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

Nomenclature

AFW	Alternating Automaton on finite Words, page 14
DFA	Deterministic Finite-state Automaton, page 15
DGM	Directed Graphical Model, page 18
DQN	Deep Q-Network algorithm, page 24
FOL	First-Order Logic, page 8
LDL_f	Linear Dynamic Logic of finite traces, page 11
LTL	Linear Temporal Logic, page 7
LTL_f	Linear Temporal Logic of finite traces, page 8
MDP	Markov Decision Process, page 17
MSO	Monadic Second-Order Logic, page 8
NFA	Nondeterministic Finite-state Automaton, page 14
NMRDP	Non-Markovian Reward Decision Process, page 30
NN	Neural Network, page 23
NNF	Negation Normal Form, page 15
PDL	Propositional Dynamic Logic, page 8
POMDP	Partially Observable Markov Decision Process, page 27
RL	Reinforcement Learning, page 17
SGD	Stochastic Gradient Descent algorithm, page 24

Chapter 1

Introduction

A classic and important branch of Artificial Intelligence (AI) aims at developing agents that select their actions through a form of logic reasoning, such as planning. One of the main advantages of these approaches is that reasoning proceeds by manipulating *abstractions*. In fact, in logic, we can define symbols that represent any meaningful event or condition that should be considered. For example, some propositional symbols might represent conditions such as “the door is closed” or “I am holding an object”, etc. We’ll also call these propositional symbols with the term “*fluents*” (a name that suggests how their truth can change over time).

These reasoning methods based on logics are powerful but they imply one fundamental ability: at each instant, the agent must be able to decide whether those propositions are true. This means that all symbols that represent conditions which happen to be true in the environment, must be true for the agent. This “grounding” process can be really hard in complex environments, because the agent’s sensors may return a noisy and multidimensional output, that is difficult to interpret.

Reinforcement Learning (RL) is a successful field of AI, in which the agent’s goal is to learn a policy that maximize the rewards received. We could argue that RL does not require the valuations just mentioned. Still, rewards and punishments must be somehow supplied in response to desirable and undesirable events. We could consider of providing these feedbacks with programmed ad-hoc conditions, but this can be easily done just for the simulations we create. Furthermore, as we will see, some complex tasks can only be solved through a combination of both RL and logic-based methods; thus, introducing all the needs of the latter.

With this thesis, we defined and implemented an agent based on temporal logics and Deep Reinforcement Learning. This required to investigate new ways to solve the fluents valuation problem that has been just described, for a specific class of environments and fluents. In Section 1.2, the goal and the achievements of this work will be described more precisely.

1.1 Related works

Reinforcement Learning (RL) is an area of Machine Learning in which the agent is trained by sending rewards and punishments in response to its actions. This technique can be also used unknown environments, where a model of the dynamics is not available, because the agent learns by trying all actions and by remembering those that lead to the highest rewards. As we will see in Chapter 3, most RL algorithms assume that the environment can be modelled with a Markov Decision Process (MDP). Many learning algorithms exist in this setting [22].

Neural Networks (NN) have brought new possibilities for RL: in Deep Reinforcement Learning (Deep RL), the agent employs a neural network as a very expressive function approximator for the quantities it is trying to learn [8]. For example, the optimal q-value is an important quantity in RL, that the agents are usually designed to learn from the observations received. The Deep Q-Network (DQN) algorithm [17] is one the first to successfully employ neural networks in RL. They have shown that a Deep RL agent can be trained directly from complex observations such as the frames of a video game. Without modifications, the same agent has been able to reach human-level performances in many games.

Games have always been a classic benchmark for AI algorithms, because they provide various levels of complexity, they have few and strict rules, and they are easy to implement and simulate. Regarding Deep RL, many authors have tested their algorithms on the collection of video games "Atari 2600" [2]. In this thesis, we'll use and experiment with the same environments.

The reinforcement learning algorithm we've adopted is called Double 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 were able to clearly achieve superior performances in most of these games combined a combination of all DQN variants [12].

If we look at the results in [16], DQN agents are able to learn excellent policies for many games. However, for many other environments of the same collection, the agents struggle to learn and, in some cases, they don't learn anything at all. The worst performances have been measured for the *Montezuma's Revenge* environment. Even in the works that followed, the only methods that were able to achieve good policies in this game adopted some form of expert imitation and manual restarts [15]. In Section 3.3, we'll investigate the main cause of these difficulties.

As we will see throughout this thesis, a promising solution for these environments is the construction provided in [3] and [5]. The former work [3] has shown that a Non-Markovian Reward Decision Process (NMRDP) can be easily declared with linear-time temporal logics, and it has provided a translation from this NMRDP to a classic MDP. The logics they used are LTL$_f$ and LDL$_f$. This idea was initially introduced in [1] for a linear temporal logic of the past. The latter work [5], instead, has shown that through same construction, it's possible to declare with temporal logics the rewards of a RL agent, thus influencing its final behaviour. This paper named this additional module with “Restraining Bolt”. Thorough this thesis, it might be handy to use this name to refer to this logic construction.

1.2 Objective and results

This work started with an initial goal: defining and implementing a Deep RL agent that, through the Restraining Bolt method, is able to achieve non-Markovian goals. As we will see, the environments adopted in Deep RL are much more complex than those of classic RL. They usually have very complex observation spaces, in which it's much harder to decide the truth of the logic propositions used in the Restraining Bolt. This general issue required to investigate new ways to solve the problem of fluents valuation, that we've opened our introduction with. So, while this thesis achieved the initial goal, also start a

The main purpose of this work is to devise and test a mechanism able to learn functions which valuate 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 Structure of the thesis

The rest of this thesis is structured as follows:

2 – Temporal logics and Linear Dynamic Logic

In this chapter, an important formalism that will be used throughout the thesis is reviewed. We introduce the reader to concepts such as fluents, traces and linear-time temporal logics. Then, we will define the Linear Dynamic Logic, which is the specific temporal logic used in this text.

3 – Deep Reinforcement Learning for non-Markovian goals

This chapter presents the second large background of this thesis. We will see Reinforcement Learning from the basic concepts and assumptions, in Section 3.1. Section 3.2 reviews some of the advancements of the Deep RL field of last years. Then, in Section 3.3, we will analyze what happens when the most common assumptions of RL (and of Deep RL) are falsified. We present the recent Restraining Bolt method in Section 3.4.2 and we apply it to a Deep RL agent, in Section 3.5, by proposing an original model.

4 – Learning to valuate fluents in games

Here, we'll see how the agent can be trained to valuate a class of fluents to their expected truth. A model for the valuation function will be proposed and a training algorithm.

5 – AtariEyes package

This chapter presents the software that implemented the concepts presented in the previous chapters. I will be first explained from a use perspective, then the most interesting implementation details will follow.

6 – Esperiments

This chapter contains experiments and training outcomes for two Atari games. Experiments will be finalized to test the effectiveness of learning the fluents valuation functions and the capabilities of the "restrained" Deep RL agents.

7 – Conclusions and future work

This thesis ends with some final considerations about: the main conclusions that can be derived from this work; the strength of this approach and its weakness; and its possibilities for improvement.

