mathbb{R}$ that is a real-valued function over finite state-action sequences. The value of policy is the expected value of infinite traces. The value of an infinite trace π is:

$$v(\pi) = \sum_{i=1}^{|\pi|} \gamma^{i-1} R(\langle \pi(1), \pi(2), ..., \pi(i) \rangle)$$

where $0 < \gamma \leq 1$ is the discount factor and $\pi(i)$ denotes the pair (s_{i-1}, a_i).

The value of a policy ρ, given an initial state s_0 is:

$$v^\rho(s) = E_{\pi \sim \mathcal{M}, \rho, s_0} v(\pi)$$

Using a set of pairs $(\varphi_i, r_i)_{i=1}^m$, LDL_f provides an intuitive language for specifying the reward implicitly: if the current (partial) trace is $\pi = \langle s_0, a_1, ..., s_{n-1}, a_n \rangle$, the agent receives at s_n a reward r_i for every formula φ_i satisfied by π [2], and

$$R(\pi) = \sum_{i \leq 1 : \pi \models \varphi_i} r_i$$

An NMRDP $\mathcal{M} = \langle S, A, Tr, R \rangle$ is equivalent to an extended MDP $\mathcal{M}' = \langle S', A', Tr', R' \rangle$ (**Theorem 2** [2]), if there exist two functions $\tau : S \longrightarrow S$ and $\sigma : S \longrightarrow S'$ such that:

- $\forall s \in S : \tau(\sigma) = s$;
- $\forall s_1, s_2 \in S$ *and* $s_1' \in S'$: *if* $Tr(s_1, a, s_2) > 0$ *and* $\tau(s_1') = s_1$, *there exists a unique* $s_2' \in S'$ *such that* $\tau(s_2') = s_2$ *and* $Tr(s_1', a, s_2') = Tr(s_1, a, s_2)$;
- *for any feasible trajectory* $\langle s_0, a_1, ..., s_{n-1}, a_n \rangle$ of $\mathcal{M}$ *and* $\langle s_0', a_1, ..., s_{n-1}', a_n \rangle$ of $\mathcal{M}'$, *such that* $\tau(s_i') = s_i$ *and* $\sigma(s_0) = s_0'$, *we have* $R(\langle s_0, a_1, ..., s_{n-1}, a_n \rangle) = R'(\langle s_0', a_1, ..., s_{n-1}', a_n \rangle)$.

So, we first have to construct for each reward formula φ_i its corresponding *DFA* and then define the equivalent extended MDP $\mathcal{M}' = \langle S', A', Tr', R' \rangle$, where: $S' = Q_1 \times ... \times Q_m$ is a set of states; $A' = A$ as set of actions; a transition function $Tr' : S' \times A' \times S' \longrightarrow [0, 1]$ defined as

$$Tr'(q_1, ..., q_m, s, a, q_1', ..., q_m', s') = \begin{cases} Tr(s, a, s') & \text{if } \forall i : \delta_i(q_i, s) = q_i' \\ 0 & \text{otherwise} \end{cases}$$

and the reward function $R' : (S' \times A) \longrightarrow \mathbb{R}$ is

$$R(q_1, ..., q_m, s, a) = \sum_{i : \delta_i(q_i, s) \in F_i} r_i$$

It is also easy to define an equivalent policy on an NMRDP $\mathcal{M}$: if ρ' is an optimal policy for an equivalent MDP $\mathcal{M}'$, then policy ρ for $\mathcal{M}$ that is equivalent to ρ' is optimal for $\mathcal{M}$ (**Theorem 3** [2]).

As mentioned above, two of the main features of this approach are (1) minimality, (2) compositional and (3) progression.
If the current MDP was minimal, the resulting MDP is minimal too: to ensure this fact, it is enough that each *DFA* $\mathcal{A}_\varphi$ is minimal (**Theorem 4** [2]).
If a new formula is added, we need not change the MDP, but simply extend it with one additional component. The states of the extended MDP are simply vectors that represent the state of the NMRDP and the state of the automaton for each reward formula.
An automaton that tracks the satisfaction of a reward formula can be constructed by progression,

but by using progression the constructed states may not be minimal. So, one may build the automata for the reward formulas, minimize them off-line before starting search, and then apply progression using the minimal automata.

