SAPIENZA
UNIVERSITÀ DI ROMA

Symbol Grounding in Deep Reinforcement Learning agents for non-Markovian rewards

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

Symbol Groundingin Deep Reinforcement Learning agentsfor non-Markovian rewards

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

Alla mia famiglia

Abstract

Reinforcement Learning (RL) and Deep RL, although different, both share the strong assumption that the environment generates Markovian rewards. A method, originally demonstrated for RL, adopts the temporal logics $\text{LTL}_f/\text{LDL}_f$ to declare non-Markovian goals and train agents on these tasks. In this thesis, we apply the same method to Deep RL, demonstrating that it's possible to successfully apply the same construction.

The complex observations used in Deep RL introduce new difficulties to be addressed: how can we *ground* the logic abstraction, composed of atomic propositions, to match the environment state? Most of this thesis focus on this central problem, for a specific class of observations and propositions. We propose an original solution that employs temporal logic to specify the expected temporal behaviour of the propositions to extract. We also define a specific model that allows to apply these ideas on the Atari games, a class of environments frequently used in Deep RL, and we evaluate the effectiveness of this method in two of those.

Contents

Nomenclature

AFW	Alternating Automaton on finite Words, page 13
DBN	Deep Belief Network, page 60
DFA	Deterministic Finite-state Automaton, page 14
DGM	Directed Graphical Model, page 18
DNF	Disjunctive Normal Form, page 64
DQN	Deep Q-Network algorithm, page 24
FOL	First-Order Logic, page 8
GA	Genetic Algorithm, page 64
LDL_f	Linear Dynamic Logic of finite traces, page 10
LTL	Linear Temporal Logic, page 7
LTL_f	Linear Temporal Logic of finite traces, page 8
MDP	Markov Decision Process, page 17
MRF	Markov Random Fields, page 54
MSO	Monadic Second-Order Logic, page 8
NFA	Nondeterministic Finite-state Automaton, page 13
NMRDP	Non-Markovian Reward Decision Process, page 31
NN	Neural Network, page 23
NNF	Negation Normal Form, page 14
PCD	Persistent Contrastive Divergence, page 58
PDL	Propositional Dynamic Logic, page 8

POMDP	Partially Observable Markov Decision Process, page 27
RBM	Restricted Boltzmann Machine, page 57
RDP	Regular Decision Process, page 40
RL	Reinforcement Learning, page 17
SGD	Stochastic Gradient Descent algorithm, page 24
UGM	Undirected Graphical Models, page 54

Chapter 1

Introduction

A classic and important branch of Artificial Intelligence (AI) aims at developing agents that select their actions through forms 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 atomic propositions with the term "*fluents*" (a name that suggests that their truth can change over time).

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 process is called *grounding* and it is essential in order to make reasoning relevant for the external environment. Unfortunately, this can be really hard in complex environments, because the agent's sensors may generate noisy and multidimensional inputs, that are difficult to interpret.

Reinforcement Learning (RL) is a successful field of AI, in which the agent's goal is to learn a policy that maximizes the rewards received. We could argue that RL does not require the valuations just mentioned. Still, rewards and punishments must be supplied somehow, 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, complex and non-Markovian 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 combination required to investigate new ways to solve the grounding problem just described, for a specific class of environments and fluents.

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 [25].

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 [10]. 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 [20] 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 any modification, the same agent has been able to learn and reach human-level performances in many of these 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 games from this collection.

The reinforcement learning algorithm we've adopted is called Double DQN [28]. 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 [19]. In fact, among Q-Network algorithms, the only ones that were able to clearly achieve superior performances in most of these games have combined many of the others DQN variants [13].

If we look at the results in [19], 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, it doesn'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 [18]. In Section 4.1, 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 [4] and [6]. The former work [4] 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 the authors 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 [6], instead, has shown that through a similar construction, it is possible to influence the agent's optimal behaviour by declaring additional non-Markovian rewards that are produced in conjunction with the original rewards. This paper named this additional module with "Restraining Bolt". Thorough this thesis, it may be practical to use this name to refer to this logic construction.

1.2 Objective and results

This work started with the following goal: defining and implementing a Deep RL agent that, through the Restraining Bolt, is able to achieve non-Markovian tasks. This first objective has been accomplished, but the Restrained Bolt is a method based on logic, and, as we've already anticipated in this introduction, it requires to correctly valuate the fluents we define. Since the environments adopted in Deep RL have much more complex observation spaces, it has been necessary to investigate new ways to solve the problem of the fluents valuation, at least for a specific class of fluents and observations.

Thus, it is possible to isolate two groups of contributions of this thesis: those concerning the definition of a Deep RL agent for non-Markovian goals, and those related to methods for grounding the fluents to the environment configuration. Regarding the former, in this thesis:

- We provide a flexible implementation of the Restraining Bolt method described in [6] and [4].
- We proposed a Neural Network architecture for a Deep RL agent that allowed to successfully employ the Restraining Bolt in Deep Reinforcement Learning.
- Tests have been conducted in a video game called *Montezuma's Revenge*, the hardest game (for a RL agent) in the Atari 2600 collection [19]. We've shown that the agent can be successfully guided through the initial room of the game, with low manual intervention.

The latter topic, instead, received much attention in this thesis: how is it possible to learn a function that, given an observation of the environment, predicts whether the fluents we have defined are true? As we may recognise, this problem is really general, and we must restrict to a specific class of fluents and observations. Observations will be frames of the Atari games, and fluents will be propositions decidable from each of these images. Every choice or assumption that will further

restrict the applicability of the proposed method will be pointed our along the text. Still, there are some interesting achievements of this work that are to be highlighted:

- We don't manually assign a meaning to some Boolean features. Instead, this is an initial investigation about how to proceed in the opposite direction: we first define a set of symbols, then we learn their valuation function.
- We adopt the temporal logic LDL_f as a formalism to define some temporal constraints that our fluents are always expected to satisfy. Candidate valuation functions will be checked against these constraints.
- The training algorithm won't require any manual annotation, nor labelled datasets at all.
- We propose an architecture, based on Deep Belief Networks, that by reducing the input space dimensionality, binds the valuations to some visual features that are recognizable in the image.

All these ideas have been implemented in a Python software and tested on games of the Atari collection.

Finally, we also strongly contributed to the development of the `flloat` package [7], a Python software for the conversion of LTL_f and LDL_f formulae to the associated automaton (NFA or DFA).

1.3 Structure of the thesis

The rest of this thesis is structured as follows:

2 – Temporal logics and Linear Dynamic Logic
Temporal logics are an important formalism for this work and they will be used throughout the text. This chapter introduces the reader to concepts such as: fluents, traces and linear-time temporal logics. Then, we define the Linear Dynamic Logic (LDL_f), that is the specific temporal logic used in this text.

3 – Reinforcement Learning
Here, we describe the second large group of background topics for this thesis. We introduce the fundamental concepts of RL, Deep RL, and we define Double DQN, the Deep RL algorithm that we use.

4 – Reinforcement Learning for non-Markovian rewards
This chapter describes how to design RL agents that are able to learn in presence of non-Markovian rewards. Here, we see an in-depth description of the problem,

we build on recent solution methods, and we propose an original view. In Section 4.1, we analyze what happens when the most common assumptions of RL, and of Deep RL, are falsified. A recent solution for these complex RL problems is presented in Section 4.2, where we explain the Restraining Bolt method. In this section, we also illustrate our original view, in which two sources of non-Markovian rewards are compared: the standard one, coming from the Restraining Bolt, and a secondary one coming from incomplete partial obaservability of the environment. In addition, in Section 4.3, we propose an original model that allows us to apply the Restraining Bolt method in the context of Deep RL.

5 – Learning to ground symbols trough RL

In this chapter, we propose an original training method to ground the fluents valuations to the features of the environment. Here, we explicitly define a model for the fluents valuation functions. Such model adopts unsupervised learning techniques to encode the relevant portion of the input frame to a set of compact features, which simplify the training process. Finally, we formulate an appropriate training algorithm for this model.

6 – Tool for learning symbols through RL: the AtariEyes package

This chapter presents the software that we've developed, which implements all the concepts presented in the previous chapters. We first review its features and its functionality from a user perspective. Then, we illustrate some interesting details of the implementation.

7 – Esperiments

Here, we look at the experiments and training outcomes in two Atari games. The experiments are finalized to test: the effectiveness of the proposed method for learning the valuation functions, and the combination of such computed fluents in a Deep RL agent with the Restraining Bolt.

8 – Conclusions

In this final chapter, we draw the main conclusions that can be derived from this work, the strength of this approach and its weakness, and many of the 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 [11]. 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) [22].

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 $V : T \times \mathcal{F} \to \{true, false\}$. 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 [5], 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 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) [5]. Its syntax combines regular expressions and propositional logic, just like Propositional Dynamic Logic (PDL) does [9][27]. So, we will review regular expressions first.

2.2 Regular Temporal Specifications

Regular languages are the class of languages exactly recognized by finite state automata and regular expressions [15]. 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

[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.

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:

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

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:

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

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 [5]. 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 [4].

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 [9], 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: $\mathit{last} \coloneqq |\pi| - 1$.

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:

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

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

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]:

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

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 $\langle\, true \,\rangle tt$. 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 $last + 1$. 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 $[\, true^* \,] A$ 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 $[\, true^* \,](A \vee End)$, or $[\, true^*; (\neg End)? \,] A$.

[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 $\langle\, A^*; true \,\rangle B$, instead, they represent the temporal operator containing a regular temporal specification.

In [5] (De Giacomo and Vardi) some important theorems have been established for LDL_f[3]:

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 [15].

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 $\delta(q_0, \sigma) = (q_1 \vee q_2 \vee q_3)$, because the agent can "choose" among these states. The dual operation is to require that *all* the following runs needs to be accepting: $\delta(q_0, \sigma) = (q_1 \wedge q_2 \wedge q_3)$. The transition function of an AFW adopts boolean functions without negations to allow both conjunctions and disjunctions. For example, we may write $\delta(q_0, \sigma) = q_1 \wedge (q_2 \vee q_3)$. 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

[3]The following theorems have been proved in [5] for the original definition of LDL_f, which is slightly different. The same results are valid for the empty trace variant presented here.

are either final or are labelled with *true*[4]. For a more precise description of runs and acceptance condition of AFWs, see [29].

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 [29].

Delta function

In this section, given a LDL_f formula φ, we define its associated AFW, $\mathcal{A}_\varphi \coloneqq \langle \Sigma, Q, q_0, \delta, F \rangle$. The alphabet, $\Sigma \coloneqq 2^{\mathcal{F}}$, 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 $q_0 \coloneqq \varphi$. The set of final states is empty, $F \coloneqq \{\}$, 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 $nnf(\cdot)$. 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:

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

[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.

$$\begin{aligned}
\delta([\,\psi?\,]\varphi,\Pi) &= \delta(nnf(\neg\psi),\Pi) \vee \delta(\varphi,\Pi)\\
\delta([\,\rho_1+\rho_2\,]\varphi,\Pi) &= \delta([\,\rho_1\,]\varphi,\Pi) \wedge \delta([\,\rho_2\,]\varphi,\Pi)\\
\delta([\,\rho_1;\rho_2\,]\varphi,\Pi) &= \delta([\,\rho_1\,][\,\rho_2\,]\varphi,\Pi)\\
\delta([\,\rho^*\,]\varphi,\Pi) &= \delta(\varphi,\Pi) \wedge \delta([\,\rho\,]\mathtt{T}_{[\,\rho^*\,]\varphi},\Pi)\\
\delta(\mathtt{F}_\varphi,\Pi) &= \mathit{false}\\
\delta(\mathtt{T}_\varphi,\Pi) &= \mathit{true}
\end{aligned}$$

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 13.

DFA translation

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$_f$ $\rightarrow$ AFW $\rightarrow$ NFA $\rightarrow$ DFA. The asymptotic worst-case cost of this procedure is already the best known, which doubly-exponential on the size of the formula. Still, 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 [4]; while a LDL$_f$-to-DFA is described in [8].