Chapter 2

Temporal logics and Linear Dynamic Logic

2.1 Temporal logics on finite traces

Temporal logics are a class of formal languages, more precisely modal logics, that allow to talk about properties and events over 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 tuples $\mathcal{M} = \langle T, \prec, V \rangle$, where T is a discrete set of time instants, such as $\mathbb{N}$, $\prec$ is a complete ordering relation on T, like $<$, and V is a valuation function $V : T \times \mathcal{F} \rightarrow \{\mathit{true}, \mathit{false}\}$. For a logic that defines a set $\mathcal{F}$ of proposition symbols, the function V assigns a truth value to each of them, in every instant of time. The symbols in $\mathcal{F}$ represent atomic propositions which may or may not hold in different time instants. They are also called "fluents" (or simply propositional symbols, in this thesis). An equivalent and compact way of defining such structures is with *traces*. A trace π is a sequence $\pi_0\pi_1 \dots \pi_n$, where each element is a propositional interpretation of the fluents $\mathcal{F}$. Each symbol π_i in the sequence is the set of true symbols at time i: $\pi_i \in 2^{\mathcal{F}}$. The i-th element is also denoted with $\pi(i)$. $\pi(i, j)$ represents the trace between instants i and j: $\pi_i, \pi_{i+1}, \dots, \pi_{j-1}$.

LTL is a logic that only allows to talk about the future. The semantics 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 in [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 all formulae, because operators semantics need to handle such instant differently. For example, the “always” operator $\Box$ translates to “until the last instant”, quite naturally. However, the formula $\Box\Diamond\varphi$ no longer requires that φ becomes true an infinite number of times (in LTL, this formula represents the “response” property); instead, it is satisfied exactly by those traces in which φ is true at the last instant. So, it assumes a completely different meaning. Furthermore, both $\Box\Diamond\varphi$ and $\Diamond\Box\varphi$ become equivalent to $\Diamond(\mathit{Last} \wedge \varphi)$: something that doesn't happen in standard LTL[1]. From this 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 will 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 than MSO.

In the next section, we will define a temporal logic, called LDL_f, that was purposefully devised to be interpreted over finite traces. This is the formalism that we will use, in Section 3.4.2, to declare plans and desired behaviours. However, many useful temporal properties can be also expressed with LTL_f. So, one may also use as alternative formalisms LTL_f or any temporal logic over finite traces that can be translated to equivalent finite-state automata; even temporal logics of the past [1].

In Section 2.3, we will define the 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.

[1] *Last* is an abbreviation for $\neg\bigcirc \mathit{true}$ and it valuates to true at last instant only. So, $\Diamond(\mathit{Last} \wedge \varphi)$ means: eventually, at the last instant, φ is true.

2.2 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:

$$\rho ::= \emptyset \mid s \mid \rho_1 + \rho_2 \mid \rho_1; \rho_2 \mid \rho^* \tag{2.1}$$

where $\emptyset$ denotes the empty language, $s \in 2^{\mathcal{F}}$ is a symbol, $+$ is the disjunction of two constraints, ; concatenates two expressions, and ρ^* requires an arbitrary repetition on ρ. Parentheses can be used to group expressions with any precedence. Regular expressions are a basic formalism in computer science and they won't be covered here. The notable difference, though, is that the symbols found in the trace, hence of the regular expression, are propositional interpretations (i.e. sets of true fluents).

Example 1. Briefly, the regular expression $\rho \coloneqq (\{A\}^* + \{B\}^*); \{\}$ accepts the following traces:

$$\begin{aligned}
\pi_a &\coloneqq \langle \{A\}, \{A\}, \{A\}, \{\} \rangle \\
\pi_b &\coloneqq \langle \{B\}, \{\} \rangle \\
\pi_c &\coloneqq \langle \{\} \rangle
\end{aligned}$$

but not $\pi_d \coloneqq \langle \{A, B\} \rangle$.

We call the regular expressions of equation (2.1) Regular Temporal Specifications RE$_f$, because they are interpreted on finite linear temporal structures. Unfortunately, writing specifications in terms of single interpretations can be very cumbersome, as we lack a construct for negation and all sets need to match exactly. Instead, we can substitute the symbols $s \in 2^{\mathcal{F}}$ with formulae of Propositional Logic. In fact, a propositional formula ϕ concisely represents all interpretations that satisfy it: $\mathrm{Sat}(\phi) \coloneqq \{s \in 2^{\mathcal{F}} \mid s \models \phi\}$.

The new definition for the syntax of Regular Temporal Specifications RE$_f$ is:

$$\rho ::= \phi \mid \rho_1 + \rho_2 \mid \rho_1; \rho_2 \mid \rho^* \tag{2.2}$$

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 $s \in \mathrm{Sat}(\phi)$. A trace that match the regular expression $\pi \in \mathcal{L}(\rho)$ is said to be generated or accepted by the specification ρ.

Example 2. As an example, let's define a RE$_f$ expression $\rho \coloneqq \mathit{true}; (\neg B)^*; (A \wedge B)$ and the following traces:

$$\begin{aligned}
\pi_a &\coloneqq \langle \{\}, \{A\}, \{A\}, \{A, B\} \rangle \\
\pi_b &\coloneqq \langle \{B\}, \{A, B\} \rangle \\
\pi_c &\coloneqq \langle \{A, B\}, \{B\}, \{B\} \rangle
\end{aligned}$$

The first two traces are accepted by the expression, $\pi_a, \pi_b \in \mathcal{L}(\rho)$, but the third is not, $\pi_c \notin \mathcal{L}(\rho)$. Of course, the symbols A and B may represent any meaningful property of the environment that we may want to ensure at some time instants.

2.3 Linear Dynamic Logic

2.3.1 Definition

We can now move on to the *Linear Dynamic Logic of finite traces* (LDL$_f$). This logic was first defined in [4]. The definition we see here, also adopted in this thesis, is a small variant that can also be interpreted over the empty trace, $\pi_\epsilon = \langle\rangle$, unlike most logics, which assume a non-empty temporal domain T. This definition appears in [3].

Definition 1. A LDL$_f$ formula φ is built as follows:

$$\begin{aligned} \varphi &::= && tt \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle\, \rho \,\rangle\varphi \\ \rho &::= && \phi \mid \varphi? \mid \rho_1 + \rho_2 \mid \rho_1; \rho_2 \mid \rho^* \end{aligned} \tag{2.3}$$

where tt is a constant that stands for logical true and ϕ is a propositional formula over a set of symbols $\mathcal{F}$. We also define the following abbreviations:

$$ff \coloneqq \neg tt \qquad [\,\rho\,]\varphi \coloneqq \neg\langle\,\rho\,\rangle\neg\varphi \qquad \phi \coloneqq \langle\,\phi\,\rangle tt$$
$$End \coloneqq [\,true\,]ff \qquad Last \coloneqq \langle\,true\,\rangle End$$

together with all those of propositional logic, which are all to be considered part of the language.

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.