2.2 Reinforcement Learning

Reinforcement Learning is a machine learning sub-field for which an agent has to learn what to do so as to maximize a numerical reward signal. In order to accomplish this task it is important that the agent has a model of the actions that can be taken into the environment in which it acts. But the learner is not told which actions to take, instead it must discover which actions yield the most reward by trying them. This leads the agent to a good behavior, learning skills incrementally, using trial-and-error experience. Typically the way in which the agent interact with the environment is modeled in the way showed in Fig 2 known as the Markovian stochastic control process.

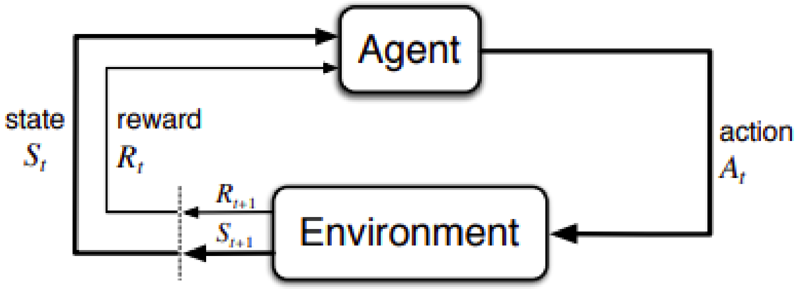

Fig. 2. The agent-environment interaction [7]

As we can see from the figure, the agent interacts with the environment (for which a model can be used or not, according to the approach used) through actions which result in a reward signal and an observation. The reward signal is used from the agent as a guide to the goal, because its job is to maximize it. The observation is used, instead, to understand what is the state of the environment after the action is taken. This kind of stochastic control process is Markovian because it has the following Markov property:

- $P(S_{t+1}|S_t, A_t) = P(S_{t+1}|S_t, A_t, ..., S_0, A_0)$
- $P(R_t|S_t, A_t) = P(R_t|S_t, A_t, ..., S_0, A_0)$

The way in which the agent selects actions, given a state, is due to a function called policy. Learning a policy that allows the agent to obtain the higher reward possible is the goal of Reinforcement Learning. To find an optimal policy, we have to maximize an expected return, that is a way to look at rewards, not only obtained with the current action, but also in the future. This is the V-value function $V^\pi(s) : S \longrightarrow \Re$:

$$V^\pi(s) = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, \pi] \tag{1}$$

where γ is the discount factor and essentially determines how much the reinforcement learning agents cares about rewards in the distant future relative to those in the immediate future. If $\gamma = 0$, the agent will be completely myopic and only learn about actions that produce an immediate reward. Then, the optimal expected return can be defined:

$$V^*(s) = \max_{\pi \epsilon \Pi} V^\pi(s) \tag{2}$$

We can also define the Q-value function $Q^\pi(s,a) : SxA \longrightarrow \Re$:

$$Q^\pi(s,a) = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a, \pi] \tag{3}$$

Similarly to the V-value function, the optimal Q-value function can be defined as follow:

$$Q^*(s,a) = \max_{\pi \epsilon \Pi} Q^\pi(s,a) \tag{4}$$

Then, the optimal policy can be obtained:

$$\pi^*(s) = \max_{a \epsilon A} Q^*(s,a) \tag{5}$$

To find these optimal functions the agent can be able to select actions both exploiting the acquired information and exploring new actions, in order to estimate the value function for states and actions not yet explored. So a Reinforcement Learning training stage results in cycles during which the agent performs some actions and estimate the value function on the way.

The two main families of Reinforcement Learning are the model-based algorithms and the model-free algorithms. The first maintain a model of the environment, the second not. Then we can have online algorithms and offline algorithms, according to the fact that we use data accumulated in a memory before the training stage, or use data during the experience of the environment. We can also have on-policy and off-policy algorithms, the first use the same policy applied on the environment to find the optimal policy, the second exploit a policy to experience the environment and optimize another one. Another subdivision is between policy-based methods and value-based methods. A taxonomy of the main algorithms used today in Reinforcement Learning is shown in Figure 3.