These automata have an input alphabet Σ of size $2^{|\mathcal{F}|}$. This means that a complete DFA has a large number of arcs. Instead of connecting each pair with propositional interpretations, which can be a huge enumeration of sets, we use formulae of propositional logic. The automaton traverses a transition $q_j \xrightarrow{\phi} q_i$ if it is in state q_j and it receives an input Π such that $\Pi \models \phi$. In a deterministic automaton, from each state, exactly one transition is satisfied for any input. The formulae ϕ are just a form of "syntactic sugar" that often leads to more concise visualizations of the automata. An example of this equivalence is shown below:

Chapter 3

Reinforcement Learning

3.1 Basic principles

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 [25], [24], and [21] 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 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

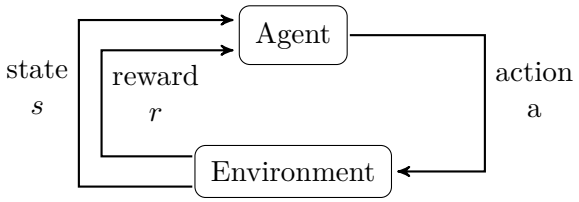


Figure 3.1. How agent and environment interact in RL.

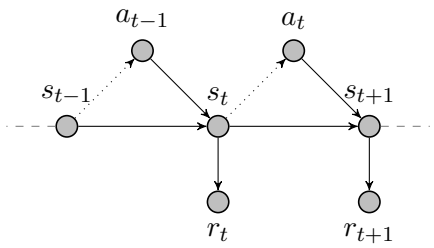


Figure 3.2. The directed graphical model of an MDP.

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: $s_{t+1} \perp s_0, \ldots, s_{t-1} \mid s_t, a_t$. 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 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

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

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 4.1, 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:

$$q_*(s,a) = \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} q_*(s_{t+1}, a') \mid s_t = s, a_t = a\right] \tag{3.6}$$

$$= \sum_{s',r'} p(s', r' \mid s, a)\left(r' + \gamma \max_{a'} q_*(s', a')\right) \tag{3.7}$$

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, $\rho \to \rho_*$; 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 policies is dangerous, because the agent may deterministically select actions that repeatedly lead to dead-ends. To mitigate this issue, we can choose some actions at random. The *ϵ-greedy policy* is defined as:

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

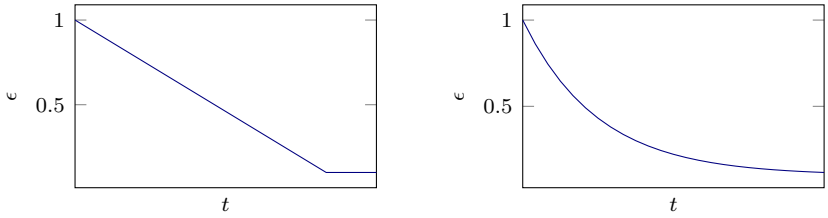


Figure 3.3. Probability of a random action over time: ϵ with linear decay (left), ϵ with exponential decay (right).

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:

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

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 common policies are those just described. They can be directly used in a RL algorithm or combined to create more complex policies. 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.

Custom exploration policies

We'll now define few other policies which we proposed and used in this thesis. Of course, there is no better policy in general. The policies defined here might be suitable for the environments we face, but may be inappropriate in many others. They just happen to be useful for one class of environments.

The first one is an ϵ-greedy policy with action repetition. It chooses between random and deterministic actions just like the ϵ-greedy. However, consecutive random moves always execute the same action. This sequence of repetitions can be interrupted by a deterministic move or by the threshold of maximum repetitions. This policy may be useful in environments where the effect of a single action is very small. This is the case, for example, for the exploration of mazes and corridors. The random policy wouldn't allow the player to cover large distances, because of the uniform sampling between left and right.

The second policy is an ϵ-greedy with random ϵ. As we will see in Section 5.6, when we need to observe the environment dynamics, we need to see the observations received in response to many different stimuli. However, when we apply ϵ-greedy to a capable agent, we see the following pattern: ϵ determines the agent's ability. To a fixed ϵ corresponds some average ability, and the cumulative rewards achieved tend to be very similar. Following this intuition, we define a policy that samples a random ϵ for each episode. So, at different episodes, the agent can explore both the early stages and more distant environment states.

The last policy we define addresses the same problem as the previous one, producing very diverse trajectories, but in environments with sparse rewards. When the reward is sparse, we can read it as successful completion of some sub-task. In this policy, for each episode, we sample a random natural number, that we call "checkpoint". When the number of rewards collected is lower than the checkpoint, the agent behaves mostly in a deterministic way, because the purpose is just to proceed. When the number of rewards reaches the checkpoint number, we act according to a random policy. The idea is to explore the environment state space at different depths.

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

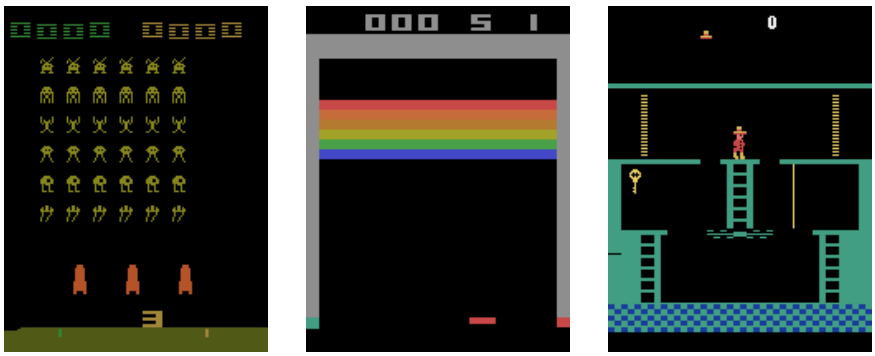


Figure 3.4. Initial frames of some Atari 2600 games (left to right): Space Invaders, Breakout, Montezuma's Revenge.

(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 Popular testbed for Deep RL: Atari 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 RGB colors of 8-bits 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 [20][19][28][13]. 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 Revenge.

3.2.2 Deep Q-Network

The *Deep Q-Network* (DQN) [20] 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 [23], 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 [19]. 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:

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

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:

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

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 [23], 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 [20] 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 4.1 and Example 7 on page 30.

The algorithm used in this thesis is called *Double DQN* [28]. 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.

Chapter 4

Reinforcement Learning for non-Markovian rewards

4.1 Non-Markovian goals

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

$$r_{t+1} \not\perp s_i, a_i, r_i \mid s_t, a_t \qquad \text{for some } t, i, \text{ with } 0 \leq i < t \tag{4.1}$$

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.

4.1.1 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 (possibly stochastic) observation function $O : S \times \Omega \to \mathbb{R}$.

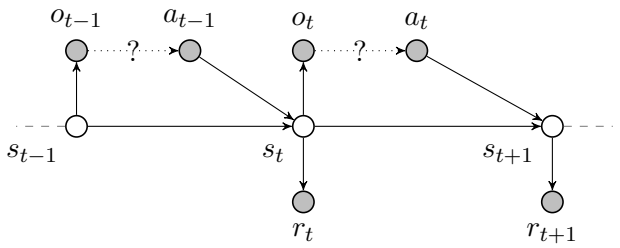
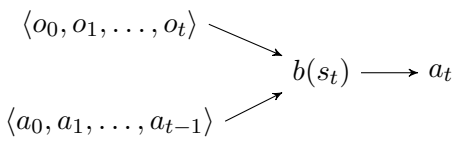


Figure 4.1. The Directed Graphical Model of a POMDP. White nodes are unobservable. For simplicity, the rewards in this graph depend just on the current state s_t, not on transitions (s_t, a_t, s_{t+1}).

The graphical model of a POMDP is shown in Figure 4.1. 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 4.1 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 POMDPs always admit an optimal policy of the form $\rho_* : \Omega \to A$? Unfortunately, the answer is no. Other informations are needed.

When the POMDP transition and observation functions are known, a common solution is to estimate the current unobservable state 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, and the estimation algorithm is some recurrent probabilistic inference. In both cases, the algorithm would produce a belief of the current state, denoted with $b(s_t)$. We can represent the general procedure, at any instant t, with the following computation:



Since these beliefs depend on the whole sequence of observations, computing the next action from $b(s_t)$ also makes it dependent from the whole history.

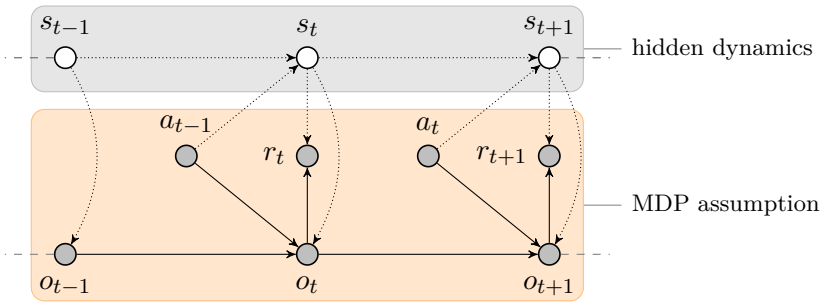


Figure 4.2. The dotted arrows ⇢ represent the dependencies in a POMDP model. Solid arrows → show the MDP assumption over the same observable. This is usually a very poor approximation of the real dynamics.

Standard RL algorithms cannot be applied to POMDPs, because the state space is not observable. Also, since we allow 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. 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 "relevant" 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 functions $T_\Omega : \Omega \times A \times \Omega \to \mathbb{R}$ 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 as an MDP. However, since this is often not the case, we cannot apply RL algorithms to POMDPs. Figure 4.2 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 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 $\dot{\theta}$.

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.

4.1.2 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. This is the case when each reward is computed from the whole history of events

$$r_t = R(\langle s_0, s_1, \ldots, s_t \rangle) \qquad \forall t \in \mathbb{Z} \tag{4.2}$$

with $R : S^* \to \mathbb{R}$. The sequence of states $\pi \coloneqq \langle s_0, s_1, \ldots, s_t \rangle$ 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 (4.2) 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 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, it's not sufficient to check whether the light is off at the end of the episode; we also need to ensure that, *during the whole episode*, it has been switched on only once.

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 $r_t = R(\langle s_0, s_1, \ldots, s_t \rangle)$.

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 as a consequence of non-Markovian rewards, 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]:

$$v_\rho(\langle s_0 \rangle) \coloneqq \mathbb{E}_\rho \left[\sum_{t=0}^{T} \gamma^t R(\langle s_0, s_1, \ldots, s_t \rangle) \right] \tag{4.3}$$

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 4.2.1) transform NMRDPs into standard MDPs, that we already know how to solve.

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 [4]. The reward r_i will be produced whenever a trace satisfies φ_i. So, the reward function is defined as:

$$R(\pi) \coloneqq \sum_{i\,:\,\pi \models \varphi_i} r_i \tag{4.4}$$

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 4.2.2.

4.2 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 4.1. 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.

4.2.1 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 $\mathcal{N} \coloneqq \langle S, A, T, R, \gamma \rangle$ is *equivalent* to an extended MDP $\mathcal{M} \coloneqq \langle S', A, T', R', \gamma \rangle$ 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 $\rho(\langle s_0, \ldots, s_n \rangle) \coloneqq \rho'(s'_n)$ 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 that allows to correlate the solutions between the two classes of problems 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 $v_\rho(s) = v_{\rho'}(\sigma(s))$.* [1]

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 that apply to 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 $\sigma : S \to S'$.

LDL$_f$ rewards

We will now define a specific MDP expansion for NMRDPs with LDL$_f$ rewards. In fact, if the rewards are specified through LDL$_f$ or LTL$_f$, it is possible to create extended MDPs that are very compact. We recall that a NMRDP with LDL$_f$ rewards is a tuple $\mathcal{N} \coloneqq \langle S, A, T, \{(\varphi_i, r_i)_{i=1}^m\}, \gamma \rangle$, where $S \coloneqq 2^{\mathcal{F}}$ is a set of propositional interpretations and φ_i are LDL$_f$ formulae on the set of fluents $\mathcal{F}$.