Example 3. All the following formulae are all well-formed:

$$A \vee \neg B$$
$$\langle\, A; B^* \,\rangle(A \wedge B)$$
$$[\,true^*\,]\neg C$$
$$[\,A^*\,]\langle\,\neg B\,\rangle tt \wedge [true^*; C]ff$$
$$[\,A?; B\,]B$$

Instead, these are not:

$$\langle\, tt\,\rangle A \qquad \langle\, A\,\rangle \qquad [\,A\,]B\,[\,A\,]B \qquad B?$$

Before moving to the semantics, we can intuitively understand the meaning of these constructs. A LDL$_f$ formula φ is a combination of temporal expressions,

$\langle\,\rho\,\rangle, [\,\rho\,]$, and propositional formulae. The former are modal expressions that allow to make statements that refer to future instants. $\langle\,\rho\,\rangle\varphi$ states that, from the current step i, there exists a future instant j, such that the path $\pi(i, j)$ is accepted by the RE$_f$ ρ, and φ is satisfied at step j. Essentially, as in PDL, regular expressions are used to select some future states in which the formulae that follow should hold. Similarly, $[\,\rho\,]\varphi$ states that, from the current step, all executions satisfying ρ are such that their last instant satisfy φ. There is a clear similarity between $\langle\ \rangle, [\ ]$ operators and $\exists, \forall$ from first-order logic, because we defined them to obey a similar relation to the De Morgan rule. In fact, if we consider the set S_ρ of future instants that are selected by a regular expression ρ, $\langle\,\rho\,\rangle$ can be read as "*there exists* one instant in S_ρ such that ...", and $[\,\rho\,]$ is read as "*for all* instants in S_ρ ...".

The LDL$_f$ semantics is defined in terms of finite traces. We denote with $|\pi|$ the length of the trace π, i.e. the total number of time instants. Also, for non-empty traces, *last* refers to the index of the last instant in the sequence: $\mathit{last} := |\pi| - 1$.

Definition 2. Given a finite trace π, we inductively define when a LDL$_f$ formula φ is true in π at time i, in symbols $\pi, i \models \varphi$, as follows:

$$\begin{aligned}
&\pi, i \models \mathit{tt}\\
&\pi, i \models \neg\varphi \quad \text{iff} \quad \pi, i \not\models \varphi\\
&\pi, i \models \varphi_1 \wedge \varphi_2 \quad \text{iff} \quad \pi, i \models \varphi_1 \quad \text{and} \quad \pi, i \models \varphi_2\\
&\pi, i \models \langle\,\phi\,\rangle\varphi \quad \text{iff} \quad i < |\pi| \quad \text{and} \quad \pi(i) \models \phi \quad \text{and} \quad \pi, i+1 \models \varphi\\
&\qquad\qquad \text{(for a propositional formula } \phi)\\
&\pi, i \models \langle\,\rho_1 + \rho_2\,\rangle\varphi \quad \text{iff} \quad \pi, i \models \langle\,\rho_1\,\rangle\varphi \vee \langle\,\rho_2\,\rangle\varphi\\
&\pi, i \models \langle\,\rho_1; \rho_2\,\rangle\varphi \quad \text{iff} \quad \pi, i \models \langle\,\rho_1\,\rangle\langle\,\rho_2\,\rangle\varphi\\
&\pi, i \models \langle\,\psi?\,\rangle\varphi \quad \text{iff} \quad \pi, i \models \psi \quad \text{and} \quad \pi, i \models \varphi\\
&\pi, i \models \langle\,\rho^*\,\rangle\varphi \quad \text{iff} \quad \pi, i \models \varphi \quad \text{or}\\
&\qquad\qquad i < |\pi| \quad \text{and} \quad \pi, i \models \langle\,\rho\,\rangle\langle\,\rho^*\,\rangle\varphi \quad \text{and} \quad \rho \text{ is not test-only}
\end{aligned}$$

We say that ρ is test-only if it is a RE$_f$ whose atoms are only tests $\psi?$.

NOTE: The only doubly-inductive semantics for the empty-trace version of LDL$_f$ that I could find was in Marco Favorito's thesis at page 15. With respect to that, I've just corrected $i < \mathit{last}$ with $i < |\pi|$.

Definition 3. A LDL$_f$ formula φ *is true* in (or, is satisfied by) a trace π, written $\pi \models \varphi$, if $\pi, 0 \models \varphi$.

Definition 4. A LDL$_f$ formula φ is *satisfiable* if there exists a trace that satisfy it; it is *valid* if it is true in every trace. A formula φ logically implies a formula φ', in notation $\varphi \models \varphi'$, if for every trace π, we have that $\pi \models \varphi$ implies $\pi \models \varphi'$.

Example 4. We'll now list few examples that may help to better understand the semantics just defined. Given the following trace,

$$\pi_a \coloneqq \langle \{A\}, \{A\}, \{A, B\} \rangle$$

we have[2]:

$$\begin{aligned} \pi_a, 2 &\models A \wedge B & \qquad \pi_a, 1 &\not\models B \\ \pi_a &\models \langle\, A^* \,\rangle (A \wedge B) & \pi_a &\not\models [\, A^* \,] B \\ \pi_a &\models [\, (\neg B)^* \,] A & \pi_a &\not\models \langle\, A; B \,\rangle tt \end{aligned}$$

Now that the fundamental mechanics are clear, we can highlight some peculiarities of the language:

- The test operator "?" is typical of PDL. In the middle of a RE$_f$ computation, I can check whether a condition is verified before moving on. As we can see from Definition 1, the condition is a full LDL$_f$ formula; so it acts as a powerful look-ahead operation.

- The two expressions *true* and *tt* may mistakenly look equivalent at first sight. Instead, *tt* is an atomic formula that is *always* satisfied (it valuates to logical true); while *true* is a propositional formula and, as such, it is an abbreviation for $\langle\, \mathit{true} \,\rangle tt$. The latter is satisfied if and only if there exists at least one next instant in the trace.

- According to the semantics of $\langle\, \phi \,\rangle \varphi$, if we valuate this formula on the last instant of a trace, φ needs to be verified at step $\mathit{last} + 1$. This is fine, indeed. As the truth of $\pi, i \models \varphi$, with $i \geq |\pi|$, is perfectly defined in LDL$_f$ (and it's equivalent to $\langle\rangle \models \varphi$).

The last observation has some profound consequences that we should consider when writing the formulae. Let's suppose we want to encode that A must always hold, just like the LTL$_f$ sentence "always A", $\Box A$. The LDL$_f$ formula $[\, \mathit{true}^* \,] A$ doesn't represent this concept; instead, it is unsatisfiable. What we're actually saying is that at each point of the trace, even at the end, A must follow; which is impossible. What we meant is $[\, \mathit{true}^* \,](A \vee \mathit{End})$, or $[\, \mathit{true}^*; (\neg \mathit{End})? \,] A$.

In [4] (De Giacomo and Vardi) some important theorems have been established for LDL$_f$[3]:

[2] We shouldn't get confused about the different uses of the angle brackets: in $\langle \{A\}, \{A, B\} \rangle$, they delimit a sequence of sets (that is a trace); in the formula $\langle\, A^*; \mathit{true} \,\rangle B$, instead, they represent the temporal operator containing a regular temporal specification.

[3] The following theorems have been proved in [4] for the original definition of LDL$_f$, which is slightly different. The same results are valid for the empty trace variant presented here.