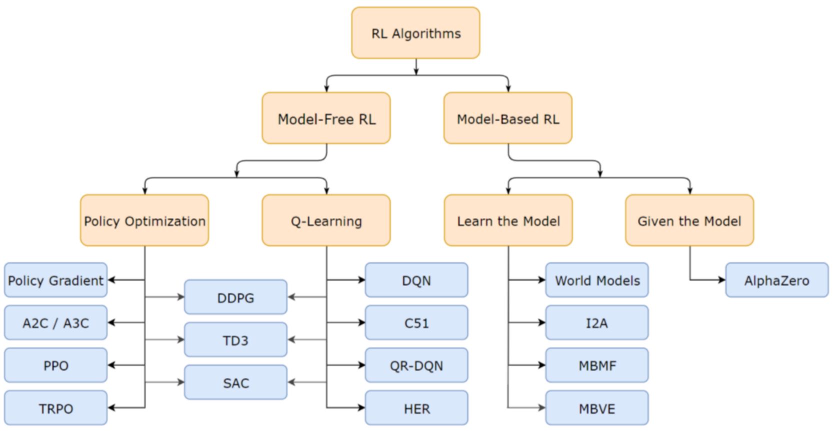

Fig. 3. Reinforcement Learning taxonomy [1]

For our project we use A2C, an algorithm of the family of the Actor-Critic, that is explained in details in Section 2.3.

2.3 Actor-Critic

Today, it is common to use Deep Learning approaches to perform Reinforcement Learning [3]. One of the most common algorithm that uses Neural Networks is Actor-Critic (AC). AC is a policy gradient method which aim is to recursively estimate a value function for the current policy and then update the policy to better exploit the information encapsulated in the estimated value function.

To do this AC uses two non-linear neural networks function approximators: an actor and a critic. The actor network approximates the policy function, whereas the critic network approximates the value function, estimated on the policy given by the actor. This method optimizes the policy that is used to experience the environment, for this reason it is a on-policy algorithm.

One of the main algorithms of the Actor-Critic family is the Advantage Actor Critic (A2C). In order to understand how it works it is useful to do some consideration about the policy gradient:

$$\nabla_\theta J(\theta) = E_\tau[\sum_{t=0}^{T-1} \nabla_\theta log\pi_\theta(a_t|s_t)G_t] \tag{6}$$

that is the gradient used to update the policy parameters. Using Monte Carlo updates (taking random samples) to update the policy with this gradient, introduces high variability in log probabilities and cumulative reward values, because each trajectory during training can deviate from each other at great degrees. This will cause noisy gradients and unstable learning, bringing the networks to a non-optimal direction. Other than this, if the cumulative reward is 0, then both "goods" and "bad" actions will not be learned. For this reason a "baseline" called Advantage value is introduced:

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)A(s_t, a_t) = r_{t+1} + \gamma V_v(S_{t+1}) - V_v(s_t) \tag{7}$$

obtaining the following update equation:

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta log\pi_\theta(a_t|s_t)(r_{t+1} + \gamma V_v(S_{t+1}) - V_v(s_t)) \tag{8}$$

With this in mind then it is easy to use two neural networks to approximate the policy and value functions. In particular the agent will first interact with the environment and collect state transitions, then, after n-steps, or at the end of the episode, calculates updates for both networks, using the policy gradient obtaining, in this way, a new policy, that is a little bit closer to the optimal policy. Then the cycle is repeated, but using the new policy, and after a certain number of cycles the optimal policy will be reached. It is important that the output of the policy network is a probability distribution among all the actions that can be taken in the current state. At the beginning, the covariance of this probability will be large, to encourage exploration, whereas, after a certain amount of updates, the covariance will diminish. The optimal policy is a policy with a probability of 1 on only the optimal action for the current state.