First, using the methods presented in Section 2.3.2, we transform each reward formula φ_i to its associated minimal DFA, $\mathcal{A}_i \coloneqq \langle 2^{\mathcal{F}}, Q_i, q_{i0}, \delta_i, F_i \rangle$. Then, we state the following:

Definition 10. [4] Given an NMRDP with LDL$_f$ rewards $\mathcal{N} = \langle S, A, T, \{(\varphi_i, r_i)_{i=1}^m\}, \gamma \rangle$, we define the equivalent extended MDP $\mathcal{M} \coloneqq \langle S', A', T', R', \gamma \rangle$, where:

- $S' \coloneqq Q_1 \times \cdots \times Q_m \times S$ is the set of states
- $A' \coloneqq A$
- $T' : S' \times A' \times S' \to [0, 1]$ is defined as:

$$T'((q_1, \ldots, q_m, s), a, (q'_1, \ldots, q'_m, s')) \coloneqq \begin{cases} T(s, a, s') & \text{if } \forall i : \delta_i(q_i, s') = q'_i \\ 0 & \text{otherwise} \end{cases}$$

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

- $R' : S' \to \mathbb{R}$ is defined as[2]:

$$R((q_1, \ldots, q_m, s)) \coloneqq \sum_{i \,:\, q_i \in F_i} r_i$$

As we can see from this definition, the extended MDP augments the original model with all the automata $\mathcal{A}_i$ corresponding to the m temporal goals. After each observation, both the original system and every component $\mathcal{A}_i$ are advanced accordingly, in parallel. The reward function, which is now Markovian, can produce the same rewards as in the original formulation (see equation (4.4)) because all the necessary information has been included in the state space.

Theorem 5. *[4] The NMRDP with LDL$_f$ rewards $\mathcal{N} = \langle S, A, T, \{(\varphi_i, r_i)_{i=1}^m\}, \gamma \rangle$ is equivalent to the MDP $\mathcal{M}$ of Definition 10.*

The last theorem states that our construction creates an equivalent MDP, according to the Definition 8. Any NMRDP can be formulated as an MDP, if enough history is included in the state space. So, what is really interesting about this translation is that the expanded MDP has a minimal state space. This is possible because the current state of the automaton $\mathcal{A}_i$ is a sufficient information that retains just enough history to render the rewards r_i Markovian. We have:

Theorem 6. *[4] If every automaton $\mathcal{A}_i$ $(1 \leq i \leq m)$ is minimal, then the extended MDP of Definition 10 is minimal.*

To recap, in this section, we've shown how to train an agent on a Non-Markovian Reward Decision Process, by applying classic RL algorithms on the equivalent MDP. Once the relevant fluents have been selected, we need to express our goal as LDL$_f$ conditions that are associated to a positive reward (or, maybe, conditions for negative rewards). We will see some practical examples in the following section, where we study how to deal with multiple representations of the same configuration of the environment.

4.2.2 RL with LDL$_f$ restraining specifications

Multiple representations

The solution for NMRDPs that we've seen in the previous section is elegant and effective. However, at first sight, it may only seem applicable in very simple state spaces, that are composed of Boolean valuations for sets of fluents (for example, at some

[2]There is a slight difference with the original definition in [4], which accounts for a small notation difference in some previous definitions: $R(s_t)$ is assumed to produce r_t, not r_{t+1}.

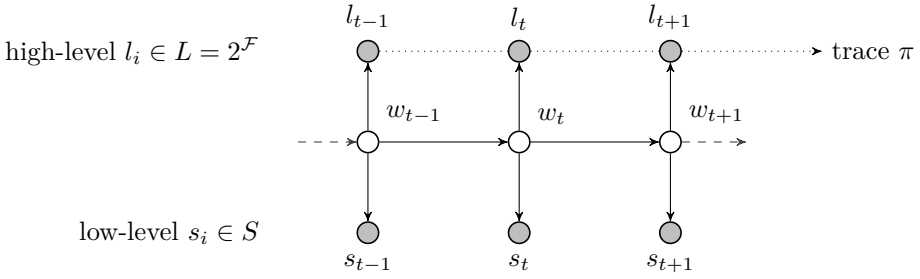


Figure 4.3. Every world state generates both high-level and low-level configurations.

time t, we might have a state s_t in which: $\{\texttt{HaveKey} = \mathit{True}, \texttt{DoorClosed} = \mathit{False}\}$). This is not true, because we must remember that the NMRDP is just a model that we've defined. We're free to adopt a new formalism, where the sequence of observations produced by the environment is decoupled from the trace where our formulae are interpreted on. The ideas presented here have been developed in [6].

Let's denote with W the set of world states. This is an abstract representation of the environment configuration that is inaccessible to the agent. Instead, it receives observations that directly depend on these states. We can represent this sensory input with a function $f_S : W \to S$. Frequently, S is a multidimensional space, so the observations $s \in S$ are also called *features vectors*, or simply *features*. Assuming that these features are the state space of a Markov Decision Process, we can apply RL on S.

We now assume that there is a second function $f_L : W \to L$, with $L \coloneqq 2^{\mathcal{F}}$, that given a world state, assigns a truth value to all fluents in $\mathcal{F}$. This creates two representations with different roles: S is a *low-level* features space that can be complex, noisy and difficult to interpret directly; L is a *high-level* logic representation of the same world states. To any configuration $w \in W$ corresponds a pair of the representations $s \in S$ and $l \in L$. See Figure 4.3.

This distinction is powerful: it allows us to declare temporally-extended goals with LDL_f on the set of fluents $\mathcal{F}$, while the agent receives and works with a different set of features. We now formally define a specific problem that is possible thanks to this distinction.

Restraining Specifications

Consider a Reinforcement Learning agent on the MDP $\mathcal{M} \coloneqq \langle S, A, T, R \rangle$[3]. This already defines the environment dynamics and the agent's optimal policy. We now want to modify the agent's behaviour by declaring an additional temporally-extended

[3]The discount factor has been omitted in this section, because it doesn't apply to the problems we study here. So, it may be simply regarded as tunable parameter.

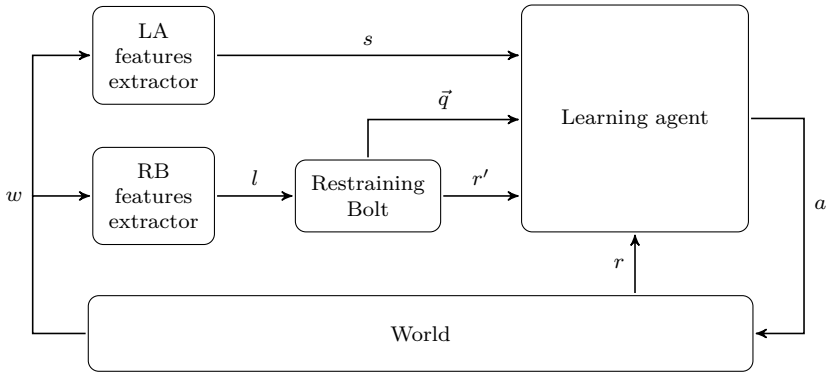


Figure 4.4. Learning agent with Restraining Bolt applied. r is the classic MDP reward; r' is the additional non-Markovian reward generated.

goal on some fluents $\mathcal{F}$. The purpose is to train an agent that pursues the original rewards, while complying with the additional specification we provided. As we know from Section 4.1.2, the LDL$_f$ goals are just a clever way of declaring a non-Markovian reward function. These will be summed with the original rewards, so that the agent will try to pursue both[4]. We call this additional module, which reads the current fluents' configuration and sends the non-Markovian reward back, as the "Restraining Bolt" [6]. This term, borrowed from Science Fiction, suggests that with this additional construction, we're able to modify the "natural" agent's behaviour. In this context, the LDL$_f$ goals $\{(\varphi_i, r'_i)_{i=1}^m\}$ are referred to as "restraining specifications". The general setup is presented in Figure 4.4. Notice, in particular, that the learning agent has access to the original observations s and rewards r, and the additional non-Markovian rewards r'. The quantity $\vec{q}$ will be discussed shortly. Let's now formalize this problem.

Definition 11. [6] A *RL problem with LDL$_f$ restraining specifications* is a pair $\langle \mathcal{M}, RB \rangle$, where: $\mathcal{M} \coloneqq \langle S, A, T, R \rangle$ represents a learning agent, and $RB \coloneqq \langle L, \{(\varphi_i, r'_i)_{i=1}^m\} \rangle$ is a Restraining Bolt formed by a set of LDL$_f$ formulae φ_i over $\mathcal{F}$ with associated rewards r'_i.

We can't simply apply a RL algorithm on the rewards r_i, r'_i over the state space S, because r'_i are non-Markovian in S. What we can do, instead, is to formulate a NMRDP with LDL$_f$ rewards, that we already know how to solve. The complete proof is shown in [6] and [8]. What we see here is a shorter explanation that just highlights the main concepts.

We first observe that, in Definition 11, the MDP and the Restraining Bolt are completely distinct; their only interaction is in the sum of the rewards they

[4]The agent can behave optimally with respect to this combination, but this doesn't necessarily mean that this is the policy we were looking for. Finding the appropriate combination of rewards is a general issue in RL.

produce (let's denote with $\bar{r}_i \coloneqq r_i + r'_i$ the combined reward). So, to simplify the computation, we may keep these two problems separate, transform the restraining specifications to their equivalent MDP and combine them later. This is possible because, given two MDPs, $\mathcal{M}_a = \langle S_a, A, T_a, R_a \rangle$ and $\mathcal{M}_b = \langle S_b, A, T_b, R_b \rangle$, the following $\mathcal{M}_{ab} \coloneqq \langle S_{ab}, A, T_{ab}, R_{ab} \rangle$, with states $S_{ab} \coloneqq S_a \times S_b$, transition function $T_{ab} : S_{ab} \times A \times S_{ab} \to \mathbb{R}$ and rewards

$$R_{ab}((s_a, s_b), a, (s'_a, s'_b)) \coloneqq R_a(s_a, a, s'_a) + R_b(s_b, a, s'_b)$$

is still an MDP. Note that we didn't define T_{ab}. This is not required, as in RL, it is sufficient that this unknown function exists; and by the laws of probability, this is certainly the case, because the existence of T_a and T_b is a stronger requirement.

Every Restraining Bolt $RB = \langle L, \{(\varphi_i, r'_i)_{i=1}^m\} \rangle$ defines a NMRDP with LDL_f rewards $\mathcal{N}_{RB} \coloneqq \langle L, A, T_L, \{(\varphi_i, r'_i)_{i=1}^m\} \rangle$, with states $L = 2^{\mathcal{F}}$ and T_L as the unknown transition function over fluents configurations. This is a problem that we already know how to solve. By directly applying Definition 10, we can write the extended MDP $\mathcal{M}_{RB} \coloneqq \langle S_{rb}, A, T_{rb}, R_{rb} \rangle$ that is equivalent to the NMRDP $\mathcal{N}_{RB}$. Notice in particular, that the state space becomes: $S_{rb} \coloneqq Q_1 \times \cdots \times Q_m \times L$, where each Q_i is the set of states of the i-th automaton. For brevity, we will denote elements of $Q_1 \times \cdots \times Q_m$ with $\vec{q}$, because they are vectors of automaton states.