Theorem 1. *LTL_f and RE_f formulae can be translated to LDL_f in linear time.*

Theorem 2. *LDL_f has the same expressive power of MSO (Monadic Second-Order logic) over finite traces.*

Theorem 3. *Satisfiability, validity and logical implication for LDL_f formulae are PSPACE-complete.*

2.3.2 Automaton translation

Automata are a well-established common formalism for talking about languages. The set of traces that satisfy a formula form a language, in fact. What we're looking for is an equivalent automaton $\mathcal{A}_\varphi$ that accepts exactly the same traces that satisfy a given LDL$_f$ formula φ. As we will see, this translation is indeed possible, because to every LDL$_f$ formula, it can be associated an equivalent alternating automaton on finite words. We assume the reader is acquainted with basic automata theory, such as DFAs, NFAs [13].

Alternating automaton

Definition 5. An *Alternating Automaton on finite Words* (AFW) is a tuple $\mathcal{A} \coloneqq \langle \Sigma, Q, q_0, \delta, F \rangle$, where Σ is a finite nonempty alphabet, Q is a finite nonempty set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is the transition function. $\mathcal{B}^+(Q)$ denotes the set of all positive boolean functions composed from the set of atoms Q, together with *true* and *false*.

Let's define a word $w \coloneqq \sigma w'$, with $\sigma \in \Sigma$ and $w' \in \Sigma^*$. In a *Nondeterministic Finite-state Automaton* (NFA), a transition $\delta(q_0, \sigma) = \{q_1, q_2, q_3\}$ means that input word, w, is accepted iff w' is accepted by *any* of the runs continuing from q_1, q_2, or q_3 (a run accepts if it ends up in a final state). We could also write this transition as $\delta(q_0, \sigma) = (q_1 \lor q_2 \lor q_3)$, because the agent can "choose" among these states. The dual operation is to require that *all* the following runs needs to be accepting: $\delta(q_0, \sigma) = (q_1 \land q_2 \land q_3)$. The transition function of an AFW adopts boolean functions without negations to allow both conjunctions and disjunctions. For example, we may write $\delta(q_0, \sigma) = q_1 \land (q_2 \lor q_3)$. Due to conjunctions, the AFW is usually thought to be in multiple states at the same time. An AFW, $\mathcal{A}$, accepts a word w iff there exists a run of $\mathcal{A}$ on w such that all the current states at the end of the computation are either final or are labelled with *true*[4]. For a more precise description of runs and acceptance condition of AFWs, see [26].

[4] $\delta(q, \sigma) = \mathit{true}$ can be considered as an empty conjunction of states: this branch of the computation accepts without the need of further tests.

Surprisingly, alternating automata have the same expressive power as NFA, but they are exponentially more succinct. So, it is possible to transform any AFW to an equivalent NFA that has an exponentially bigger state space. Therefore, we can also build an equivalent *Deterministic Finite-state Automaton* (DFA) through a double exponential transformation. The AFW-to-NFA transformation is provided in [26].

Delta function

In this section, given a LDL$_f$ formula φ, we define its associated AFW, $\mathcal{A}_\varphi := \langle \Sigma, Q, q_0, \delta, F \rangle$. The alphabet, $\Sigma := 2^{\mathcal{F}}$, is the set of all propositional interpretations for the fluents $\mathcal{F}$ (we will indicate them with $\Pi \in \Sigma$). It will be handy to use LDL$_f$ formulae to refer to the states $q \in Q$: each formula will be an (implicitly quoted) identifier for a state. Formally, the set Q is the Fisher-Ladner closure of φ extended with two special constructs, but we don't need to define it explicitly. The initial state is $q_0 := \varphi$. The set of final states is empty, $F := \{\}$, so acceptance can only be ensured by reaching *true*. We now assume that the formula φ has been already transformed in Negation Normal From (NNF), trough a function $\mathit{nnf}(\cdot)$. A formula is in NNF if negations only appear in front of atomic propositions. We can finally define the transition function, also called the *delta function*, as:

$$
\begin{aligned}
\delta(tt, \Pi) &= \mathit{true} \\
\delta(\mathit{ff}, \Pi) &= \mathit{false} \\
\delta(\phi, \Pi) &= \delta(\langle\, \phi \,\rangle tt, \Pi) \qquad (\phi \text{ propositional}) \\
\delta(\varphi_1 \wedge \varphi_2, \Pi) &= \delta(\varphi_1, \Pi) \wedge \delta(\varphi_2, \Pi) \\
\delta(\varphi_1 \vee \varphi_2, \Pi) &= \delta(\varphi_1, \Pi) \vee \delta(\varphi_2, \Pi) \\
\delta(\langle\, \phi \,\rangle \varphi, \Pi) &= \begin{cases} \mathbf{E}(\varphi) & \text{if } \Pi \models \phi \\ \mathit{false} & \text{if } \Pi \not\models \phi \end{cases} \qquad (\phi \text{ propositional}) \\
\delta(\langle\, \psi? \,\rangle \varphi, \Pi) &= \delta(\psi, \Pi) \wedge \delta(\varphi, \Pi) \\
\delta(\langle\, \rho_1 + \rho_2 \,\rangle \varphi, \Pi) &= \delta(\langle\, \rho_1 \,\rangle \varphi, \Pi) \vee \delta(\langle\, \rho_2 \,\rangle \varphi, \Pi) \\
\delta(\langle\, \rho_1; \rho_2 \,\rangle \varphi, \Pi) &= \delta(\langle\, \rho_1 \,\rangle \langle\, \rho_2 \,\rangle \varphi, \Pi) \\
\delta(\langle\, \rho^* \,\rangle \varphi, \Pi) &= \delta(\varphi, \Pi) \vee \delta(\langle\, \rho \,\rangle \mathtt{F}_{\langle\, \rho^* \,\rangle \varphi}, \Pi) \\
\delta([\, \phi \,] \varphi, \Pi) &= \begin{cases} \mathbf{E}(\varphi) & \text{if } \Pi \models \phi \\ \mathit{true} & \text{if } \Pi \not\models \phi \end{cases} \qquad (\phi \text{ propositional}) \\
\delta([\, \psi? \,] \varphi, \Pi) &= \delta(\mathit{nnf}(\neg \psi), \Pi) \vee \delta(\varphi, \Pi) \\
\delta([\, \rho_1 + \rho_2 \,] \varphi, \Pi) &= \delta([\, \rho_1 \,] \varphi, \Pi) \wedge \delta([\, \rho_2 \,] \varphi, \Pi) \\
\delta([\, \rho_1; \rho_2 \,] \varphi, \Pi) &= \delta([\, \rho_1 \,][\, \rho_2 \,] \varphi, \Pi)
\end{aligned} \tag{2.4}
$$

$$\begin{aligned}\delta([\,\rho^*\,]\varphi,\Pi) &= \delta(\varphi,\Pi) \wedge \delta([\,\rho\,]\mathtt{T}_{[\,\rho^*\,]\varphi},\Pi)\\ \delta(\mathtt{F}_\varphi,\Pi) &= \mathit{false}\\ \delta(\mathtt{T}_\varphi,\Pi) &= \mathit{true}\end{aligned}$$

where $\mathbf{E}(\varphi)$ recursively replaces in φ all occurrences of atoms of the form $\mathtt{T}_\psi$ and $\mathtt{F}_\psi$ by $\mathbf{E}(\psi)$; and $\delta(\varphi,\epsilon)$ is defined inductively as above, except for the following base cases:

$$\delta(\langle\,\phi\,\rangle\varphi,\epsilon) = \mathit{false} \qquad \delta([\,\phi\,]\varphi,\epsilon) = \mathit{true} \qquad (\phi \text{ propositional})$$

The role of $\mathtt{T}_\varphi$ and $\mathtt{F}_\varphi$ is to valuate as *true*/*false* or as φ depending on the context. This allows to respect the “test-only” requirement in Definition 2 for the star operator.

