Sapienza
Università di Roma

Fluents valuation in Deep Reinforcement Learning and logic for temporal goals

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea Magistrale in Artificial Intelligence and Robotics

Candidate
Roberto Cipollone
ID number 1528014

Thesis Advisor
Prof. Giuseppe De Giacomo

Academic Year 2019/2020

Thesis not yet defended

Fluents valuation in Deep Reinforcement Learning and logic for temporal goals
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: cipollone.rt@gmail.com

Contents

Chapter 1

Introduction

In Artificial Intelligence (AI), among the many approaches for building intelligent agents, we can distinguish those mainly focused on knowledge and planning, and those that mainly try out different actions to discover the goodness of their outcomes. With the former, we refer to those developed from *classical planning*, while the latter is the recently-successful field of *reinforcement learning*. While they can be integrated, they use quite different techniques. However, they share a common basic need: the agent must be able to perceive meaningful events happening in the outside world.

In planning, in almost every practical case, there is some form of partial observability or nondeterminism. So, agent's observations become essential [20]. Observing, however, does not simply refers to reading a raw input from the sensor. Instead, it means "grounding" all symbols that compose the abstraction the agent adopts: essentially, all symbols representing conditions which happen to be true in the environment, should become true for the agent. We refer to these symbols with the term *fluents*. Fluents are atomic propositions can change in time, whose valuation should always reflect the current state of the world.

We could argue that reinforcement learning does not require such valuations. Still, rewards and punishments must be somehow supplied in response to desirable and undesirable events. We could think of providing these feedbacks with programmed ad-hoc conditions, but this can be done just for the simulations we create. As we will see, when the agent needs a component that is based on logic and reasoning, we still need to valuate the truth of fluents, even in the context of reinforcement learning. In fact, the most successful approaches mix these components somehow.

This thesis addresses the problem of valuating fluents from complex observations of the environment. However, this is a general topic and we'll only work with specific classes of fluents and observations. Every choice or assumption that restricts the applicability of this method will be pointed out along the text. The first distinction to do is that we'll only work with games.

1.1 Reinforcement Learning in games

Games in AI are a class static environments with discrete actions. They have always been a classic benchmark for AI, because they provide various levels of complexity, they have few and strict rules and are easy to implement and simulate.

Reinforcement learning (RL) is a field of AI that has shown to be successful for many games. This is the learning method adopted here. In RL, the agent tries out different actions and observes the reward received. Its goal is to learn the optimal policy, the one maximizing the cumulative reward over the whole episode. Most RL algorithms assume that observations and rewards can be modelled with a Markov Decision Process (MDP). This means that: the sequence of states create a Markov chain (the next state only depends on the previous state and action); the rewards only depend on the current state; the observations are an exact representation of the current state. Many learning algorithms exist for this setting [22].

Neural Networks (NN) have brought new possibilities for RL: in Deep Reinforcement Learning, the agent employs a neural network as a very expressive approximation to the quantities it is trying to learn [8]. The Q-value, for example, is a classic quantity in RL that estimates the expected cumulative reward from each pair of observation and action. A Deep Q-Network (DQN) is able to learn this estimate for a complex observation such as the frame of a video game, which is a high-dimensional input [17] that would be hard to manage without neural networks.

All models and experiments in this thesis use games from the Atari 2600 collection. The framework adopted is an interface for the Atari simulator [2] that maps actions to controller inputs and returns images of the game as observations. This is almost the same condition a human player faces when playing the same games. Other successful works also read the current number of lives the player has. Other than that, no internals are employed to simplify the task of the agent.

The reinforcement learning algorithm adopted in this thesis is a deep variant of Double DQN that solves few issues with simple DQN [25]. The motivation of this choice is that this is a relatively simple algorithm, based on DQN, which has also proven to be successful for the specific environments that we'll use in our experiments [16]. In fact, among Q-Network algorithms, the only ones that clearly achieve superior performances in most games adopt a combination of all variants [12].

Not all games inside the Atari 2600 collection are equally hard to solve. Reinforcement learning agents can be trained to achieve higher performances with respect to an average human player, mostly for environments with static map and background, and simple strategies. Many other games, instead, require the agent to remember previous steps and observations, for example, in exploration tasks. One notable example is *Montezuma's Revenge*, in which agents could not improve in any way [16].

Other approaches could succeed in this game with additional information able to guide the agent. For example, with carefully chosen initializations and examples from human experts [15].

One issue with games like Montezuma's Revenge, is that they require long sequences of correct actions before rewarding the agent. This is called a *sparse reward*. However, there is a more fundamental problem to be considered first: observations and rewards, together, can't define a Markov Decision Process, because it is essential for the agent to remember some informations collected during the game. For example, the Montezuma's agent may walk to the right only if it *remembers* that the door in the right room has been previously opened. This additional ability is required because of partial observations: a view of the current room can can't be considered a complete state of the game, sufficient to predict future rewards.