We can now combine the original MDP $\mathcal{M}$ with the one generated from the Restraining Bolt $\mathcal{M}_{RB}$, just like we've done for $\mathcal{M}_{ab}$, to obtain a new unified MDP that is defined as $\mathcal{M}' \coloneqq \langle S', A, T', R' \rangle$, where:

- $S' \coloneqq Q_1 \times \cdots \times Q_m \times L \times S$
- $T' : S' \times A \times S' \to \mathbb{R}$ with:

$$T'((q_1, \ldots, q_m, l, s), a, (q'_1, \ldots, q'_m, l', s')) \coloneqq \begin{cases} T_{l,s}((l, s), a, (l', s')) & \text{if } \forall i : \delta_i(q_i, l') = q'_i \\ 0 & \text{otherwise} \end{cases}$$

- $R' : S' \times A \times S' \to \mathbb{R}$ with:

$$R'((q_1, \ldots, q_m, l, s), a, (q'_1, \ldots, q'_m, l', s')) \coloneqq \sum_{i \,:\, q'_i \in F_i} r'_i + R(s, a, s')$$

Both $T_{l,s}$, that is the joint transition function of the symbols s and l, and the original reward function, R, are unknown: we only observe the samples produced while the agent plays. Instead, we have to move all automata and return the associated rewards, because this is a dynamics we've defined.

To this point, we've reduced the original problem of Definition 11 to standard RL on the MDP $\mathcal{M}'$. However, we can move one step further. In fact, the combined rewards $\bar{r}_i$ do not depend on the fluents configurations $l_i \in L$, if both s_i and $\vec{q}_i$ are given, that is:

$$\bar{r}_t \perp l_0, \dots, l_t \mid \vec{q}_t, s_t \qquad \text{for any } t$$

This means that an optimal policy exists for $\mathcal{M}'$ with the form: $\rho_* : Q_1 \times \cdots \times Q_m \times S \to A$. To prove it formally, we would need to show that the value of any state $(\vec{q}, l, s)$, defined in equation (3.2), does not depend on l. We finally get to the following result:

Theorem 7. *[6] RL with LDL_f restraining specifications $\langle \mathcal{M}, RB \rangle$, with $\mathcal{M} = \langle S, A, T, R \rangle$ and $RB = \langle L, \{(\varphi_i, r'_i)_{i=1}^m\} \rangle$, can be reduced to RL over the MDP $\mathcal{M}'' \coloneqq \langle Q_1 \times \cdots \times Q_m \times S, A, T'', R'' \rangle$, and optimal policies for $\langle \mathcal{M}, RB \rangle$ can be learnt by learning corresponding optimal policies for $\mathcal{M}''$.*

If we denote with S'' the state space $Q_1 \times \cdots \times Q_m \times S$, the functions T'' and R'' are partially unknown functions $S'' \times A \times S'' \to R$, that are defined respectively as T' and R', marginalized with respect to L. With the MDP $\mathcal{M}''$, we assumed that at each instant t, we observe the current state $(\vec{q}_t, s_t)$. This is true for s_t, but not for $\vec{q}_t$. What we can do instead, is receiving an observation of the symbols l_t, moving all the automata accordingly, and collecting the resulting states $\vec{q}_t$. So, the new state $(\vec{q}_t, s_t)$ is composed of the computed vector of automata states, and the observed symbols s_t. The symbols L don't need to be passed to the learning agent. Refer to Figure 4.4, once again.

Example 9. Let's extend Example 8 on page 31. An agent can control a light bulb through a switch, but now it can only observe images of the room. Suppose we want to reward the agent with the same condition of the previous example: the light must have been switched on, then off, only once during the episode. To do so, we can define a single fluent, $\mathcal{F} \coloneqq \{\mathit{LightOn}\}$, representing the status of the light. The states S are the low-level symbols, which in this case are images. L, the high-level symbols, are composed just by the valuation for $\mathit{LightOn}$, which we can compute from S (it is true, when the light is on in the image). Now we state the following LDL_f restraining specification:

$$\varphi_1 \coloneqq \langle\, (\neg \mathit{LightOn})^*; \mathit{LightOn}^+; (\neg \mathit{LightOn})^+ \,\rangle \mathit{End}$$

where ρ^+ is an abbreviation of $\rho; \rho^*$. The DFA that is associated to this formula is shown in Figure 4.5. The MDP associated to this simple problem is: $\langle Q \times \Omega, A, T, \{(\varphi_1, r')\} \rangle$, where the actions are $A \coloneqq \{\mathit{Toggle}, \mathit{NoOp}\}$, Ω is an image, and $Q \coloneqq \{0, 1, 2, 3\}$. With this composite state space, the agent has a complete view

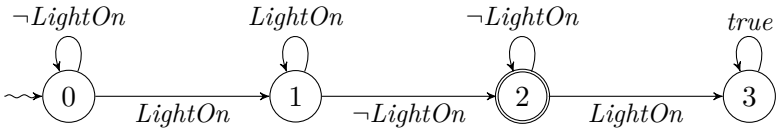


Figure 4.5. The DFA associated to the formula φ_1, in Example 9.

about the missing steps that are needed, in order to achieve the reward; something that it wouldn't know just from the current image or the current value of *LightOn*.

Since, in this extremely simple example, we didn't define any environment reward, this mostly resemble a problem of RL with LDL_f reward. However, the setup with the two groups of symbols, S and $\mathcal{F}$, are more precisely modelled with a RB. In this case, *LightOn* is enough, but usually rewards will also depend from S.

4.2.3 Restraining Bolt for partial observations

We've thoroughly analyzed solutions for the non-Markovian goals of Section 4.1.2: namely, temporally-extended goals. We've solved them both in isolation, and as additional restraining specifications in preexisting MDPs. We now want to ask: is it possible to address partial observations, which is the second source of non-Markovian rewards, in a similar way? As we'll now discuss, for a class of these problems, the answer is yes. Although this may not be practical for all of them.

Partially observable stochastic environments are properly modelled with POMDPs $\mathcal{M}_p = \langle S, A, T, R, \Omega, O \rangle$, because they are able to correctly distinguish between the world state space S and the observations Ω that the agent receives. POMDPs are a large class of problems with a greater complexity with respect to MDPs. As we've discussed in Section 4.1.1, we can't simply apply a RL algorithm from the visible quantities. The classic learning method for POMDPs is to transform them to MDPs over the belief-state for S. However, since we assume that transition and observation functions can be unknown, we won't be able to apply classic estimation algorithms. In this section, we'll show that a class of POMDPs can be learnt by performing RL on MDPs, defined over a extended, but compact, state space. This idea is very similar to that of the previous sections, but we're facing a different class of problems.

In this text, we'll only consider POMDPs whose set of states, S, is finite. These can be represented with a factored state space, composed of assignments for a finite set of propositions $\mathcal{P}$.

Definition 12. A POMDP is a tuple $\mathcal{M}_p \coloneqq \langle \mathcal{P}, S, A, T, R, O, \mathcal{P}_O \rangle$, where, A, T and R are defined as in MDPs, $\mathcal{P}$ is a finite set of propositions, the states $S = 2^{\mathcal{P}}$ are the assignments for the variables $\mathcal{P}$, and $\mathcal{P}_O \subset \mathcal{P}$ is the subset of observable propositions. The observation space is implicitly defined as the set of assignments for $\mathcal{P}_O$.

First we state a general result for POMDPs: learning over a POMDP can be equivalently formulated as a learning problem over a non-Markovian Decision Process. The latter model is identical to an NMRDP, except that, in addition to rewards, also the transition function can depend on the entire history of configurations.

Definition 13. A Non-Markovian Decision Process (NMDP) is a tuple $\mathcal{M}_n \coloneqq \langle \mathcal{P}, A, S, T, R \rangle$, where:

- $\mathcal{P}$ is a finite set of propositions.
- $S = 2^{\mathcal{P}}$ is the set of assignments for the variables $\mathcal{P}$.
- A is the discrete set of actions.
- $T : S^+ \times A \times S \to \mathbb{R}$ is the transition function that maps a non-empty sequence of states (a trace) and the current action, to the probability of the next state.
- $R : S^+ \to \mathbb{R}$ is the reward function.

So, it is possible to state the following result:

Lemma 1. *[3] For every POMDP, $\mathcal{M}_p \coloneqq \langle \mathcal{P}, S, A, T, R, O, \mathcal{P}_O \rangle$, there exists a NMDP as $\mathcal{M}_n \coloneqq \langle \mathcal{P}_O, A, 2^{\mathcal{P}_O}, T', R' \rangle$, where T' and R' are defined based on T, R, and $\mathcal{P}_O$, such that $\mathcal{M}_p$ and $\mathcal{M}_n$ are equivalent*[5].

In every POMDP, it is possible to compute the optimal action from the entire history of observations. The lemma above is the first step in order to reduce the minimal informations about the past that the agent must take into consideration. In fact, for Boolean features, we can assume some regularities on the functions T and R, which will allows us to define a compact representation of the past trace. In particular, we assume that all the POMDP states and rewards can be generated from a finite state transducer (a machine that operates on regular languages in input). Let's formalize this model more precisely.

Definition 14. We define a *Regular Decision Process* (RDP) to be a NMDP $\mathcal{M}_r \coloneqq \langle \mathcal{P}, A, S, T, R \rangle$, with functions T and R that are generated by a finite state transition transducer $\mathcal{T} = \langle Q, q_0, \delta, \varrho, \eta \rangle$, where:

- Q is the finite set of transducer states.
- $q_0 \in Q$ is the initial state.

[5] More precisely, we mean that the two models produce computation trees which are isomorphic. However, we won't formally define isomorphism and the computation trees of the respective models.

- $\delta : Q \times S \to Q$ is the deterministic transition function of the transducer; we extend δ to sequences in the natural manner: $\delta(q, \langle s_1, s_2 \rangle) = \delta(\delta(q, s_1), s_2)$.
- $\varrho : Q \times A \times S \to \mathbb{R}$ is the transition output function of the transducer, such that: $T(\langle s_1, \ldots, s_k \rangle, a, s') = \varrho(\delta(q_0, \langle s_1, \ldots, s_k \rangle), a, s')$.
- $\eta : Q \to \mathbb{R}$ is the reward output function of the transducer, such that: $R(\langle s_1, \ldots, s_k \rangle) = \eta(\delta(q_0, \langle s_1, \ldots, s_k \rangle))$.

The RDP model has been introduced in [3], where the authors have defined T and R using formulae of LDL_f. The definition we see here, doesn't specify the formalism in which these functions are defined, it only expresses the general properties they need to respect: it must be possible to generate states and rewards from a finite state transition transducer. Since LDL_f exactly captures the class of regular properties on the trace S^+, this definition and the original, in [3], are completely equivalent.

RDPs are a more restricted class of models with respect to NMDPs: not all POMDPs can be expressed as RDPs. We can more precisely state for which of those it is possible to do so. Closely following [3], we obtain:

Theorem 8. *For every POMDP, $\mathcal{M}_p \coloneqq \langle \mathcal{P}, S, A, T, R, O, \mathcal{P}_O \rangle$, if, in all reachable states, the value of every $p \in \mathcal{P}$ is a regular function of the past and current values of $\mathcal{P}_O$ (i.e. p is true if some RE_f is satisfied), there exists an equivalent RDP $\mathcal{M}_r \coloneqq \langle \mathcal{P}_O, A, 2^{\mathcal{P}_O}, T', R' \rangle$, for some transducer $\mathcal{T}$.*

We've shown that a class of POMDPs can be translated to equivalent RDPs. This is important, because, as we can see from Definition 14, an RDP doesn't have any direct dependency from the past observations. Instead, all the necessary informations needed to predict the next states and rewards are contained in the transducer state Q. This allows us to forget about complete trace of observations $2^{\mathcal{P}_O}$, which may be any high-dimensional space.