3 AGENT AND ENVIRONMENT

Our experiments were conducted on the `SapientinoCase` environment, a special instance of `Sapientino` [4] with a non–Markovian reward, while the A2C agent used feed–forward neural networks in different configurations for its actor and critic models. This section describes the details of the environment and its goal, along with the architectures we employed for the agent models.

3.1 The `SapientinoCase` environment

In the `SapientinoCase` environment, a unicycle–like robot navigates a grid of cells, which can be either empty, full, or colored; full cells cannot be traversed by the robot, forcing it to move around them. Besides movement actions, the agent can perform the special *beep* action, which "visits" the cell it is standing on; the (non–Markovian) goal is to visit colored cells in a specific order. The order is given as a parameter to the environment, together with the actual grid map. The reward is only given when the agent has successfully visited the last correct colored cell.

The base environment, without considering the goal, has a continuous observation space S, with elements being in the form (x, y, c, s), where:

- x and y are the continuous coordinates of the robot position in the grid, with $(x, y) = (0, 0)$ being the top leftmost cell;
- c and s are respectively the sine and cosine of the robot orientation angle.

The action space A, on the other hand, is discrete, composed by the 3 movement actions (*turn left*, *turn right*, *forward*), plus the special actions *no-op* and *beep*; any action is represented simply as an integer in 0..4. The transition model $Tr : S \times A \longrightarrow S$ of the base environment is deterministic, defined by

$$Tr((x, y, \cos\theta, \sin\theta), 0) = (x, y, \cos(\theta + \delta_\theta), \sin(\theta + \delta_\theta))$$

$$Tr((x, y, \cos\theta, \sin\theta), 1) = (x, y, \cos(\theta - \delta_\theta), \sin(\theta - \delta_\theta))$$

$$Tr((x, y, \cos\theta, \sin\theta), 2) = (x + \delta_x \cos\theta, y + \delta_y \sin\theta, \cos\theta, \sin\theta)$$

$$Tr(s, 3) = s$$

$$Tr(s, 4) = s$$

Note that the model of the environment is not exploited in the agent algorithm which assumes that the transition model is unknown. To integrate the non–Markovian reward in the environment MDP model, we use an extended MDP approach, based on [2]: we define the appropriate temporal goal using an LDL$_f$ formula, compute the associated DFA, and finally extend the original MDP of the environment with this automaton. The resulting extended environment will have a deterministic transition model Tr' defined as

$$Tr'((s, s_a), a) = (Tr(s, a), Tr_a((s, s_a), a))$$

where Tr_a and s_a are the transition model and state of the automaton, respectively.

In the implementation, since the goal is simple, the automaton was generated manually, without using a temporal formula. It has $n + 2$ states, one for each color plus one for the initial state and one for the “sink” state. Instead of directly specifying the general transiton model for the automaton, a visual example of the generated DFA for a 2 color goal is depicted in figure 5. For convenience, a state of the DFA is represented as a single integer in $0..n+1$; note that the state 2 always corresponds to the “sink” state, regardless of the number of colors. The “sink” state is reached whenever the robot visits the target colors in the wrong order, or visits the same color twice.

To optimize the available resources, we decided to end episodes when the agent reached either the goal or the “sink” state, since the subsequent states do not carry any useful information for the training. We also enforced a maximum number of time steps available to complete (or fail) the goal; see the section 4 for details.

3.2 Agent models

Instead of directly feeding the extended state to a single network, we instead used a collection of networks, one for each automaton state, applying this approach to both the actor and critic models. In the baseline implementation, these were trained from scratch and their parameters were not shared. As a possible improvement, we also experimented with architectures in which the weights of the first layers were shared across the networks; for details, see section 4.

We used the PyTorch library [6] to define and train the network models, by defining two `nn.Module` subclasses, one for the actor, and one for the critic. We implement the A2C algorithm by ourselves using the original paper [5]. Each keeps a `nn.ModuleList` as a field, which contains the networks associated to each automaton state. The selection is then performed in the `forward` method, by simply indexing the module list by the automaton state and passing the environment observation to the network.

4 EXPERIMENTS

In this section all the experiments performed on the task are exposed and the results illustrated.