Let φ be an LDL$_f$ formula and $\mathcal{A}_\varphi$ the corresponding AFW of equation 2.4. Then, for every trace π, we have that $\pi \models \varphi$ if and only if $\mathcal{A}_\varphi$ accepts π. Also, the space-size of $\mathcal{A}_\varphi$ is linear in the size of φ. Since the emptiness problem for AFWs is PSPACE-complete, we get a proof for Theorem 3 on page 14.

DFAs are the easiest automata to visualize and simulate. A LDL$_f$ formula can be translated to an equivalent DFA through the following transformations: LDL$_f$ → AFW → NFA → DFA. Although the asymptotic worst-case cost of this algorithm is already the best known (which doubly-exponential on the size of the formula), it is possible to combine some or all of these steps in a single algorithm, that may allow some optimizations. A LDL$_f$-to-NFA algorithm is presented in [3]; while a LDL$_f$-to-DFA is described in [6].

Chapter 3

Deep Reinforcement Learning for non-Markovian goals

3.1 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 3.1.

3.1.1 Markov Decision Processes

Most RL algorithms assume that the environment dynamics can be modelled with a *Markov Decision Process* (MDP). They do so, because under the independence

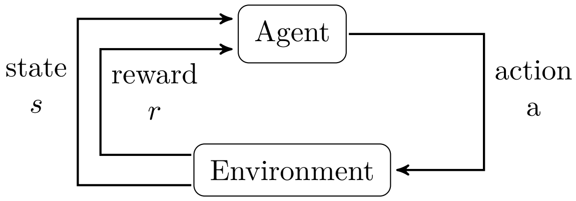

Figure 3.1. How agent and environment interact in RL.

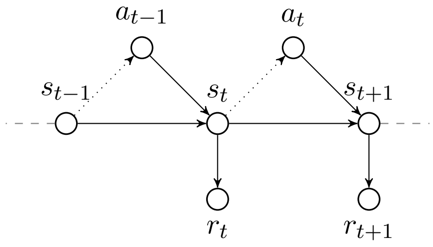

Figure 3.2. The directed graphical model of an MDP.

assumptions taken by MDP, it's possible to efficiently find the optimal agent's policy.

Definition 6. A Markov Decision Process is a tuple $\langle S, A, T, R, \gamma \rangle$, where: S is the set of states 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 taking each action and observing the outcomes. Even if they are unknown, 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-1}, a_{t-1}, s_t). All these assumptions are summarized in the Directed Graphical Model (DGM) of Figure 3.2. In a DGM, edges indicate direct conditional probabilities, while missing arcs indicate conditionally independent variables. In Figure 3.2, 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 5. 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, and a reward function $R(s)$ that returns 1, if the configuration s is a win, -1 for a loss, and 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

[1] In this chapter, variables with an integer subscript or index refer to the value at the discrete time indicated.

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 how the game has been 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 3.3, when T is unknown, as always happens in the real-world, it's much more difficult to prove that we're in fact facing an MDP.

3.1.2 Optimal policies

The *policy* is the criterion the agent uses to select the actions to perform. If the environment dynamics can be modelled with an MDP, the optimal action at time t only depends on s_t. So, there must exist an optimal policy as $\rho_* : S \to A$. However, 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 $a_t \sim \rho(s_t)$. This dependency is represented by the dotted arrows of Figure 3.2. A policy that is a function only of the state is called “stationary”.

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:

$$G \coloneqq r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots = \sum_{t=0}^{T} \gamma^t r_t \tag{3.1}$$

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, where $T \to \infty$. Since the environments we will experiment with 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 ρ computes this quantity from each state s:

$$v_\rho(s) \coloneqq \mathbb{E}_\rho[G \mid s_0 = s] \tag{3.2}$$

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

$$\rho_* : \quad v_{\rho_*}(s) \geq v_\rho(s) \qquad \forall s \in S, \quad \text{for all } \rho \tag{3.3}$$

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 ρ is a similar measure to the value function:

$$q_\rho(s, a) \coloneqq \mathbb{E}_\rho[G \mid s_0 = s, a_0 = a] \tag{3.4}$$

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

$$\rho_*(s) = \arg\max_{a \in A} q_{\rho_*}(s, a) \tag{3.5}$$

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

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} q_*(s_{t+1}, a') \mid s_t = s, a_t = a\right] && (3.6) \\ &= \sum_{s', r'} p(s', r' \mid s, a)\left(r' + \gamma \max_{a'} q_*(s', a')\right) && (3.7) \end{aligned}$$

for any t.

Many learning algorithms exist for estimating q_*. Briefly, on-policy algorithms, estimate q_ρ of the policy ρ that is being used and improved, $\rho \rightarrow \rho_*$; off-policy algorithms, instead, act according to any exploration policy ρ_e 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.

3.1.3 Exploration policies

If q_* were know, equation (3.5) would be enough to always select the optimal action. Generalizing for any q, we call that the *greedy policy*, because it always selects the best action according to q:

$$\rho_q(s) \coloneqq \arg\max_{a \in A} q(s, a) \tag{3.8}$$

Unfortunately, while learning, we only have a rough estimate of the optimal function, $\hat{q} \approx q_*$. Being greedy with respect to sub-optimal values is dangerous, because the agent may deterministically select actions that repeatedly lead to dead-ends. To

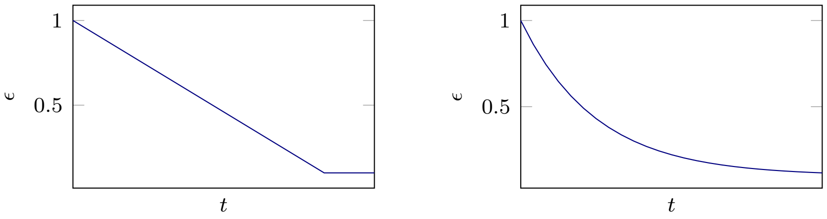

Figure 3.3. Probability of a random action over time: ϵ with linear decay (left), ϵ with exponential decay (right).

mitigate this issue, we can choose some actions at random. The *ϵ-greedy policy* is defined as:

$$\rho_{q,\epsilon}(s) := \begin{cases} \text{random action } a \in A & \text{with probability } \epsilon \\ \arg\max_{a \in A} q(s,a) & \text{otherwise} \end{cases} \tag{3.9}$$

More precisely, random actions are sampled from a uniform distribution over the set of actions A. By making random moves, the agent might escape from suboptimal environment configurations. If $\epsilon = 1$, definition (3.9) reduces to the random policy:

$$\rho_r(s) := \text{random action } a \in A \tag{3.10}$$

When training begins, the agent has no clue about the optimal q-function. It can just try out all actions by executing the random policy. In this phase, the agent receives low rewards but observes a lot of different outcomes for its actions. This is the purpose of exploration. After a while, the agent can begin to trust in its predictions. So, it may gradually choose the most promising actions in order to achieve higher rewards. This is the exploitation phase. The exploitation–exploration trade-off is a fundamental problem in AI. Unfortunately, there's no general solution in RL, because the agent has no way to tell when the policy is "good enough". Usually, we need to try some compromises between the two.