The setting just described can be modelled with a Non-Markovian Reward Decision Process (NMRDP). Fortunately, it is possible to cast any NMRDP as a MDP, if enough information about the history is included in each observation. In order to render this transformation feasible, we must include as few additional data as possible, still with an exhaustive state with respect to the reward. As humans, we understand which sequences lead to rewards. So, an elegant way to do this, is to declare such sequences with *temporal logics* [1]. As we will see, by tracking the satisfaction of such temporal formula, we can provide enough information to the agent so to employ standard algorithms developed for MDPs [3][6].

This type of construction can be considered as a "logic component" inside the agent, as we've previously called it. While an abstraction like this is powerful, it is essential to correctly valuate the symbols it uses, in order to reason about the current situation. This is a complex task for environments with rich observations, as those allowed by Deep Reinforcement Learning.

1.2 Objective of this work

The main purpose of this work is to devise and test a mechanism able to learn functions which valuates the fluents we define. Specifically, learn a function that computes the truth value for a set of boolean conditions, given a frame of an Atari game. Among the many different ways to accomplish this, the most interesting techniques are those which pose the least number of assumptions on the specific environment. In this respect, the following are important achievements of this work to be highlighted:

- Fluents are selected first. Then, the function to evaluate them is trained from a description of each fluent. This is harder to do than just training a features extractor and manually trying to associate a meaning to each feature.
- To describe the fluents we use temporal logic over finite traces such as LTL_f and LDL_f. These are employed as tools to formalize any type of temporal constraints the fluents are always expected to satisfy. The use of such logics for this purpose can be a really generic approach. This thesis is an initial investigation about this possibility. As a description of a fluent, we must consider everything that guides the training process. So, we will certainly consider other types of hints that is useful to include, such as visual hints.
- The training algorithm won't require any manual annotation, nor labelled datasets at all. The main idea is that, inside the agent, two components should coexist: the player and the observer. While the player explores the environment, the observer can be trained from the images received, without further intervention.

The second goal of this thesis is to demonstrate how such trained features can be exploited by a Reinforcement Learning agent to solve hard games. Tests will be conducted on Montezuma's Revenge, a game known to be difficult in this class [16]. In this thesis:

- We provide a flexible implementation of the construction described in [3][6], for temporal goals.
- A deep agent architecture is proposed to merge the technique above for the Deep Reinforcement Learning case.
- This implementation is then used to specify a temporal goal in LDL_f, sufficient to guide the agent through hard environments.

1.3 Results

1.4 Structure of the thesis

Chapter 2

Deep Reinforcement Learning for non-Markovian goals

2.1 Reinforcement Learning

In this section we will briefly review the most important aspects of classic *Reinforcement Learning* (RL). These concepts are relevant because they are also found in Deep Reinforcement Learning (Deep RL), which is a central component of the agent we will design. Excellent references for these topics are [22], [21], and [18] for graphical models.

In AI, we commonly isolate two entities, the agent and the environment, which continuously interact. At each instant, the agent receives observations from the environment and it executes actions in response. In RL specifically, the agent observes the current state of the environment and a numerical reward. The environment produces high rewards in response to desirable events. The agent's goal is to maximize the rewards received. The basic setup is illustrated in Figure 2.1.

Most RL algorithms assume that the environment dynamics can be modelled with a *Markov Decision Process* (MDP). They do so, because under the independence assumptions taken by MDP, it's possible to efficiently find the optimal agent's policy. A Markov Decision Process is a tuple $\langle S, A, T, R, \gamma \rangle$, where: S is the set of states

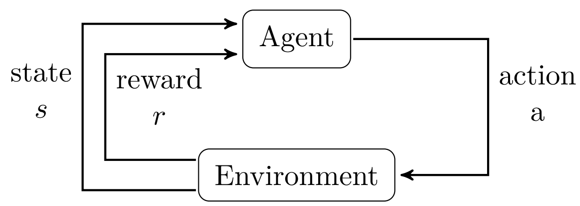

Figure 2.1. How agent and environment interact in RL.

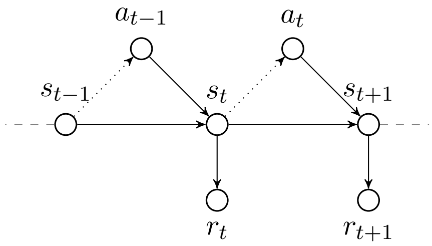

Figure 2.2. The directed graphical model of a MDP.

of the environment; A is the action space; $T : S \times A \times S \to \mathbb{R}$ is the transition function, which, for $T(s_t, a_t, s_{t+1})$, returns the probability $p(s_{t+1} \mid s_t, a_t)$ of the transition $s_t \xrightarrow{a_t} s_{t+1}$; $R : S \times A \times S \to \mathbb{R}$ is the reward function; and $\gamma \in [0, 1]$ is called "discount factor"[1].