The final result that we need is that RDPs can be transformed to MDPs. Very similarly to Lemma 1 of [3], we can state the following:

Theorem 9. *Let $\mathcal{M}_r$ be any RDP $\mathcal{M}_r \coloneqq \langle \mathcal{P}, A, S, T, R \rangle$ and $\mathcal{T}$ its finite state transition transducer $\mathcal{T} = \langle Q, q_0, \delta, \varrho, \eta \rangle$. Then, there exists an MDP $\mathcal{M} \coloneqq \langle Q, A, T', \eta \rangle$, that is equivalent to $\mathcal{M}_r$, whose transition function is defined as $T'(q, a, q') \coloneqq \sum_{s' \in S \,:\, \delta(q, s') = q'} \varrho(q, a, s')$.*

We've now reached the desired result. By combining Theorems 8 and 9 we obtain:

Corollary 1. *For any POMDP $\langle \mathcal{P}, S, A, T, R, O, \mathcal{P}_O \rangle$, in which, in all reachable states, the value of every $p \in \mathcal{P}$ is a regular function of past and current values of $\mathcal{P}_O$, there exists an MDP $\mathcal{M}$ defined as in Theorem 9.*

This corollary states *when* a POMDP admits an equivalent MDP with a finite and countable set of states Q. This is a general result, but it doesn't immediately tells us *how* to define such equivalent MDP. Since, in RL, we assume the transition and reward functions of the original model to be unknown, we can't start from the POMDP model to write the appropriate finite state transducer of the RDP. Furthermore, if the set of variables $\mathcal{P}$ is very large, we'll be also unable to explicitly define the equivalent MDP, even if we had a complete knowledge of the environment. Still, this general result states that such MDP indeed exists, for the group of POMDPs specified.

However, in the small remainder of this section, we'd like to discuss how some of these partially observed environments could be addressed with the Restraining Bolt. Creating a formal connection between the two solution approaches is between the scope of this thesis. Still, we'd like to draw some observations here.

Suppose we're given a partially observable environment, which produces bounded rewards $r \in \mathbb{R}$ and observations $o \in \Omega$. When Ω is a complex high-dimensional space it's often not practical to recognize regular properties in the trace Ω^+. Instead, we introduce a new set $\mathcal{F}$ of atomic propositions which are computed from Ω. So, we define $\mathcal{F}$ such that there exists an appropriate features extractor $f : \Omega \to 2^{\mathcal{F}}$. Can we assume that the value of every POMDP variable $p \in \mathcal{P}$ is a regular function of $\mathcal{F}^+$? Usually, not. Since we won't be able to define an exhaustive set of variables.

However, let's suppose for a moment the our goal is to *imitate* the rewards generated by the environment. To do so, we specify an MDP $\mathcal{M} = \langle \Omega, A, T', R' \rangle$ and a set of LDL_f restraining specifications, such that the rewards generated by this Restraining Bolt reproduce those generated by the environment. Let's denote with $\mathcal{M}'$ the equivalent MDP associated to the Restraining bolt just defined. Since the rewards generated satisfy the Markov property with respect to the states of $\mathcal{M}'$, we argue that the same holds for those generated by the environment. Essentially, there could exists an appropriate set of LDL_f formulae, such that the additional components associated to the Restraining Bolt's states, restore the Markov property on rewards.

A Restraining Bolt is just a clever way to define a Markovian state with respect to some LDL_f rewards. It can't be used to generate the full POMDP dynamics, like we did with the transducer. This idea serves just to suggest some parallelisms between the two techniques, and indicates future research directions.

4.3 Restrained Deep RL agents

As illustrated in Section 3.2.2, the RL algorithm used in this thesis is Double DQN, a Deep RL method. Double DQN agents contain a Q-Network, which is a function $Q : S \to \mathbb{R}^{|A|}$, that is parametrized in θ. Given a MDP state $s \in S$, this computes the state-action value for every action $a \in A$. In this section, we will propose an original Q-Network model that properly handles the additional inputs received from the Restraining Bolt.

The Restraining Bolt is an interesting method because we can regard it as a module that, added to the original setup, generates additional rewards and observations. In principle, no structural modifications would be required in the agent's design: new rewards can be simply summed with the previous ones, and the Restraining Bolt's states $\vec{q}$ can be stacked with the environment's observations to produce a composite MDP state $(o_t, \vec{q_t})$. Agents can learn from this new state space without modifications[6].

Deep RL agents, instead, learn approximate value functions and policies. In this case, we should carefully select the agent's model, a neural network, that has the appropriate expressive power. A network that is suitable to approximate a function $\Omega \to \mathbb{R}^{|A|}$ is not necessarily appropriate for a function from the new state space, $\Omega \times Q_1 \times \cdots \times Q_m$. Overfitting and underfitting are well known issues in Machine Learning.

4.3.1 Q-Network for the Atari games

We will first illustrate the agent's model that we use in this thesis when the Bolt is not applied. Then, in Section 4.3.2, we'll propose a modification of this network that accounts for the additional states of the Restraining Bolt. The environments used in this thesis are games from the collection "Atari 2600". As illustrated in Section 3.2.1, the observations produced are frames of size $(210, 160)$, with an RGB colour depth of 8-bit. Each game defines a different number of actions, 18 at most.

We use the same architecture as [19], which is illustrated here. Slight modifications will be listed in the implementation part, in Section 6.2.2. First, a preprocessing function applies a fixed transformation to each image. Every frame is converted to a gray-scale picture, by computing the luminance value of each pixel. The image is then resized to $(84, 84)$, in order to reduce the input dimensionality. Finally, 4 consecutive images are combined together, producing a tensor of size $(84, 84, 4)$ that can be passed to the network. This last combination allows to create an observation

[6] Extending the size of a table, in order to account for the additional number of states, is not considered a real modification in the agent's design.

that encodes how the objects in the scene are moving. The lack of this information is one of the causes of non-Markovian rewards that can be easily solved. See Example 7, for an explanation.

Let's define the following abbreviation: Conv(n, s, t) represents a 2D convolutional layer composed of a number of n filters of size $s \times s$ with a stride of t. Similarly, Dense(n) represents a fully-connected layer of n units. We can now define the network structure as:

$$\begin{gathered}\text{Conv}(32,\ 8,\ 4),\ \text{ReLU},\\ \text{Conv}(64,\ 4,\ 2),\ \text{ReLU},\\ \text{Conv}(64,\ 3,\ 1),\ \text{ReLU},\\ \text{Dense}(512),\ \text{ReLU},\\ \text{Dense}(|A|)\end{gathered}$$

where ReLU is the rectifier linear unit applied to each element. In neural networks, images are frequently transformed with a cascade of 2D convolutions, followed by a number of dense layers. The authors of the original paper [19] have shown that this network size generates a model with the appropriate expressive power for our environments.

4.3.2 Q-Network for the Restraining Bolt

We now want to apply the method presented in Section 4.2.3 in those games in which low performances are caused by partial observations. This means that our agent would need to receive the original observation, which is a frame of the game, and the vector of the Bolt's states, $\vec{q}$. We'll now suppose that our temporally-extended goal can be expressed with a single pair (φ, r'). So, the Restraining Bolt's state is a single scalar identifier q.

As anticipated, we cannot simply stack o and q. Even if the network architecture would allow that, we would assign a very low relative importance to q among the thousands of pixels of which o is composed. Most importantly, the role of q must not be confused with pixels. All these considerations are important because every model introduces some biases, and we want our model's bias to capture the following basic intuition: q is an important index that parametrizes value functions. The state q is a parametrization over value functions because, to different automaton states, there may correspond dramatically different value functions over inputs.

Example 10. Let's consider again the light bulb of Example 9 and the automaton of Figure 4.5. Suppose the initial MDP state is $s_0 = (o_0, 0)$, where o_0 is an image of a dark room. In this case, the model should learn an high state-action value for the action *Toggle*, and a low value for the action *NoOp*. Later on, at some time t, the

agent may observe the following input: $s_t = (o_0, 2)$. Even though the image is the same, the agent should assign the highest value to *NoOp*. A different automaton state dramatically changes the most promising actions that will lead to the goal.

A very drastic choice would be to maintain a number of $|Q|$ different networks, with the same architecture but different parameters $\theta_1, \ldots, \theta_{|Q|}$. At each step, given an input (o, q) the agent may use the network θ_q to predict the actions values for the input o. This strong parametrization would completely separate the value functions. An immediate problem with this approach is space inefficiency (that would be very evident with large Q or vectorial $\vec{q}$). Most importantly, the networks associated to states that are rarely encountered would be trained on too few input samples.

The model that we propose here is a variant of the network of the previous section. We substitute the last fully-connected layer with one of dimension $|A| \times |Q|$. This means that a number of $|A| \cdot |Q|$ linear units is arranged as a matrix, whose first index is an action and the second index is a Bolt's state. So, for each automaton state, the net will generate a different column of state-action values. The idea behind this choice is that we can safely share the initial layers, whose main goal is to provide an encoding of the observed input. Instead, separating the last layer provides the greatest flexibility among other combinations[7]. This is an intermediate approach between completely shared and completely separate parameters $\theta_1, \ldots, \theta_{|Q|}$. For a vectorial $\vec{q}$, it can be easily extended: the last fully-connected layer would produce tensors of shape $(|A| \times |Q_1| \times \cdots \times |Q_m|)$. Since the greatest number of parameters is shared, each combination of automaton states $\vec{q}$ requires a smaller number of training samples to train on.

The resulting Q-Network for the Atari games is:

$$\begin{gathered}\text{Conv(32, 8, 4), ReLU,}\\ \text{Conv(64, 4, 2), ReLU,}\\ \text{Conv(64, 3, 1), ReLU,}\\ \text{Dense(512), ReLU,}\\ \text{Dense}(|A| \times |Q|),\\ \text{Slice}(\cdot, q)\end{gathered}$$

where $\text{Slice}(\cdot, q)$ indicates that we select the q-th column of the input matrix. This is the agent's model used in this thesis. As we can see, we didn't need to define more than one temporal goal in our experiments.

Some other variants may exists. In fact, we should remember that the automata states are generated from the conversion of LDL_f or LTL_f expressions. Since, this

[7] We didn't motivate why the first layers of the original net should behave as an encoder. However, after the modification, they will be shared and trained with different outputs. So, this role will be encouraged.

translation has a worst case complexity that is doubly exponential in the size of the formula, the state space may be quite large. One possibility would be to investigate whether is it possible to adopt the NFA states, instead of the DFA's, producing a state space that may be exponentially smaller (multiple columns would be active at the same time, in this case). But these variants have not been investigated yet.

Chapter 5

Learning to ground symbols trough RL

In Chapter 3, we've presented how to apply Deep Reinforcement Learning to achieve non-Markovian goals. We've seen that a construction based on temporal logics, that we call the Restraining Bolt, is an elegant solution that transforms the original problem to a classic MDP, by producing additional rewards and observations. We report the general scheme here, in Figure 5.1. The two blocks at the bottom are the additions, and part of the solution. We've thoroughly addressed the Restraining Bolt in Section 4.2.2, already. In this chapter, we want to focus on the other essential component: the features extractor.

The purpose of the features extractor is to receive an observation from the environment and produce a Boolean valuation for some predefined propositional symbols, that we call fluents. The problem of creating a sensible connection between the true state of the outside world and the agent's atomic propositions is called *grounding*. This process is essential, because it ensures that any decision that is appropriate for the agent's abstract representation is also appropriate in the real scenario.

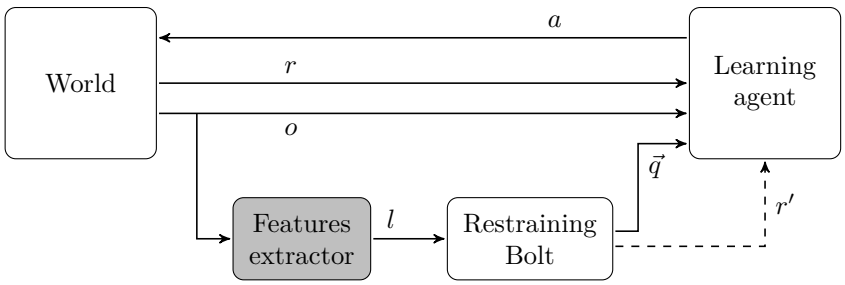


Figure 5.1. In this chapter, we focus on the features extractor.