To address this issue, during training, the agent can act according to a policy that is initially stochastic but gradually approaches the greedy policy, over time. There are many ways to do this. One of the most simple options is to select the ϵ-greedy policy of equation (3.9) with ϵ that varies over time according to some schedule. Figure 3.3 shows two common possibilities. On the left-hand figure, the probability of a random action is linearly decreased over time, while on the right, it follows an exponential decay. In both cases ϵ never becomes zero, because that would effectively terminate the learning process. The rate of this decrease is a hyperparameter that can be tuned.

The most important policies are those just described. They can be directly used in a RL algorithm or combined to create more complex policies. The variants designed in this thesis will be presented in the next chapters. With “exploration policy” we refer to any policy that has a strong component of nondeterminism and it’s suitable to drive the agent’s behaviour during training.

3.2 Deep Reinforcement Learning

Classic RL algorithms, such as SARSA and Q-learning, are tabular methods. In fact, 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 NNs, 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 on the set of parameters from input-output samples. Then, the model can be able to generalize to inputs that have been never observed, in a meaningful way.

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

3.2.1 Environment: Atari 2600 games

The Atari 2600 is a video game platform that was developed in 1977. There are hundreds of classic games available to play: Space Invaders, Ms. Pacman, Breakout and many others. The screen is 160 pixels wide and 210 pixels high, with 8-bits colour depth. The joystick has 9 positions (3 for each axis) and one button, for a total of 18 possible actions. For this reason, we'll only focus on RL methods for discrete action spaces.

The Arcade Learning Environment [2] is a simple interface to the Atari 2600 emulator. It allows agents to play and be trained on these games. At each step, the agent chooses one of the 18 actions available and receives in return a frame of the game and a reward. The reward is the increment in the player's score for the original game. This is really the same interface that a human player would use. Figure 3.4 shows the frames from few games in this collection.

Although these games come from an early stage of video games development, they represent the appropriate challenge for current (Deep) Reinforcement Learning agents. In fact, many papers tested their RL algorithms on these games [17][16][25][12]. In this thesis, we also tested with some of these environments. We will also show how improve on the hardest game in this collection for a RL agent: Montezuma's

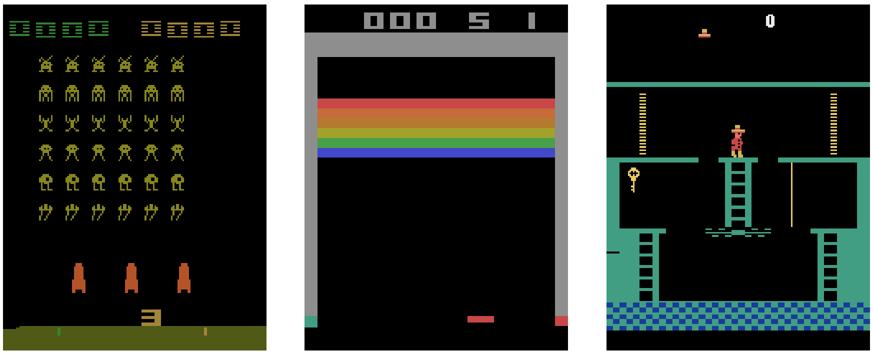

Figure 3.4. Initial frames of some Atari 2600 games (left to right): Space Invaders, Breakout, Montezuma's Revenge.

Revenge.

3.2.2 Deep Q-Network

The *Deep Q-Network* (DQN) [17] was the first algorithm to successfully combine deep learning models and Reinforcement Learning. Although many basic ideas presented here have been already introduced by the Neural Fitted Q iteration algorithm [20], DQN addressed some causes of training instability. They also demonstrated that exactly the same agent can be trained in many Atari games and achieve human-level performances in many of those [16]. These promising results sparked a lively interest in Deep RL, recently.

In DQN, the state-action value is approximated by a deep neural network $Q(s, a; \theta)$, on the parmeters θ, that we call Q-Network. The purpose of learning, is to train this network to approximate the optimal q-function: $\hat{\theta} : Q(s, a; \hat{\theta}) \approx q_*(s, a)$. Then, the estimated optimal policy will be:

$$\hat{\rho}(s) = \arg\max_{a \in A} Q(s, a; \hat{\theta}) \tag{3.11}$$

A trained network, for each input (s, a), should return the expected value of some target $y_{s,a}$. To do so, we select the parameters that minimize the squared difference between the estimates and the targets:

$$\text{loss}(\theta) \coloneqq \big(Q(s, a; \theta) - y_{s,a}\big)^2 \tag{3.12}$$

Since this is a Q-Network, the targets are the optimal state-action values $q_*(s, a)$ that the net should estimate. The loss (3.12) contains some random variables. So, we minimize it through any stochastic optimization algorithm. In Stochastic Gradient Descent (SGD), at each step t, we observe an input (s_t, a_t) and the associated target

y_t. Then, we take a small step toward the negative gradient of the loss:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \left((Q(s_t, a_t; \theta) - y_t)^2 \right) \Big|_{\theta=\theta_t} \tag{3.13}$$

in which $0 < \alpha < 1$ is a small learning rate. This equation is not the only update rule possible. There are more advanced optimization algorithms, such as: Momentum, RMSprop and Adam. In this thesis, we've mostly experimented with Adam.

What has just been described is the usual way of fitting a neural network to a dataset of samples. In RL, however, the targets $q_*(s_t, a_t)$ are unknown, because they depend recursively from the same optimal q-function that we're trying to learn (see equation (3.7)). In classic RL, this is not a problem: the 1-step approximation of the q-values (derived from equation (3.7)),

$$y_t \coloneqq r_{t+1} + \gamma \max_{a \in A} \hat{q}(s_{t+1}, a) \tag{3.14}$$

or the n-step approximation, are a valid targets for the function $\hat{q}$. By updating toward these values on the whole input space, convergence is guaranteed. In other words, targets can be estimates themselves.

With neural networks, instead, any update to the parameters also affects the target, because the weights have a global influence on the function. It's not possible apply a correction for just one tiny region of the input space (nor it's desirable, after all). It has been shown [20], that due to this effect, propagating errors slow down convergence or even render the training unstable. To address this issue one must ensure that the targets do not move much.

The DQN [17] algorithm addresses this issue in two ways. First, the targets in equation (3.14) are not generated by the network that is being trained, $Q(s, a; \theta)$, but they are computed from a second net, $Q(s, a; \theta')$. Every C iterations, the target net is updated to match the trained net, with the assignment: $\theta' \leftarrow \theta$. This keeps the targets constant for C steps and helps to stabilize the training.

Second, the network is not trained from the last sample, but from transitions of the recent experience. At each step, the agent acts according to some exploration policy, $a_t \sim \rho_e$. Each transition, of the form $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$, is recorded in a buffer of size n_r, called "experience replay". Then, at each training step, we sample a number of n_b transitions, thus creating a batch, and we perform an update $\theta_{i+1} = \theta_i - \alpha\, g_i$ on the cumulative gradient g_i of the whole batch.

DQN also includes a number of heuristics that greatly help the training but are specific to the Atari 2600 environments:

- Rewards can be really high, so they are limited in the range $[-1, +1]$; this is called *reward clipping.* It helps to keep the same learning rate for diverse games.