In a RL problem, the functions T and R are unknown. The agent can only learn them by observing the samples it receives from the environment. However, by assuming that they can be modelled with functions $S \times A \times S \to \mathbb{R}$, we introduce some Markov assumptions. In particular, we assume that the next state of the environment is conditionally independent on the whole history, given the previous state and action: $s_{t+1} \perp s_0, \dots, s_{t-1} \mid s_t, a_t$. Similarly, the reward only depends on the last transition of the environment. Although it's not required by the model, it is common that rewards are computed just from desirable configurations of the environment s_t, not from specific transitions (s_t, a_t, s_{t+1}). All of these assumptions are summarized in the Directed Graphical Model (DGM) of Figure 2.2. In a DGM, directed edges indicate direct conditional probabilities, while missing arcs indicate conditionally independent variables. In Figure 2.2, the lack of any arrow between s_{t-1} and s_{t+1} means that future states, hence the rewards, do not depend on the past history, given the current state s_t. This is the essence of a Markov assumption.

Example 1. Tic-Tac-Toe, Chess and many other board games can be modelled with an MDP. Even games with dice, such as Backgammon. To do so, we define as state space S the set of configurations of the board; as reward function $R(s) = 1$, if the configuration s is a win, -1 for a lose, 0 otherwise. Even though most games are deterministic, the presence of an opponent makes the transition function T of the MDP nondeterministic. What these games have in common, is that the player gets to see the complete state of the game, which is the current configuration of the board. Future states of the game and rewards only depend on the current situation, not on the whole play. In Chess, for example, we can determine whether a configuration is a win or loss just by looking for a checkmate, there is no need to ask the players

[1] In this chapter, a variable with a subscript refers to the value of the variable at the discrete time indicated.

how the game has carried out.

Proving that Markovian T and R exist is easy for board games, because the rules of the game define them. As we will see in Section 2.2, when T is unknown, as always happens in the real-world, it's much more difficult to prove that we're in fact facing a MDP.

The *policy* is the criterion the agent uses to select the actions to perform. If the environment dynamics can be modelled with a MDP, the optimal action at time t only depends on s_t. So, there must exist an optimal policy as $\pi^* : S \to A$. Due to common estimation errors, it is always better to prefer nondeterministic policies, which return a probability distribution over the actions. The action at time t will be sampled according to $a_t \sim \pi(s_t)$. This dependency is represented by the dotted arrows of Figure 2.2.

We will now introduce few basic quantities of RL that serve to define what it means for an action or a policy to be optimal. The *discounted return* G is the combination of all rewards collected:

$$G \coloneqq r_0 + \gamma\, r_1 + \gamma^2 r_2 + \dots = \sum_{t=0}^{T} \gamma^t r_t \tag{2.1}$$

The discount factor, $0 \leq \gamma \leq 1$, decides the relative importance of immediate and future rewards. Usually, this factor is strictly less than 1 because this stimulates the agent to achieve rewards as soon as possible. It also produces a finite discounted reward, even for an infinite run $T \to \infty$. Since our environments are video games, each play is an episode and the total number of steps in each episode is finite.

It is now clear, that the optimal policy should always maximize the expected discounted return. The *value function* of a policy π computes this quantity for each state s:

$$v_\pi(s) \coloneqq \mathbb{E}_\pi[G \mid s_0 = s] \tag{2.2}$$

which computes the expected value of G, when the agent starts from state s and it follows the policy π. The notation $\mathbb{E}_\pi$ indicates that the estimation assumes that the actions are sampled according to π. Finally, we can define the *optimal policy* π^* as the one maximizing the value function at all states:

$$\pi^* : \quad v_{\pi^*}(s) \geq v_\pi(s) \qquad \forall s \in S, \quad \text{for all } \pi \tag{2.3}$$

The typical Reinforcement Learning problem is to find the optimal policy for an MDP with unknown T and R.

The *action-value function* of a policy π is a similar measure to the value function:

$$q_\pi(s,a) \coloneqq \mathbb{E}_\pi[G \mid s_0 = s, a_0 = a] \tag{2.4}$$

that forces the first action to be a. Since the agent only observes outcomes of single actions, this is usually a much more convenient form for updating the estimate of the expected discounted return. Also, the optimal policy can be simply expressed as:

$$\pi^*(s) = \arg\max_{a \in A} q_{\pi^*}(s, a) \tag{2.5}$$

So, instead of learning the optimal policy directly, we can learn the optimal state-value function, q_{π^*} (also denoted with q^*). Fortunately, we don't need π^* to valuate q^* because, assuming optimality, we know it satisfies the Bellman optimality equation:

$$q^*(s, a) = \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} q^*(s_{t+1}, a') \mid s_t = s, a_t = a\right] \tag{2.6}$$

$$= \sum_{s', r'} p(s', r' \mid s, a) \left(r' + \gamma \max_{a'} q^*(s', a')\right) \tag{2.7}$$

for any t.