We assume that the set of fluents $\mathcal{F}$ has already been defined, and the truth of every atomic proposition in $\mathcal{F}$ can be decided from a single input o. If we did define a symbol that cannot be decided from one observation, for example a generic "*GoalReached*" condition, we need to split that symbol into much simpler events, and define *GoalReached* in terms of the new fluents.

Usually, the features extractor is not a really interesting component. Once, we've defined a fluent $p \in \mathcal{F}$, we could manually program a function, $f_p : \Omega \to \mathbb{B}$, that predicts when that event occurs or that condition is verified[1]. This approach is perfectly fine, when applicable. However, the environments used in Deep Reinforcement Learning usually produce high-dimensional or noisy observations. As we may imagine, it becomes really hard to manually classify such inputs in the two classes. So, in order to apply the Restraining Bolt or any other logic-based method to Deep RL, we must resort to some Machine Learning model that will help us deciding the truth of our atomic propositions.

Any Deep RL agent processes the input observation with a Neural Network. A reasonable choice would be to use a NN also as model for the features extractor. We may use a joint network that predicts the value for every fluent defined: $f : \Omega \to \mathbb{B}^{|\mathcal{F}|}$. The simplest way to train this model is through supervised learning, where we provide many input-output samples. Supervised learning can generate very accurate models, but, for every image in the training dataset, we would need to manually label the desired outputs, i.e. the fluents that are true in that image. Unfortunately, the effort of this manual intervention would completely dominate over the advantages of the high-level, logic approach. If possible, we would certainly like to avoid this manual work.

A very promising alternative is unsupervised learning. These models don't return predictions. Instead, they memorize patterns and features that the training inputs have in common. These models keep two representations: the input space, and the latent space. To any input that is presented to the model corresponds a compact representation in the latent space. The purpose of this representation is to distinguish the specific input among all of the training set[2]. Since the latent vector is much more compact, it can be used in other computations in place of the original input. In this case, the latent vector is called an *encoding*.

Unsupervised learning will be a central part of the solution proposed here. However, it may not be the only part, because unsupervised models makes no guarantees about the meaning of the latent representation. This means that we cannot predict what each number in the latent vector represents. So, the proposed

[1] $\mathbb{B}$ is the set $\{0, 1\}$, which represents $\{\mathit{false}, \mathit{true}\}$.

[2] To emphasize this concept: the latent vector serves to identify the input sample among the training distribution.

model will transform the encodings through a second function, which computes the truth value of the fluents.

To better understand the motivation behind our choices, we list few general goals that we want to pursue with this work:

1. Learning should not require manual labelling.
2. We select the fluents that should be learnt.
3. Model should make as few environment-specific assumptions as possible.

The second principle means that we first choose the propositions to use in our temporal formulae, then we train a model to valuate them. An opposite approach could have been to train an extractor of Boolean features, then trying to recognize the meaning of those fluents.

With the general goals 2 and 3 above, in particular, we express that the model should be generalizable to a wide range of output fluents and input observations. Of course, this is only possible to some extent. As we'll discuss in Section 5.2 and 5.7, this method, as realized in this thesis, makes some assumptions that limit its applicability to a specific class of fluents and observations. However, this work poses some interesting ideas, such as temporal constraints, that certainly needs to be further investigated in the future. This thesis is just an initial study in this direction.

5.1 Temporal constraints

In this section, we illustrate the concept of "temporal constraints". This is an important idea introduced with this work, that will help us pursue the three principles above. We can start from the following observation: a dataset of labelled samples is a description, by examples, of the desired meaning of the fluents. A good model would interpolate between these samples to inputs that have never been observed. Without these examples, how do we specify the desired meaning of a fluent? Note that by "meaning", we mean the set of inputs in which the propositional symbol should be valuated to true.

What we propose here is to specify the desired temporal behaviour of these fluents with temporal logics: we write a temporal formula, in LTL_f or LDL_f, that describes all the possible traces of the fluents we want to define. We don't talk about *desired* trajectories. Instead, we define all the *possible* trajectories according to the environment dynamics. For example, suppose that two conditions A and B, according to their intended meaning, cannot be true at the same time. Regardless of what we're trying to achieve, we can write the following temporal constraint: $[\,\mathit{true}^*\,](\neg A \vee \neg B)$. This is a simple propositional property that should hold in every

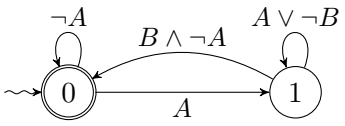


Figure 5.2. The DFA associated to the formula $[\, true^*; A \,]\langle\, true^* \,\rangle B$.

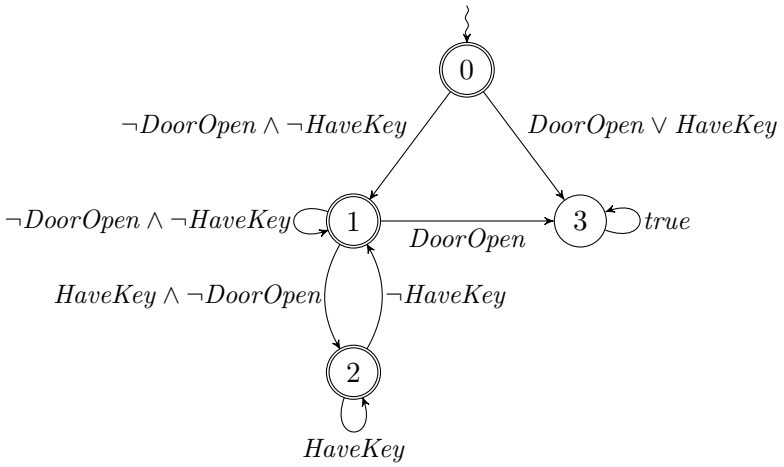


Figure 5.3. The DFA associated to the Example 11.

instant, but there are many other interesting constraints that we may specify with temporal logic. For example, A and $\langle\, true^* \,\rangle(Last \wedge A)$ respectively mean: every episode starts/ends with an instant where A is true. Also, $[\, true^*; A \,]\langle\, true^* \,\rangle B$ means that every time A becomes true, the event B must follow later on. This is a frequent pattern in request–response behaviours. The automaton associated to this constraint is shown in Figure 5.2. Temporal logics like LDL_f are very expressive and allows to write many complex properties that the symbols satisfy.

Usually, we won't be able to write complete constraints or exact definitions of the fluents. This is not necessary, though. It is sufficient to exclude as many inconsistent trajectories as we can, given the symbols available.

Example 11. Suppose that an agent should open a door that is closed with a key, and we've defined the fluents $\mathcal{F} \coloneqq \{HaveKey, DoorOpen\}$. We need to train a feature extractor that valuates these two propositions with their intended interpretation. We may write the following constraint:

$$(\neg HaveKey \wedge \neg DoorOpen) \wedge \neg\langle\, true^*; \neg HaveKey \,\rangle DoorOpen$$

which says that the door cannot be opened if, at the previous instant, we don't have a key. Also, initially, the door is closed and the agent has no key. The associated automaton is shown in Figure 5.3. Note that we didn't specify that the door should be eventually opened. The automaton only excludes the trajectories that certainly

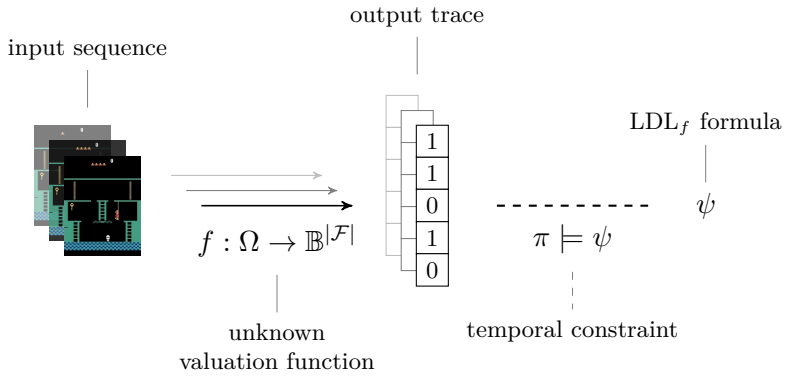


Figure 5.4. The trace generated by the valuation function must satisfy the temporal constraint.

can't happen.

The general idea is shown in Figure 5.4. We're trying to learn the unknown function $f : \Omega \to \mathbb{B}^{|\mathcal{F}|}$, which, from a single input observation, computes the truth value for all fluents. The trace generated must always satisfy the temporal constraint defined in ψ.

We've just described how temporal constraints work. However, there is one important consideration to do: these constraints are very weak. This means that there will be lots of fluents' valuations functions that are consistent with this specification. Consider Example 11, a feature extractor $f(o) \coloneqq \{\}$ completely ignores the input and always predicts that both fluents are false. It is wrong but it's perfectly consistent with the specification. Similarly, many other valuation functions that respect the DFA dynamics of Figure 5.3 will be completely meaningless. This may not surprise us, as most constraints can only relate the valuation functions with each other, but they cannot force arbitrary input–output associations. The initial and final conditions (such as $\neg \mathit{HaveKey} \wedge \neg \mathit{DoorOpen}$ in the Example 11) are some of the few examples that creates a strong binding between input observations and desired fluents' output.

5.2 Assumptions

In this section, we'll list the initial choices and assumptions taken in this work. Of course, assumptions like these restrict the range of valuation functions that can be learnt. However, they are essential in order to devise a solution. The purpose of most of them is to address the issue mentioned in the previous section: temporal constraints are only very weak indications of the desired meaning of the fluents. Other assumptions, instead, describe the range of environments which the proposed

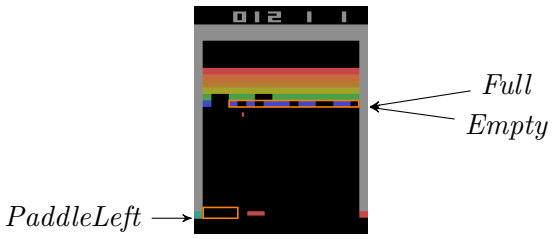


Figure 5.5. Fluents and regions for the environment Breakout in Example 12.

model can be applied to.

First, we remember that the environments we're dealing with are video games from the Atari collection. So, the input space is composed of images of size $(210, 160)$. In the following, we will always use images from this games, because this is how environment's observations look like.

Then, we assume that each atomic proposition can be decided just from a fixed region of the input image. In other words, to each fluent, we associate a rectangular portion of the input and we assume that an observation of this region is sufficient to decide the truth of the symbol. Regions can overlap and different fluents can be defined on the same region. By restricting the input space of the model so much, we're partially reducing the complexity of the problem.

Example 12. In order to better understand what regions are, we anticipate one environment that we'll encounter in the experiments Section 7.1: Breakout. In this famous game, the player's goal is to direct the ball toward the bricks. Suppose we want to learn three fluents $\mathcal{F} \coloneqq \{\mathit{Empty}, \mathit{Full}, \mathit{PaddleLeft}\}$, as shown in Figure 5.5. The region associated to each of these symbols is indicated with an orange box. *PaddleLeft* should be true when the paddle is inside the area on the left; *Empty* and *Full* should be true respectively when all the bricks, or no bricks, inside the region have been shot down.

As it may be clear from this example, the regions also help to do something that would be impossible in logic: indicate to which portion of the input the fluents refer. Also, we should choose each region as a small selection of the image in which the condition we'd like to extract is *visually evident*: to different valuations should correspond noticeable differences in the region. In the following sections, we'll refine what we mean by "noticeable". Essentially, we mean that the model must be able to learn an encoding that captures the properties we're interested in. Unfortunately, this is not something that can be described with extreme precision, in machine learning. We'll discuss this issue in Section 5.4.3.

Due to these fixed regions, this method is only applicable to games with a fixed view of the scene. If every object in the image were moving, we couldn't apply the

simplification of static regions. Few examples of games that respect this constraint are: Breakout, Mr. Pac-Man, Video Pinball, Pong. We'll also experiment with Montezuma's Revenge, but we'll limit to the first room.