- The agent has a single life available. When a life is lost, the episode ends. This prevents the agent to rely on restarts.
- The frames are slightly down-scaled to further reduce the resolution, they are transformed to gray-scale and mapped to the range $[-1, +1]$. These are common preprocessing steps for NNs.
- Every observation is composed by the last 4 frames stacked together. This allows the agent to observe how the objects in the scene move. See Section 3.3 and Example 7 on page 30.

The algorithm used in this thesis is called *Double DQN* [25]. It is a slight variant of DQN, so all details mentioned so far also apply. The motivation of this algorithm is a known issue of Q-learning: it is likely to make overoptimistic value estimates. To show this, let's rewrite the targets of (3.14) as:

$$y_t \coloneqq r_{t+1} + \gamma\, Q(s_{t+1}, \arg\max_{a \in A} Q(s_{t+1}, a; \theta_t); \theta_t) \tag{3.15}$$

where the estimates $\hat{q}$ are computed with the Q-Network. This form makes more evident that the same model is used both to select the next greedy action and to estimate the q-value of state s_t. As result, any action with an overestimated q-value will be selected and its value propagated. To remove this bias, Double DQN decouple the two operations by using different sets of parameters, $\theta^{(1)}$ and $\theta^{(2)}$. The targets y_t are computed as:

$$y_t \coloneqq r_{t+1} + \gamma\, Q(s_{t+1}, \arg\max_{a \in A} Q(s_{t+1}, a; \theta_t^{(1)}); \theta_t^{(2)}) \tag{3.16}$$

Then, just the parameters $\theta^{(1)}$ are updated toward this targets; this is called the online network. With random chance, the roles of the two parameters are continuously swapped at each step.

To compute the target, we need to compute the q-values for all actions in state s_{t+1}. To speed up this computation, the network is defined as a function that takes in input a state and computes a vector of state-action values, one for each action. So, just one forward pass is required to select the next action. Common Q-Networks for images are composed of a number of convolutional layers and some fully-connected layers. The specific structure may change, and the network used will be defined in the implementation section.

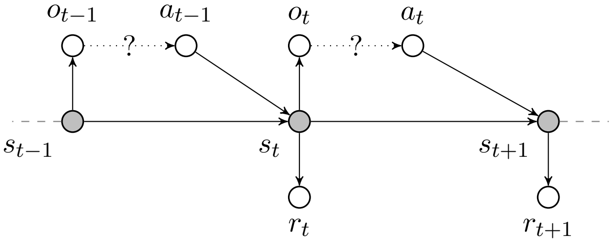

Figure 3.5. The Directed Graphical Model of a POMDP. Gray nodes are unobservable. For simplicity, the rewards in this graph depend just on the current state s_t, not on transitions (s_t, a_t, s_{t+1}).

3.3 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:

$$r_{t+1} \not\perp s_i, a_i, r_i \mid s_t, a_t \qquad \text{for some } t, i, \text{ with } 0 \leq i < t \tag{3.17}$$

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.

3.3.1 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 in MDPs, 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 3.5. 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 3.5 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 $\rho_* : \Omega \to A$? 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, probabilistic methods can be effective estimation algorithms. The iterative probabilistic filter applied to the sequence of observations would produce the belief distribution of the current state. We can represent the general procedure, at any instant t, with the following computation:

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

where $b(s)$ denotes the belief of s, being either a set of states or a probability distribution. Since these beliefs depend 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? 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 3.6 represents this situation.

Example 6. As we've seen from Example 5 on page 18, the game of Chess can be modelled with an MDP if we consider as states the vectors of positions of all pieces

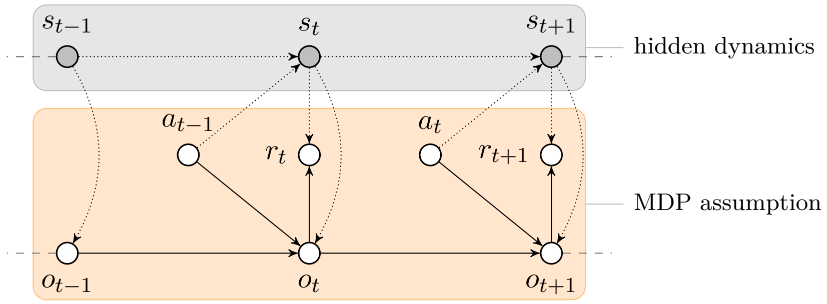

Figure 3.6. The dotted arrows ⇢ represent the dependencies in a POMDP model. Solid arrows $\longrightarrow$ show the MDP model over the same quantities.

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 the 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 without any memory about the position of the hidden pieces, it's not possible to predict the next image and reward from the current observation.

Example 7. 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, which may be represented in 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 $S \coloneqq \{\theta \in [-\pi, +\pi]\}$? 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 in the state space 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, in a video game where the agent has to hit a moving ball, the optimal policy certainly needs to observe also its direction.

3.3.2 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 events

$$r_t = R(\langle s_0, s_1, \dots, s_t \rangle) \qquad \forall t \in \mathbb{Z} \tag{3.18}$$

with $R : S^* \to \mathbb{R}$. The sequence of states $\pi \coloneqq \langle s_0, s_1, \dots, s_t \rangle$ will be also called execution *trace*. In general, with the term "trace" we indicate any sequence that is produced during a run. We adopt a similar notation to those we've seen for interpretations of temporal logics.

Goals defined by rewards of equation (3.18) are said "temporally-extended" because they take into account multiple timesteps. Why should we define a reward function that is explicitly non-Markovian? One possibility is that we might want our agent to drive the environment through a *sequence* of states, instead of just reaching a single configuration. However, as we will see, we don't need to restrict to sequences, because we may define very complex reward functions.

Example 8. Let's suppose the agent can control a light bulb through a switch, and we want the light to be set on, then off again. The agent will be rewarded if, at the end of the episode, the light has been set on only once. The environment is extremely simple: its state may be completely described by a Boolean variable, "lightOn", which reflects the status of the light. Still, in order to valuate whether the task has been accomplished, we need to consider the whole sequence of states, not just the final configuration.

We now define a model that, by generalizing MDPs, can describe this large class of problems.

Definition 7. A *Non-Markovian Reward Decision Process* (NMRDP) [1] is a tuple $\langle S, A, T, R, \gamma \rangle$, where S, A, T, γ are defined as for MDPs, and $R : S^* \to \mathbb{R}$ is a non-Markovian reward function, which computes the reward at time t as $r_t = R(\langle s_0, s_1, \dots, s_t \rangle)$.

Every NMRDP admits an optimal policy as $\rho_* : S^* \to A$, which computes actions from the history of states. So, we'll only consider policies with this form. In order to define optimality, we would need to proceed as for MDPs, by defining value functions. However, this is sightly more complex, since the effect of non-Markovian rewards is that value functions can only predict the future expected discounted return, if the past history is given. They effectively compare policies on traces, rather than single states. The simplest case is the valuation of any initial state, whose value function is [1]:

$$v_\rho(\langle s_0 \rangle) \coloneqq \mathbb{E}_\rho \left[\sum_{t=0}^{T} \gamma^t R(\langle s_0, s_1, \dots, s_t \rangle) \right] \tag{3.19}$$

Informally, an NMRDP policy ρ_* is optimal if it maximizes the value function of future states. However, we won't further delve into the definition of optimality and value functions, because common solution methods (that we'll see in Section 3.4.1) transform NMRDPs into standard MDPs.