Many learning algorithms exist for estimating q^*. Briefly, on-policy algorithms, estimate q_π of the policy currently in use $\pi \to \pi^*$; off-policy algorithms, instead, act according to any exploration policy π' and directly estimate q^*. Two famous algorithms in these classes are SARSA and Q-learning, respectively. The one used in this thesis is derived from the latter.

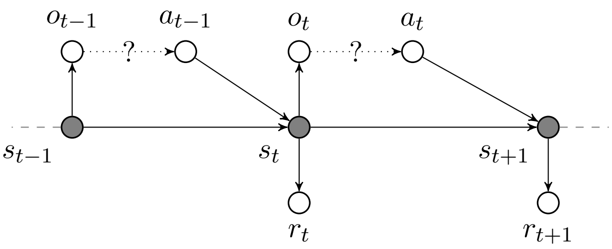

Figure 2.3. The Directed Graphical Model of a POMDP. Gray nodes are unobservable. For simplicity, the rewards in this graph depend just on the current state s_t, not on transitions (s_t, a_t, s_{t+1}).

2.2 Non-Markovian goals

The goal of a RL agent is to maximize the rewards received. A goal, or a task, is said *non-Markovian* if the rewards do not satisfy the Markov assumption on rewards, i.e:

$$r_{t+1} \not\perp s_i, a_i, r_i \quad 0 \leq i < t \qquad \text{for some } t \tag{2.8}$$

Of course, this can happen only if the environment cannot be modelled with an MDP. Excellent algorithms exists for MDPs; instead, non-Markovian goals are much more difficult to learn. There are two main causes for non-Markovian rewards: partial observations and temporally-extended tasks. We'll thoroughly analyze both scenarios.

2.2.1 Partial observations

Up to this point, we didn't need to distinguish between observations and states. In fact, we assumed that the agent can directly observe the environment states and act accordingly (we defined the policy as a function of the state). Unfortunately, this is often not the case: we only get to see something that depends on the current state, but it's not. These systems can be modelled with a *Partially Observable Markov Decision Process* (POMDP). POMDPs are a generalization of MDPs for partial observations. From now on, we will denote with S the environment state space and with Ω the observation space. Formally, a discrete-time POMDP is a 7-tuple $\langle S, A, T, R, \Omega, O, \gamma \rangle$, where S, A, T, R are defined as usual, Ω is the observation space, and O is the observation function $O : S \to \Omega$.

The graphical model of a POMDP is shown in Figure 2.3. The sequence of states $\langle s_0, s_1, \dots \rangle$, which is the environment dynamics, still satisfies the Markov assumption (it forms a Markov chain). In a POMDP, this dynamics exists but is unobservable. What we can see, instead, is a sequence of observations $\langle o_0, o_1, \dots \rangle$. Each of them is generated from the corresponding state, through the (possibly nondeterministic) observation function. Actions and policies can only act in response to observations, not states.

The dotted arrows in Figure 2.3 have a question mark on them, because that dependency is our choice. As designers, we're free to select the informations that the agent should take into account when selecting an action. Is the last observation enough to decide? Or, more precisely, among all possible policies, do non-Markovian goals always admit an optimal policy of the form $\pi^* : \Omega \to A$? Unfortunately, the answer is no. As we will see, other informations are needed.

If the transition and observation functions are known, a common solution is to estimate the states and decide the action from this belief. With deterministic functions, the agent can iteratively restrict the set of possible states by eliminating those inconsistent with the observations received. More commonly, these functions are nondeterministic. In this case, a probabilistic methods can be effective estimation algorithms. The iterative probabilistic filter applied to the sequence of observations would produce the belief distribution on the current state. We can represent the general procedure, at any instant t, with the following computation:

$$
\begin{tikzcd}
\langle o_0, o_1, \ldots, o_t \rangle \arrow[dr] & & \\
& b(s_t) \arrow[r] & a_t \\
\langle a_0, a_1, \ldots, a_{t-1} \rangle \arrow[ur] & &
\end{tikzcd}
$$

where $b(s)$ denotes the belief of s, being either a set of states or a probability distribution. Since each state estimate depends on the whole sequence of observations, also the next action is implicitly based on the whole history.

Standard RL algorithms cannot be applied to POMDPs, because the state space is not observable. Also, since we commonly assume the transition and observation functions to be unknown, no estimation could be carried out anyway. There is a clear difference between MDPs and POMDPs. Still, RL algorithms are frequently applied to POMDPs. Not surprisingly, they perform very poorly on these environments. See, for example, the games with worst performances in [16]. This is a subtle mistake, because determining whether we're observing the state space is the same as answering the following question: does the observation space capture the whole dynamics of the system being observed? Or, more precisely, does an equivalent MDP $\langle \Omega, A, T_\Omega, R_\Omega, \gamma' \rangle$ that produces the same rewards exist? If both $T_\Omega : \Omega \times A \times \Omega \to \mathbb{R}$ and $R_\Omega : \Omega \times A \times \Omega \to \mathbb{R}$ exist and produce the same rewards, the environment can be successfully modelled and solved with an MDP. Figure 2.4 represents this situation.