In Section 5.7, "Limitations and improvements", we'll suggest few ideas about how many of these limitations might be relaxed.

5.3 General structure of the model

In this section, we introduce the general structure of the model that we propose in this thesis. The remaining parts of the chapter will cover all the details and definitions.

As we already know, temporal constraints can exclude many inconsistent assignments but they are only weak indications about the desired valuation functions. By assuming regions, we've strongly reduced the size of the input space, hence of the possible functions. Still, the problem is only mitigated. To address this issue, we propose to learn each valuation function from an encoding vector that represents the region of interest, rather than from the pixel values directly. An encoding is a low-dimensional vector that represents a more complex input. So, we might use this vector in place of the original input in order to simplify the learning process of the valuation function.

An encoding can be viewed as a lossy compression of the input. The meaning of the encoding vector depends on the specific model, which will be presented in Section 5.4. Here, we only want to highlight that all input images that are considered similar according to the model will correspond to the same encoding vector. Hence, these images will produce the same interpretation for the fluents. However, this is certainly a desirable effect: the encoder can extract few relevant indicators from which we can compute the fluents' values, while noises and tiny variations of the input will be ignored.

So, the model that we propose is a function composed of two consecutive parts: an array of *encoders* and an array of *Boolean functions*. This scheme is illustrated in Figure 5.6. Let's assume that the set $\mathcal{F}$ of fluents to learn is given, along with their associated regions. We create an encoder for each region and a Boolean function for each symbol in $\mathcal{F}$. There might be less encoders than Boolean functions, because multiple fluents can share the same region. This scheme has two encoders and three Boolean functions (just like the model associated to the example in Figure 5.5). Of course, that is just a specific instance. There will be as many parts as are needed.

To summarize:

1. The input frame is cropped around each region;

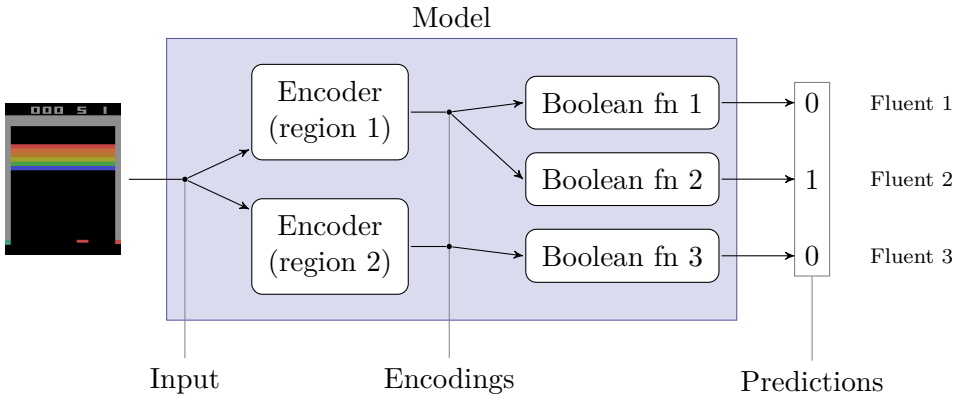


Figure 5.6. The model is composed by a sequence of encoders and a set of Boolean functions.

2. Each small image is transformed with its own encoder;
3. Each Boolean function computes the value for one fluent from the corresponding encoding vector.

5.4 Encoding

The encoder is a Machine Learning model that is trained in an unsupervised way, whose purpose is to convert the input of one region to the associated low-dimensional representation. The model we've selected for this role is the Deep Belief Network. As we will see, this choice is mainly motivated by the output produced by this model, which is a vector of binary features. In the following, we'll thoroughly describe this model, and discuss why a binary output is so important.

Since the layers of a Deep Belief Network are a specific type of Markov Random Field, it is better to quickly review these first. This will allow us to better understand the goal and properties of the final encoder.

5.4.1 Markov Random Fields

In the previous chapters, we have sometimes used Directed Graphical Models (for example in Figure 3.2 on page 18). These are probabilistic models in which directed arcs represent known conditional probabilities between two variables. Here, instead, we'll adopt a different kind of formalism: *Undirected Graphical Models* (UGMs), which are also called *Markov Random Fields* (MRFs). Most of the topics presented in this section are a reorganization of the material in [21].

A UGM is an undirected graph, where variables are represented by nodes, as usual, and undirected arcs connect variables which are directly dependent. A

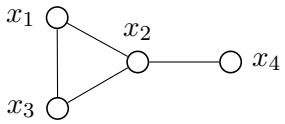


Figure 5.7. A small MRF where: $x_1 \not\perp x_4$, but $x_1 \perp x_4 \mid x_2$.

missing edge means that two variables are conditionally independent given the others. For example, in Figure 5.7, x_1 and x_4 are dependent variables, but they become conditionally independent if x_2 is observed. In this and the following graphs, we assume that each node represents a scalar quantity. Vectorial quantities will be indicated in bold fonts.

MRFs are very convenient when we what to express that two variables are related, but we can't establish any causal relation between them. Consider, for example, the noisy pixels of an image. The values of neighbouring pixels are clearly dependent, as they are related by the subject of the image, but we can't establish any useful conditional probability between the two.

In every MRF, the joint probability of all the variables can be expressed as[3]:

$$p(\mathbf{x} \mid \boldsymbol{\theta}) \propto \prod_{c \in \mathcal{C}} \psi_c(\mathbf{x}_c; \boldsymbol{\theta}_c) \tag{5.1}$$

where each $c \in \mathcal{C}$ is a maximal clique of the graph, and each function ψ_c computes how likely is the observation of variables $\mathbf{x}_c$, according to the parameters $\boldsymbol{\theta}_c$. The joint probability is only proportional to that quantity, because one normalizer has been omitted. As an example, the joint probability for Figure 5.7 can be written:

$$p(\mathbf{x} \mid \boldsymbol{\theta}) \propto \psi_1(x_1, x_2, x_3; \boldsymbol{\theta}_1)\, \psi_2(x_2, x_4; \boldsymbol{\theta}_2)$$

Therefore, a model for a MRF is fully defined by two factors: the structure of the graph, and the functions ψ_c. While the former strongly depends on the process to be modelled, we can present the most common choice for the latter. First, let's rewrite each term as:

$$\psi_c(\mathbf{x}_c; \boldsymbol{\theta}_c) \coloneqq \exp(-E(\mathbf{x}_c; \boldsymbol{\theta}_c)) \tag{5.2}$$

for some function E. E is called the *energy function*. Since it is inversely correlated to the probability, it is high for unlikely configurations of variables and low for very probable configurations (according to the model parameters, of course). Due to this property, the energy is sometimes a very useful indication of the final

[3] In this chapter, vectors are denoted with bold lower-case letters. So, the entries of a vector $\mathbf{x}$ are x_i. Similarly, matrices are indicated with bold upper-case letters. A matrix $\mathbf{A}$ has columns $\mathbf{a}_{:j}$ and entries a_{ij}.

probability[4]. Usually, the energy function is of the simplest kind, a linear combination: $E(\mathbf{x}_c; \boldsymbol{\theta}_c) \coloneqq -\boldsymbol{\theta}_c^T \boldsymbol{\phi}_c(\mathbf{x}_c)$, for some basis function $\boldsymbol{\phi}_c$. In the following, we'll assume this form.

As we may recognize, in a MRF there is no output. We're not trying to learn an input–output function. Training a model means to find the optimal parameters $\boldsymbol{\theta}_*$ that maximize the probability of the input samples. The trained model should represent the probability distribution of the input data, as closely as possible. This is clearly different from supervised learning.

Let's denote with $\mathcal{D}$ a dataset of training samples, $\mathcal{D} \coloneqq \{\mathbf{x}^{(i)}\}$, and with $l(\boldsymbol{\theta})$ the log-probability of the dataset according to the model, $l(\boldsymbol{\theta}) \coloneqq \log p(\mathcal{D} \mid \boldsymbol{\theta})$. The optimal parameters are those maximizing the likelihood $l(\boldsymbol{\theta})$. So, we may approach this solution by following the positive gradient of the likelihood. The gradient of the likelihood, with respect to each group c of parameters, is:

$$\frac{\partial l}{\partial \boldsymbol{\theta}_c} = \mathbb{E}_{\mathbf{x}' \sim \mathcal{D}}\big[\boldsymbol{\phi}_c(\mathbf{x}')\big] - \mathbb{E}_{\mathbf{x}}\big[\boldsymbol{\phi}_c(\mathbf{x})\big] \tag{5.3}$$

The two parts compute the expected value of same quantity according to a different distribution of $\mathbf{x}$ [5]. The positive term simply stands for the average value of that quantity, computed from the samples in the training dataset; while in the right-most term, we take an expectation on $\mathbf{x}$ according to the distribution induced by the model.

We'll end this quick review of MRFs, by discussing the role of latent variables. As we already know from the comparison between MDPs and POMDPs, probabilistic models that include some unobservable variables can be much more expressive than the others. Such variables are referred to as *latent* variables, or simply *hidden* variables. The latent variables of a model can be a very effective explanation of the visible quantities. Just like the states of a POMDP explain the visible observations, a latent variable representing the subject of an image is a clear motivation for the observed pixel values. Connecting the visible units directly would be very hard to do. Latent units in a MRF do not define such causal relations, but the added expressiveness is the same.

We partition the units of the model as $\mathbf{x} = (\mathbf{v}, \mathbf{h})$, where $\mathbf{v}$ and $\mathbf{h}$ denote the visible and hidden variables, respectively. A model with latent variables can be trained very similarly to what we've seen above. The only difference to equation (5.3) is that we need to always take the expectation on $\mathbf{h}$ according to the model, because

[4] Computing the exact probability $p(\mathbf{x} \mid \boldsymbol{\theta})$, as many other quantities, is intractable for generic UGMs, because often we won't be able to compute the normalizer.

[5] The function $\boldsymbol{\phi}_c$ only receives the values of the clique c. In Equation (5.3), we write $\mathbf{x}$ instead of $\mathbf{x}_c$, to ease the notation.

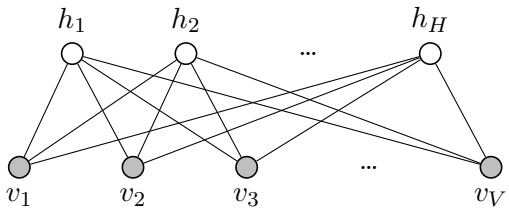


Figure 5.8. The UGM of a Restricted Boltzmann Machine. Gray nodes are the visible units.

they will never be observed in the dataset:

$$\frac{\partial l}{\partial \boldsymbol{\theta}_c} = \mathbb{E}_{\mathbf{v}\sim\mathcal{D},\mathbf{h}}\big[\boldsymbol{\phi}_c(\mathbf{v},\mathbf{h})\big] - \mathbb{E}_{\mathbf{v},\mathbf{h}}\big[\boldsymbol{\phi}_c(\mathbf{v},\mathbf{h})\big] \tag{5.4}$$

Restricted Boltzmann Machine

The specific model that we'll use is the *Restricted Boltzmann Machine* (RBM). An RBM is a MRF composed of a visible and a hidden layer. Its graph is shown in Figure 5.8. As we can see, there are no connections between variables of the same layer, and every clique has two nodes. So, the joint probability is simply:

$$p(\mathbf{v},\mathbf{h}) \propto \prod_{i=1}^{V}\prod_{j=1}^{H} \psi_{ij}(v_i, h_j) \tag{5.5}$$

with one function ψ_{ij} for each arc (v_i, h_j). This simplification, along with other nice properties, are only possible thanks to its bipartite structure. This is why it is said "restricted". Useful RBMs have strictly less hidden than visible units.