A first try, with a simple map, in a Markovian setting, with a single color in the SapientinoCase environment, results in a very fast training, as expected. In only 2700 episodes, with a maximum number of time-steps of 300, a learning rate of 1e-3 and gamma with a value of 0.99, the agent learn how to perform the task. The map used for this task is shown in Figure 4.

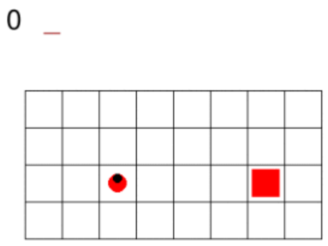

Fig. 4. Map used for the single color experiment

To introduce a non-Markovian behaviour, the second experiment consists in a two color map (see Figure 6). In particular two approaches are tried, with different network configurations. In the first case four separate networks are used, one for each state of the automaton (see Figure 5).

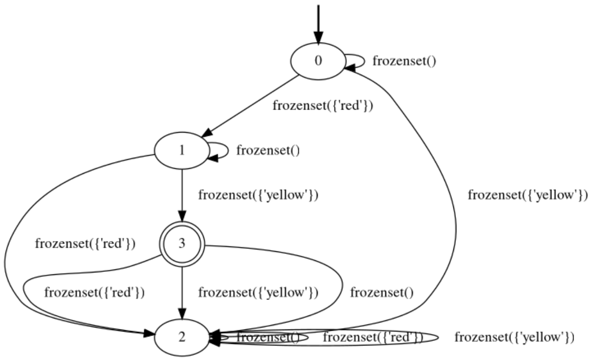

Fig. 5. Automaton for the two colors case

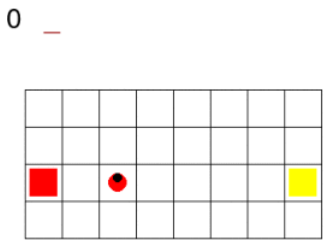

Fig. 6. Map used for the two colors experiment

In the second case, a single, shared network is used, with a 32 feature vector as output. Then, this shared feature vector is used to classify the result of the relative network (the same approach is used for both the actor and the critic network), with a fully connected layer. The same configurations (400 time-steps maximum limit, a learning rate of 5e-4 and a gamma value of 0.99) are used for both cases. The discounted rewards evolution during training are showed in the Figures 7 and 8. As we can see, the behaviour and the number of episodes needed to learn the task are the same with a little bit of additive noise in the shared network setting. But there is an advantage in using the shared networks, because less weights are needed and so less memory is used, obtaining a lightweight model.

Then, a sort of transfer learning approach is tried, in order to understand if the behaviour learned is easy to generalize. The trained weights, resulting from the two colors case training, with separate networks, are used to initialize the agent for which a slightly different environment is presented. First, a wall is introduced in the environment (see Figure 9), then the initial position of the agent is modified (see Figure 10). As we can see from the Figures 11 and 12 both agents are able to re-use the knowledge acquired during the base case to learn how to accomplish the tasks, in a sensible less amount of episodes with respect to a "from scratch" training. In particular in Figure 11 we can notice that initially the agent perform the actions learned with the previous environment, so never obtain a positive reward. Then, after about 7000 episodes, starts to deviate from the initial behaviour and adapts to the new environment. The same parameters of learning rate, time-steps limit and gamma of the base case are used.

A more complex environment is also tried, with three colors and a separate network for each state of the automaton in Figure 13. The evolution of the training is showed in Figure 15. As we

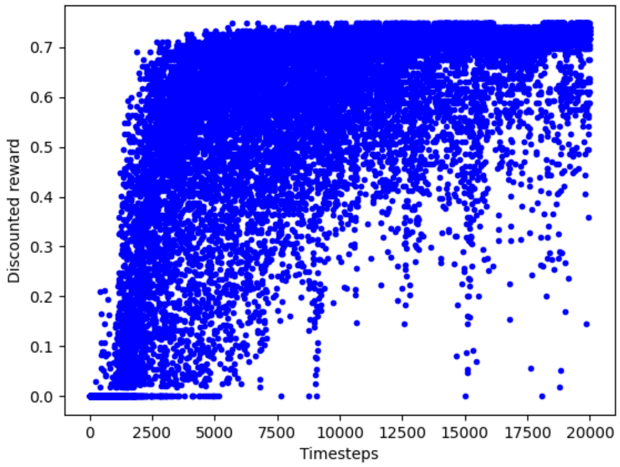

Fig. 7. Evolution of the discounted rewards during training for the two colors case with separated networks