Example 2. As we've seen from Example 1 on page 8, the game of Chess can be modelled with an MDP if we consider as states the vectors of positions of all pieces

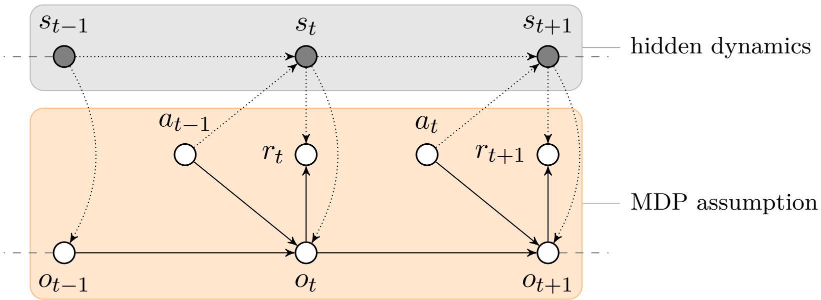

Figure 2.4. The dotted arrows ⇢ represent the dependencies in a POMDP model. Solid arrows ⟶ show the MDP model over the same quantities.

on the board. Let's suppose, instead, the observations available are images of the board after each move (if the pieces can be distinguished, these could even come from a real play). Each image completely captures the state of the game because, for each move of the agent and the opponent, we're able to accurately predict image that will follow. This is a transition T_Ω over images. Similarly, a reward function R_Ω can simply return $+1$ or -1 for images with checkmates and 0 otherwise. These functions can be unknown and don't need to be defined.

Suppose, instead, that the agent can only observe the left-hand side of the board (columns a-d, for example). In this case, each image provides an incomplete view over the state of the game. In fact, in order to determine the best action we must consider whether there are some attacking pieces on the hidden region. In this case, classic RL algorithms would perform poorly, because an image it's not sufficient to predict the next image and reward.

Example 3. Let's consider a classic control problem: the swing-up of an inverted pendulum. A pendulum can freely rotate by 360° around a hinge. The agent, at each discrete time step, can apply torques to this active joint. The goal is to stabilize the pendulum in the upward position, which is the configuration of unstable equilibrium. In order to solve this problem with Reinforcement Learning, we need to define the spaces S and A of the MDP. In this domain, actions are continuous torques. So, assuming a normalized range, $A \coloneqq [-1, +1] \subseteq \mathbb{R}$. The angle of the pendulum θ with respect to some fixed reference completely determines the position of the masses. Is the reward Markovian with respect to $S = \{\theta \in [-\pi, +\pi]\}$? No, because the agent is rewarded when the pendulum stops in the upward position. So, the appropriate state space consists of both θ and $\dot{\theta}$ (or, rather its discrete-time approximation).

Including the momentum is very common for mechanical systems. However, this can be also necessary for games. In fact, just looking at a single frame, the agent has no clue about how all the elements in the picture are moving. For example, an optimal policy would take into account the direction of an approaching ball in order to hit it.

2.2.2 Temporally-extended goals

The previous section has shown how partial observations may falsify the Markov assumption on rewards. A second possibility is to have a complete observation of the state ($\Omega = S$) but a task that is intrinsically non-Markovian. In this case, each reward is computed from the whole history of past events:

$$r_t = R(\langle s_0, a_0, \ldots, s_{t-1}, a_{t-1} \rangle) \qquad \forall t \in \mathbb{Z} \tag{2.9}$$

with $R : S^* \to \mathbb{R}$.

2.3 Deep Reinforcement Learning

Classic RL algorithms, such as SARSA and Q-learning, are tabular methods because they store and update the estimate for each pair (s, a) independently. Unfortunately, this requires discrete and small states and actions spaces. To overcome this very limiting assumption, we need parametrized value functions and policies. *Deep Reinforcement Learning* (Deep RL) is a recent field of RL in which Neural Networks (NN) are used as powerful function approximators for policies or value functions.

The main advantage of NN, and parametric models in general, is that they can be trained in high-dimensional and continuous input spaces. In fact, a good fit does not require a complete exploration of the input space, which may be unfeasible or impossible. Instead, they are trained with some form of Stochastic Gradient Descent (SGD) on the set of parameters from input-output samples. Then, the model will be able to generalize to inputs that have been never observed.

Deep RL algorithms allow very little guarantees about convergence and optimality. Even if the input space would be explored completely, due to parametrization, updates for recent samples also affect the regions previously visited. In fact, any Deep RL algorithm introduces some techniques in order to generate a stable training.

The Deep Q-Network (DQN) [17] is an important algorithm because it successfully combined previous techniques for a stable training, and an interesting model based on deep NN for the state-value function. The outcome is very satisfactory: the same neto This resulted in an They demonstrated that the same network can be trained in different games and achieve These promising results sparked a lively interest in Deep RL.