NMRDP with LDL$_f$ rewards

Non-Markovian reward functions have huge domains. Defining them by listing all the traces that should be (positively or negatively) rewarded is unfeasible, even for the simplest cases. Fortunately, as we already know from Chapter 2, temporal logics are powerful formalisms that allow to concisely define groups of traces. So, a very effective way to declare non-Markovian rewards is through a set of pairs $\{(\varphi_i, r_i)_{i=1}^m\}$, where each φ_i is a LDL$_f$ formula and r_i is its associated reward [3]. The reward r_i will be produced whenever a trace satisfies φ_i. So, the reward function is defined as:

$$R(\pi) \coloneqq \sum_{i \,:\, \pi \models \varphi_i} r_i \tag{3.20}$$

It follows that an equivalent way to define NMRDPs is: $\langle S, A, T, \{(\varphi_i, r_i)_{i=1}^m\}, \gamma \rangle$.

In this thesis, rewards will be always declared with LDL$_f$ formulae. However, the same discussion also applies to LTL$_f$. Also, we may have noticed that the adoption of temporal logics requires a state space that is composed of propositional interpretations. This will be addressed in Section 3.4.2.

3.4 Reinforcement Learning with LDL$_f$ specifications

This section illustrates how to learn optimal policies for a large class of problems among those introduced in Section 3.3. The main idea behind the techniques presented here is to formulate an appropriate NMRDPs with LDL$_f$ rewards, and to solve it through an equivalent Markov Decision Process. Since many learning algorithms exist for MDPs, this translation can be considered as a solution for the original problem.

3.4.1 RL for NMRDPs with LDL$_f$ rewards

RL for NMRDPs

Before looking at the construction, we need to define what is an *equivalent* MDP and what are its properties.

Definition 8. [1] An NMRDP $\mathcal{N} := \langle S, A, T, R, \gamma \rangle$ is *equivalent* to an extended MDP $\mathcal{M} := \langle S', A, T', R', \gamma' \rangle$ if there exist two functions $\tau : S' \to S$ and $\sigma : S \to S'$ such that:

1. $\forall s \in S : \tau(\sigma(s)) = s$;
2. $\forall s_1, s_2 \in S$ and $s'_1 \in S'$: if $T(s_1, a, s_2) > 0$ and $\tau(s'_1) = s_1$, there exists a unique $s'_2 \in S'$ such that $\tau(s'_2) = s_2$ and $T'(s'_1, a, s'_2) = T(s_1, a, s_2)$.
3. For any feasible trajectory $\langle s_0, a_1, \ldots, s_{n-1}, a_n \rangle$ of $\mathcal{N}$ and $\langle s'_0, a_1, \ldots, s'_{n-1}, a_n \rangle$ of $\mathcal{M}$, such that $\tau(s'_i) = s_i$ and $\sigma(s_0) = s'_0$, we have $R(\langle s_0, a_1, \ldots, s_{n-1}, a_n \rangle) = R'(\langle s'_0, a_1, \ldots, s'_{n-1}, a_n \rangle)$.

Conditions 1 and 2 require that every feasible trajectory of the NMRDP can be simulated with a trajectory of the MDP. Condition 3 forces corresponding trajectories to produce the same rewards. So, the equivalent MDP completely captures the dynamics of the NMRDP. As we will see, in order to do this, the new state space S' needs to include the old states S and some history-related informations. Since S' is always larger than S, the equivalent MDP is also called “extended”.

Definition 9. [1] Let $\rho' : S' \to A$ be a policy for the MDP $\mathcal{M}$. The corresponding policy $\rho : S^* \to A$ of the NMRDP $\mathcal{N}$ is defined as $\rho'(s'_n) := \rho(\langle s_0, \ldots, s_n \rangle)$ where $\langle s'_0, \ldots, s'_n \rangle$ is the corresponding trajectory for $\langle s_0, \ldots, s_n \rangle$.

As we can see ρ' is a stationary policy. A very important result for equivalent MDPs is the following:

Theorem 4. *[1] For any policy ρ' for the MDP $\mathcal{M}$, its corresponding policy ρ for the NMRDP $\mathcal{N}$, and $s \in S$, we have $v_\rho(s) = v_{\rho'}(\sigma(s))^2$.*

As a corollary of the previous theorem, any optimal policy of the MDP has a corresponding policy that is optimal for the NMRDP. This is is the result we were looking for: by applying classic RL algorithms we can learn optimal policies of MDPs and their equivalent NMRDP. In practice, we don't need to translate the policy ρ' to the non-stationary equivalent ρ. It is possible to apply the trained RL agent directly to the NMRDP, by continuously transforming each observation s through the translation function $\sigma : S \to S'$.

[

LDLf rewards]LDL$_f$ rewards

We will consider NMRDPs with LDL$_f$ rewards: $\mathcal{N} \coloneqq \langle S, A, T, \{(\varphi_i, r_i)_{i=1}^m\}, \gamma \rangle$, where $S \coloneqq 2^{\mathcal{F}}$ is a set of propositional interpretations and φ_i are LDL$_f$ formulae on the set of fluents $\mathcal{F}$.

3.4.2 RL with LDL$_f$ restraining specifications

3.4.3 Restraining Bolt for partial observations

How RL for non-Markovian goals can solve partial observations?

[2] In this equation, v refers to the value function for NMRDPs and MDP respectively.

3.5 Restrained Deep RL agents

Deep agent with RB.

Chapter 4

Learning to valuate fluents in games

The importance of correctly valuate the fluents.

4.1 Temporal constraints

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

4.2 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.

4.3 General structure of the model

Illustration and general description of the model.

4.4 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].

4.4.1 Model: Deep Belief Network

4.4.2 What does it learn

4.5 Boolean functions

The fluents are true in a set of those configurations.

4.5.1 Learning with genetic algorithms

Ideas from concept learning; genetic algorithm.

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

4.5.2 Boolean rules

Representation of boolean functions and training details.

Chapter 5

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.

5.1 How to use the software

5.1.1 Tools and setup

5.1.2 Commands

Small user reference.

5.2 Implementation

5.2.1 `agent` Module

`training` Module

`playing` Module

5.2.2 `streaming` Module

5.2.3 `features` Module

`models` Module

`genetic` Module

`temporal` Module

Chapter 6

Esperiments

6.1 Breakout

6.1.1 Definitions

6.1.2 Training

6.1.3 Comments

6.2 Montezuma's Revenge

6.2.1 Definitions

6.2.2 Training

6.2.3 Comments

Chapter 7

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] Martin Riedmiller. “Neural fitted Q iteration - First experiences with a data efficient neural Reinforcement Learning method”. In: *Lecture Notes in Computer Science* 3720 LNAI (2005), pp. 317–328. ISSN: 03029743. DOI: `10.1007/11564096_32`.

[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.

[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.

[26] Moshe Y. Vardi. "An automata-theoretic approach to linear temporal logic". In: *Lecture Notes in Computer Science* 1043 (1996), pp. 238–266. ISSN: 16113349.