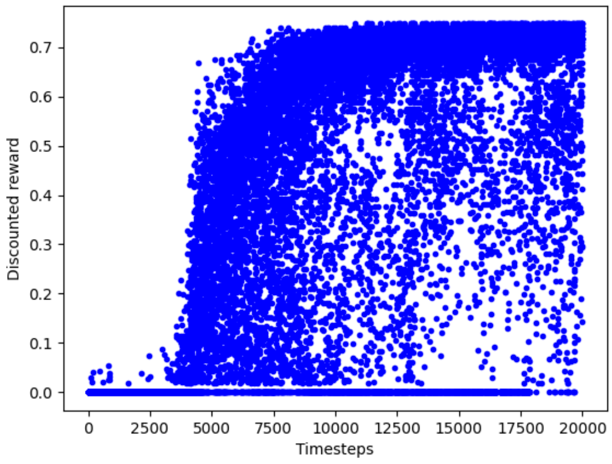

Fig. 8. Evolution of the discounted rewards during training for the two colors case with unified networks

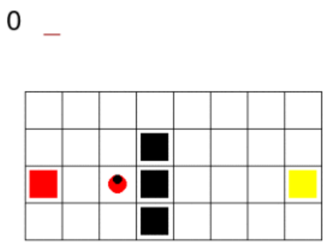

Fig. 9. Map used for the transfer learning case with a wall

can see from the Figure 14 the map used is too simple, so the task is easy to accomplish also with a Markovian agent. More complex environments results in a very low amount of initial "correct"

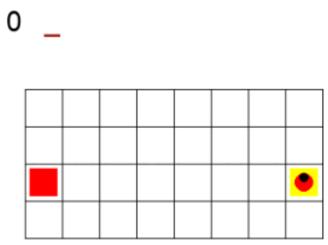

Fig. 10. Map used for the transfer learning case with a different starting position for the agent

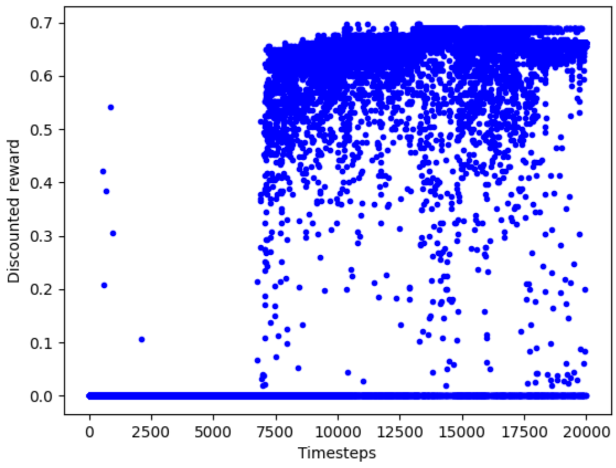

Fig. 11. Discounted rewards during training for the transfer learning case with a wall

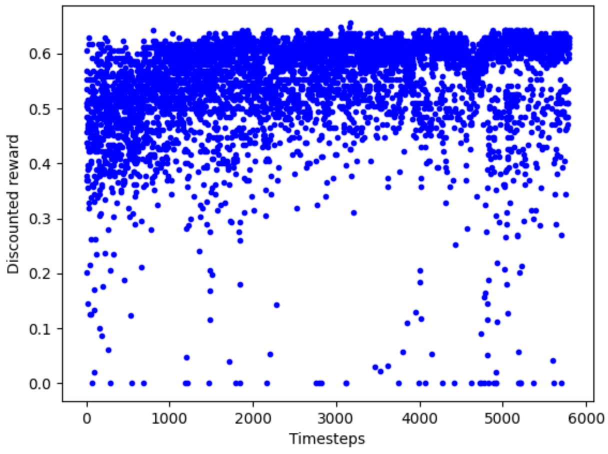

Fig. 12. Discounted rewards during training for for the transfer learning case with a different starting position for the agent

examples, so, with a policy learning is not able to learn a good behaviour. A possible solution is to use reward shaping in order to guide the agent during the training, but this approach was not of our interest.