2.4 Temporal logics and Linear Dynamic Logic

2.4.1 Temporal logics on finite traces

Temporal logics are a class of formal languages, more precisely modal logics, that allow to talk about time [10]. Among all formalisms, we care about logics that assume a linear time, as opposed to branching, and a discrete sequence of instants, instead of continuous time. In computer science, the most famous logic in this group is the Pnueli's Linear Temporal Logic (LTL) [19].

The assumptions about the nature of time directly reflect to the type of structures these logics are interpreted on: their models are $\mathcal{M} = \langle T, \prec, V \rangle$, where the set T is a discrete set of time instants, such as $\mathbb{N}$, and $\prec$ is a complete ordering relation on T, like $<$. If a logic defines a set $\mathcal{F}$ of atomic propositions, the evaluation function $V : T \times \mathcal{F} \rightarrow \{true, false\}$, for each instant of time, assigns a truth value to each fluent. An equivalent and compact way of defining such structures is with *traces*. A trace π is a sequence of propositional interpretations $2^{\mathcal{F}}$ of the fluents $\mathcal{F}$. Each element of the trace, $\pi(i)$, is the set of true symbols at time i. $\pi(i, j)$ represents the trace between instants i and j.

LTL is a logic that only allows to talk about the future. The semantic of its temporal operators, neXt $\bigcirc$, Until $\mathcal{U}$, and of those derived, eventually $\Diamond$, always $\Box$, can only access future instants on the sequence. Interpretations for this logic are infinite traces with a first instant, which are equivalent to valuations on the temporal frame $\langle \mathbb{N}, < \rangle$.

As it has been pointed out [4], most practical uses of LTL interpret the formulae on *finite* traces, not infinite. The pure existence of a last instant of time has strong consequences on the meaning of the operators, because they need to handle such instant differently. The Always operator $\Box$, translates to "until the last instant", quite naturally. However, writing $\Box\Diamond\varphi$ does not require that φ becomes true an infinite number of times, that is the "response" property; instead, it is satisfied exactly by those traces in which φ is true at *Last* (*Last* is an abbreviation for $\neg\bigcirc true$ and it evaluates to true at last instant only). Furthermore $\Box\Diamond\varphi$ and $\Diamond\Box\varphi$ are both equivalent to $\Diamond(Last \land \varphi)$, something that doesn't happen in standard LTL. From last example, it should be clear that the expressive power of the language has changed and LTL interpreted over finite traces should be regarded as a different logic, that we denote with LTL_f. More precisely, over infinite linearly-ordered interpretations, LTL has the same expressive power of Monadic Second Order Logic (MSO), while LTL_f is equivalent to First-Order Logic (FOL) and star-free regular expressions, which are strictly less expressive.

In the next section, we will define a temporal logic, called LDL_f, that is purposefully devised for finite traces. This is the formalism that we use in the implemented construction for RL agents. However, many plans and behaviours to be rewarded can be also expressed with LTL_f. So, for this construction, any temporal logic over finite traces which can be translated to equivalent finite-state automata can be used as an alternative to LDL_f; even temporal logics of the past [1].

2.4.2 Linear Dynamic Logic

In this section, we will define Linear Dynamic Logic of finite traces (LDL_f) [4]. Its syntax combines regular expressions and propositional logic, just like Propositional Dynamic Logic (PDL) does [7][24]. So, we will review regular expressions first.

Regular Temporal Specifications

Regular languages are the class of languages exactly recognized by finite state automata and regular expressions [13]. So, we will use regular expressions as a compact formalism to specify them. Regular expressions are usually said to accept strings. Traces are in fact strings, whose symbols $s \in 2^{\mathcal{F}}$ are propositional interpretations of the fluents $\mathcal{F}$. Such regular expressions would be:

$$\rho ::= \emptyset \mid s \mid \rho_1 + \rho_2 \mid \rho_1; \rho_2 \mid \rho^* \tag{2.10}$$

where $\emptyset$ denotes the empty language, $s \in 2^{\mathcal{F}}$ is a symbol, $+$ is the disjunction of two constraints, ; separates concatenated expressions, and ρ^* requires an arbitrary repetition on ρ. Parentheses can be used to group expressions with any precedence.

We call the regular expressions of equation (2.10) Regular Temporal Specifications RE_f, because they are interpreted on finite linear temporal structures. However, writing specifications in terms of single interpretations is very cumbersome. So, we substitute the symbols $s \in 2^{\mathcal{F}}$ with formulae of Propositional Logic. A propositional formula ϕ represents all interpretations that satisfy it: $\text{Sat}(\phi) = \{s \in 2^{\mathcal{F}} \mid s \models \phi\}$.

The new definition for the syntax of Regular Temporal Specifications RE_f:

$$\rho ::= \phi \mid \rho_1 + \rho_2 \mid \rho_1; \rho_2 \mid \rho^* \tag{2.11}$$

where ϕ is a propositional formula on the set of atomic symbols $\mathcal{F}$. The language generated by a RE_f ρ, denoted $\mathcal{L}(\rho)$, is the set of traces that match the temporal specification. The only difference with regular expressions' standard semantics is that a symbol $s \in 2^{\mathcal{F}}$ matches a propositional formula ϕ if and only if $s \in \text{Sat}(\phi)$. A trace that match the regular expression $\pi \in \mathcal{L}(\rho)$ is said to be generated or accepted by the specification ρ.

Example 4. As an example, let's define a RE_f expression $\rho \coloneqq \mathit{true}; (\neg B)^*; (A \wedge B)$

and the following traces:

$$\begin{aligned}
\pi_1 &\coloneqq \langle \{\}; \{A\}; \{A\}; \{A, B\} \rangle \\
\pi_2 &\coloneqq \langle \{B\}; \{A, B\} \rangle \\
\pi_3 &\coloneqq \langle \{A, B\}; \{B\}; \{B\} \rangle
\end{aligned}$$

The first two traces are accepted by the expression, $\pi_1, \pi_2 \in \mathcal{L}(\rho)$, but the third is not, $\pi_3 \notin \mathcal{L}(\rho)$. Of course, the symbols A and B could represent any meaningful property of the environment to be ensured.

Linear Dynamic Logic

Linear Dynamic Logic is a temporal logic for finite traces that was first defined in [4]. The definition we see here, also adopted by the implementation we'll use, is a small variant that can also be interpreted over the empty trace, $\pi_\epsilon = \langle\rangle$, unlike most logics that assume a non-empty temporal domain T.

Definition 1. A LDL_f formula φ is built as follows:

$$\begin{aligned}
\varphi \quad &::= \quad tt \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle\rho\rangle\varphi \\
\rho \quad &::= \quad \phi \mid \varphi? \mid \rho_1 + \rho_2 \mid \rho_1; \rho_2 \mid \rho^*
\end{aligned} \tag{2.12}$$

where tt is a constant that stands for logical true and ϕ is a propositional formula over a set of symbols $\mathcal{F}$.

The syntax just defined is really similar to PDL [7], a well known and successful formalism in Computer Science for describing states and events of programs. However, LDL_f formulae are interpreted over finite traces instead of Labelled Transition Systems.

Before moving to the semantics, we can intuitively understand the meaning of the constructs. The second line of definition (2.12) is a Regular Temporal Specification RE_f, with the addition of the test operator ?, typical of PDL. In $\langle\rho\rangle\varphi$, the RE_f expression ρ is used as a modal operator to move to future states: it states that there exists at least one

2.5 Reinforcement learning with restraining specifications

Restraining Bolt method [5][9].

Chapter 3

Learning to valuate fluents in games

The importance of correctly valuate the fluents.

3.1 Temporal constraints

How we can use temporal logic to express legal traces of interpretations; e.g. expected behaviours.

3.2 Assumptions

A temporal constraints aren't definitions; they are just minimal constraints. We need additional clues: visual description of fluents. Now follow my assumptions:

- Local propertes (with regions I don't have to find elements in a frame).
- The property is visually apparent, inside the region.

Limitations and other ideas for a stronger grouding.

3.3 General structure of the model

Illustration and general description of the model.

3.4 Encoding

Encoder: the model, how it works, what does it learn, size of the encoding.

References: Training Restricted Boltzmann Machines and Deep Belief Neworks [23][18].

3.4.1 Model: Deep Belief Network

3.4.2 What does it learn

3.5 Boolean functions

The fluents are true in a set of those configurations.

3.5.1 Learning with genetic algorithms

Ideas from concept learning; genetic algorithm.

References: Genetic Algorithms for Concept learning[14], Genetic Algorithms review[11].

3.5.2 Boolean rules

Representation of boolean functions and training details.

Chapter 4

AtariEyes package

Intro to the software. What we can do:

- Train a Reinforcement Learning agent.
- Train the features extraction.
- Run a Restraining Bolt while training and playing an agent.

4.1 How to use the software

4.1.1 Tools and setup

4.1.2 Commands

Small user reference.

4.2 Implementation

4.2.1 `agent` Module

`training` Module

`playing` Module

4.2.2 `streaming` Module

4.2.3 `features` Module

`models` Module

`genetic` Module

`temporal` Module

Chapter 5

Experiments

5.1 Breakout

5.1.1 Definitions

5.1.2 Training

5.1.3 Comments

5.2 Montezuma's Revenge

5.2.1 Definitions

5.2.2 Training

5.2.3 Comments

5.3 New example

I'll try to train one additional environment just to show a case in which the role of the temporal constraints is more evident, like in Breakout (something like a simple game with many rules).

Chapter 6

Conclusions and future work

What I have done (concretely); what I haven't done; how I'd improve the results and how to possibly relax some assumptions.