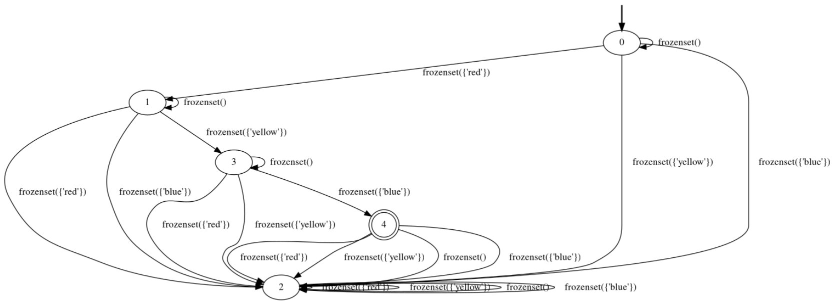

Fig. 13. Automaton for the three colors case

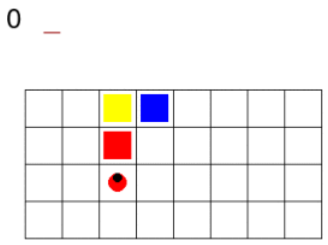

Fig. 14. Map used for the three colors experiment

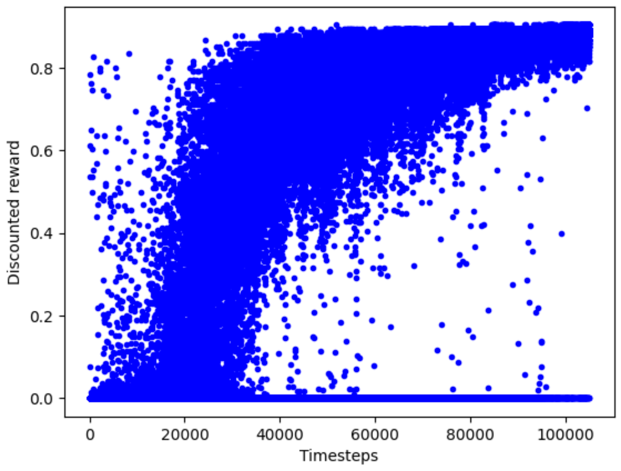

Fig. 15. Evolution of the discounted rewards during training for the three colors case with separated networks

5 CONCLUSIONS

Analyzing the results shown before we can notice that, using a shared neural architecture we are able to accomplish a non-Markovian task in an amount of time that is similar to the baseline approach. This has the advantage of using a less memory consuming model, with the same results.

We have also shown that, using a pre-trained model we can solve harder configurations of the same task in a sensibly less amount of time with respect to a "from scratch" training.

A possible extension is to try with an harder environment as a SapientinoCase with more than two colors or a different one (e.g. Atari gym environment) with reward shaping.

REFERENCES

[1] [n.d.]. *A Taxonomy of RL Algorithms.* https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

[2] Ronen I. Brafman, Giuseppe De Giacomo, and Fabio Patrizi. 2018. LTL_f/LDL_f Non-Markovian Rewards. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018* (2018), 1771–1778. http://www.diag.uniroma1.it/degiacom/papers/2018/aaai18bdp.pdf

[3] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. 2018. An Introduction to Deep Reinforcement Learning. *Foundations and Trends® in Machine Learning* 11, 3-4 (2018), 219–354. https://doi.org/10.1561/2200000071

[4] Giuseppe De Giacomo, L. Iocchi, Marco Favorito, and F. Patrizi. 2019. Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications. In *ICAPS*.

[5] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. arXiv:1602.01783 [cs.LG]

[6] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[7] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html