Bibliography

[1] Fahiem Bacchus, Craig Boutilier, and Adam Grove. "Rewarding Behaviors". In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*. AAAI'96. Portland, Oregon: AAAI Press, 1996, pp. 1160–1167. ISBN: 026251091X.

[2] Marc G. Bellemare et al. "The arcade learning environment: An evaluation platform for general agents". In: *IJCAI International Joint Conference on Artificial Intelligence* 2015-January (2015), pp. 4148–4152. ISSN: 10450823. DOI: `10.1613/jair.3912`. arXiv: `1207.4708`.

[3] Ronen I. Brafman, Giuseppe De Giacomo, and Fabio Patrizi. "LTLf / LDLf non-markovian rewards". In: *32nd AAAI Conference on Artificial Intelligence, AAAI 2018* (2018), pp. 1771–1778.

[4] Giuseppe De Giacomo and Moshe Y. Vardi. "Linear temporal logic and Linear Dynamic Logic on finite traces". In: *IJCAI International Joint Conference on Artificial Intelligence* (2013), pp. 854–860. ISSN: 10450823.

[5] Giuseppe De Giacomo et al. "Foundations for restraining bolts: Reinforcement learning with LTLf/LDLf restraining specifications". In: *Proceedings International Conference on Automated Planning and Scheduling, ICAPS* Brooks 1991 (2019), pp. 128–136. ISSN: 23340843.

[6] Marco Favorito. "Reinforcement Learning for LTLf / LDLf Goals : Theory and Implementation". MA thesis. La Sapienza Università di Roma, 2018.

[7] Michael J. Fischer and Richard E. Ladner. "Propositional dynamic logic of regular programs". In: *Journal of Computer and System Sciences* 18.2 (1979), pp. 194–211. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(79)90046-1`.

[8] Vincent François-Lavet et al. "An Introduction to Deep Reinforcement Learning". In: *Foundations and Trends in Machine Learning* 11.3-4 (Nov. 2018), pp. 219–354. ISSN: 1935-8237. DOI: `10.1561/2200000071`. arXiv: `1811.12560`.

[9] Giuseppe De Giacomo et al. “Imitation Learning over Heterogeneous Agents with Restraining Bolts”. In: *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*. Ed. by J. Christopher Beck et al. AAAI Press, 2020, pp. 517–521.

[10] Valentin Goranko and Antje Rumberg. “Temporal Logic”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2020. Metaphysics Research Lab, Stanford University, 2020.

[11] Ahmad Hassanat et al. “Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach”. In: *Information* 10.12 (Dec. 2019), p. 390. ISSN: 2078-2489. DOI: `10.3390/info10120390`. URL: `https://www.mdpi.com/2078-2489/10/12/390`.

[12] Matteo Hessel et al. “Rainbow: Combining improvements in deep reinforcement learning”. In: *32nd AAAI Conference on Artificial Intelligence, AAAI 2018* (2018), pp. 3215–3222. arXiv: `1710.02298`.

[13] John E. Hopcroft et al. *Introduction to Automata Theory, Languages and Computability*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201441241.

[14] Kenneth A. de Jong, William M. Spears, and Diana F. Gordon. “Using Genetic Algorithms for Concept Learning”. In: *Machine Learning* 13.2 (1993), pp. 161–188. ISSN: 15730565. DOI: `10.1023/A:1022617912649`.

[15] *Learning Montezuma’s Revenge from a Single Demonstration*. 2018. URL: `https://openai.com/blog/learning-montezumas-revenge-from-a-single-demonstration/`.

[16] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533. ISSN: 14764687. DOI: `10.1038/nature14236`. URL: `http://dx.doi.org/10.1038/nature14236`.

[17] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: (2013), pp. 1–9. arXiv: `1312.5602`. URL: `http://arxiv.org/abs/1312.5602`.

[18] K.P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning series. MIT Press, 2012. ISBN: 9780262018029.

[19] Amir Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)* (1977), pp. 46–57.

[20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597.

[21] Xudong Sun and Bernd Bischl. “Tutorial and Survey on Probabilistic Graphical Model and Variational Inference in Deep Reinforcement Learning”. In: *2019 IEEE Symposium Series on Computational Intelligence, SSCI 2019*. December. 2019, pp. 110–119. arXiv: 1908.09381.

[22] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. 2018. ISBN: 9780262039246. URL: https://www.tandfonline.com/doi/full/10.1080/24699365.1977.11669658.

[23] Tijmen Tieleman. “Training restricted boltzmann machines using approximations to the likelihood gradient”. In: *Proceedings of the 25th International Conference on Machine Learning*. ICML ’08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 1064–1071. ISBN: 9781605582054. DOI: 10.1145/1390156.1390290.

[24] Nicolas Troquard and Philippe Balbiani. “Propositional Dynamic Logic”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2019. Metaphysics Research Lab, Stanford University, 2019.

[25] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with double Q-Learning”. In: *30th AAAI Conference on Artificial Intelligence, AAAI 2016* (2016), pp. 2094–2100. arXiv: 1509.06461.