The classic RBM, which is the one we'll use here, assumes that all variables are binary: $v_i, h_j \in \{0, 1\}$. Also, it defines the following energy function[6]:

$$E(\mathbf{v},\mathbf{h};\boldsymbol{\theta}) \coloneqq -(\mathbf{v}^T\mathbf{W}\mathbf{h} + \mathbf{v}^T\mathbf{b} + \mathbf{h}^T\mathbf{c}) \tag{5.6}$$

With $\boldsymbol{\theta}$, we denote the column vector of all the parameters of the model. Namely, the weights $\mathbf{W}$ and the biases $\mathbf{b}$, $\mathbf{c}$.

In RBMs, we can easily compute the posterior probability of a layer, given an observation of the other. For binary RBMs, this is:

$$\begin{aligned} p(\mathbf{v} \mid \mathbf{h},\boldsymbol{\theta}) &= \prod_{i=1}^{V} \mathrm{Ber}(v_i; \mathrm{sigm}(\mathbf{w}_{i:}\,\mathbf{h} + b_i)) \\ p(\mathbf{h} \mid \mathbf{v},\boldsymbol{\theta}) &= \prod_{j=1}^{H} \mathrm{Ber}(h_j; \mathrm{sigm}(\mathbf{v}^T\,\mathbf{w}_{:j} + c_j)) \end{aligned} \tag{5.7}$$

[6]Thanks to the symmetry of equation (5.5), we can combine the products of all ψ_{ij} and write a single cumulative energy in matrix notation.

where sigm : $\mathbb{R} \to (0, 1)$ is the sigmoid function, $\text{sigm}(x) \coloneqq e^x/(e^x + 1)$, and $\mathbf{w}_{i:}, \mathbf{w}_{:j}$ denote the i-th row and the j-th column of $\mathbf{W}$, respectively. From the previous equations, it follows:

$$\begin{aligned} \mathbb{E}[\mathbf{v} \mid \mathbf{h}, \boldsymbol{\theta}] &= \text{sigm}(\mathbf{W}\,\mathbf{h} + \mathbf{b}) \\ \mathbb{E}[\mathbf{h} \mid \mathbf{v}, \boldsymbol{\theta}] &= \text{sigm}(\mathbf{W}^T\,\mathbf{v} + \mathbf{c}) \end{aligned} \tag{5.8}$$

To train a binary RBM, we apply equation (5.4) without modifications. To obtain the basis function $\boldsymbol{\phi}(\mathbf{x})$ we need, it's sufficient to rewrite equation (5.6) as $E(\mathbf{x}) = -\boldsymbol{\theta}^T\,\boldsymbol{\phi}(x)$. We would obtain the following training gradient[7]:

$$\begin{aligned} \nabla_{\mathbf{W}}\, l &= \mathbb{E}_{\mathbf{v}\sim\mathcal{D},\mathbf{h}}\big[\mathbf{v}\,\mathbf{h}^T\big] - \mathbb{E}_{\mathbf{v},\mathbf{h}}\big[\mathbf{v}\,\mathbf{h}^T\big] \\ \nabla_{\mathbf{b}}\, l &= \mathbb{E}_{\mathbf{v}\sim\mathcal{D},\mathbf{h}}\big[\mathbf{v}\big] - \mathbb{E}_{\mathbf{v},\mathbf{h}}\big[\mathbf{v}\big] \\ \nabla_{\mathbf{c}}\, l &= \mathbb{E}_{\mathbf{v}\sim\mathcal{D},\mathbf{h}}\big[\mathbf{h}\big] - \mathbb{E}_{\mathbf{v},\mathbf{h}}\big[\mathbf{h}\big] \end{aligned} \tag{5.9}$$

The last quantity we'll discuss is the *free energy*. As we've noted, the energy of a MRF is an useful indication about the *un*-likelihood of an observation. However, latent variables do not allow an efficient computation of the energy. So, we usually resort to a different measure, called the "free energy", $F(\mathbf{v}) \coloneqq -\log \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}$, which, for binary RBMs, can be efficiently computed as:

$$F(\mathbf{v}) = -\mathbf{b}^T\,\mathbf{v} - \sum_{j=1}^{H} \log\big(\mathbf{1} + \exp(\mathbf{W}^T\,\mathbf{v} + \mathbf{c})\big)_j \tag{5.10}$$

Training algorithm

We already know how RBMs are trained: the parameters are updated according to the gradients of equation (5.9). However, we didn't mention how to actually compute the expectations in those expressions. Here, we will look at a specific algorithm that computes an approximation to those gradients. Even though some topics also apply to different MRFs, we'll only discuss the specific case of binary RBMs.

This is not an unusual situation. In Machine Learning, we never have access to the optimal gradient. That is why we always apply a stochastic optimization algorithm, such as SGD, from the samples of the training dataset. We'll do the same here, by approximating:

$$\mathbb{E}_{\mathbf{x}\sim q}[\mathbf{v}\,\mathbf{h}^T] \approx \mathbf{v}'\,\mathbf{h}'^T \tag{5.11}$$

where the vector $(\mathbf{v}'^T, \mathbf{h}'^T)^T$ is a sample obtained from the distribution q. So, the question is: how to we obtain those samples?

Persistent Contrastive Divergence (PCD) [26] is an algorithm that allows a

[7]The third gradient, that of the biases $\mathbf{c}$, might seem null. This is not true, as different values of $\mathbf{v}$ also affect the model estimates for $\mathbf{h}$.

very efficient sampling of the variables in an RBM. The algorithm implemented in this thesis is a small variant of PCD, that we define in Algorithm 1. This explicit formulation has been obtained elaborating previous algorithms (Contrastive Divergence) and the description provided by the authors [26].

Algorithm 1 Persistent Contrastive Divergence variant

Input: A training dataset $\mathcal{D} = \{\mathbf{v}^{(i)}\}$
Output: Trained model $\{\mathbf{W}, \mathbf{b}, \mathbf{c}\}$

Initialize $\mathbf{W} \in \mathbb{R}^{V \times H}$, $w_{ij} \sim \mathcal{N}(0, \sigma)$
Initialize $\mathbf{b} \leftarrow \mathbf{0}, \mathbf{c} \leftarrow \mathbf{0}$
Initialize vectors $\{\mathbf{v}_s^{(1)}, \dots, \mathbf{v}_s^{(B)}\}$
repeat
    Collect a batch $\mathcal{B}$ of size B from $\mathcal{D}$
    Reset batch gradients, $\mathbf{g}_w \leftarrow \mathbf{0}, \quad \mathbf{g}_b \leftarrow \mathbf{0}, \quad \mathbf{g}_c \leftarrow \mathbf{0}$
    for each sample in batch $\mathbf{v}^{(i)} \in \mathcal{B}$ **do**
        Sample $\mathbf{h}_d \sim p(\mathbf{h} \mid \mathbf{v}^{(i)})$
        Compute $\hat{\mathbf{h}}_m = \mathbb{E}[\mathbf{h} \mid \mathbf{v}_s^{(i)}]$
        Sample $\mathbf{h}_m \sim p(\mathbf{h} \mid \mathbf{v}_s^{(i)})$
        Compute $\hat{\mathbf{v}}_m = \mathbb{E}[\mathbf{v} \mid \mathbf{h}_m]$
        Sample $\mathbf{v}_m \sim p(\mathbf{v} \mid \mathbf{h}_m)$
        Compute gradients
            $\mathbf{g}_w^{(i)} \leftarrow \mathbf{v}^{(i)} \mathbf{h}_d^T - \hat{\mathbf{v}}_m \hat{\mathbf{h}}_m^T$
            $\mathbf{g}_b^{(i)} \leftarrow \mathbf{v}^{(i)} - \hat{\mathbf{v}}_m$
            $\mathbf{g}_c^{(i)} \leftarrow \mathbf{h}_d - \hat{\mathbf{h}}_m$
        Accumulate $\mathbf{g}_w \leftarrow \mathbf{g}_w + \mathbf{g}_w^{(i)}, \quad \mathbf{g}_b \leftarrow \mathbf{g}_b + \mathbf{g}_b^{(i)}, \quad \mathbf{g}_c \leftarrow \mathbf{g}_c + \mathbf{g}_c^{(i)}$
        Save $\mathbf{v}_s^{(i)} \leftarrow \mathbf{v}_m$
    end for
    Update $\mathbf{W} \leftarrow \mathbf{W} + \frac{\mu}{B}\mathbf{g}_w, \quad \mathbf{b} \leftarrow \mathbf{b} + \frac{\mu}{B}\mathbf{g}_b, \quad \mathbf{c} \leftarrow \mathbf{c} + \frac{\mu}{B}\mathbf{g}_c$
until convergence

Few clarifications are needed. The gradients in equation 5.9 are composed of two terms. The left-most expectation is said "clamped" or "data-driven", and it's easier to compute. We directly approximate it as the product of two samples: $\mathbf{v}^{(i)} \mathbf{h}_d^T$. Instead, the "unclamped" term on the right is harder to compute. So, we repeatedly sample each group of variables in turn, $\mathbf{v}_m \mapsto \mathbf{h}_m \mapsto \mathbf{v}_m \mapsto \dots$, until we enter the model "stationary distribution"[8]. The last pair $(\mathbf{v}_m, \mathbf{h}_m)$ from this sequence can be considered as a full sample from the model, $p(\mathbf{x} \mid \boldsymbol{\theta})$. What PCD does is to abbreviate this long process by saving and restoring this chain from the variables $\mathbf{v}_s^{(i)}$.

In our small variant, we compute the unclamped term from expectations, rather than samples. This will reduce the variance of our estimates.

RBMs are frequently trained with simpler algorithms, such as CD-1. They

[8] This is known as "block Gibbs sampling".

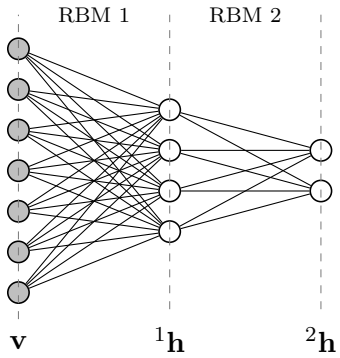


Figure 5.9. Diagram of a Deep Belief Network.

converge faster than PCD, but they are often unable to train on very uneven distributions (datasets with rare samples). In fact, when the expectations are roughly approximated, the model mostly learns to directly reconstruct the input, rather than really learning a distribution.

5.4.2 Deep Belief Networks

We're now ready to discuss the complete model of the encoder: the Deep Belief Networks. Thanks to our rather in-depth discussion of RBMs, they will be much easier to present.

A *Deep Belief Network* (DBN) is a machine learning model composed by a stack of RBMs. They are combined such that the hidden units of one RBM become the visible units of the next. This arrangement is shown in Figure 5.9. Because of this structure, we will efficiently train this model in a greedy way, one layer at the time. Assume we have a training dataset as usual, $\mathcal{D} = \{\mathbf{v}^{(i)}\}$. Initially, we train RBM 1 from $\mathcal{D}$, with our Algorithm 1. Then, before proceeding to the next layer, we keep these parameters fixed for the rest of the process. RBM 2 is trained exactly in the same way but from a dataset composed of samples generated from RBM 1: ${}^1\mathbf{h}^{(i)} \sim p({}^1\mathbf{h} \mid \mathbf{v}^{(i)})$. We repeat this process for each additional hidden layer of the network.

The graph in Figure 5.9 does not represent a UGM as the previous figures. If that graph would be really considered a single MRF, we would have to train the model as a whole, with a much less efficient procedure. So, we should consider it just a composition of RBMs. Still, it has been shown [14] that this training procedure computes indeed an approximation to the unified model.

We can now look at the general picture of how the encoder operates. As we've seen from the structure of Figure 5.6 on page 54, we define one encoder, that is one DBN, for each region. Then, the sequence of RBMs is trained from observations of

this portion of the input frame. We consider as encoding vector the values of the deep-most hidden layer of the DBN. Once a DBN is fully trained, we predict the encoding vector ${}^{D}\mathbf{h}$ associated to an input $\mathbf{v}$ through the following chain of samples: $\mathbf{v} \mapsto {}^{1}\mathbf{h} \mapsto \cdots \mapsto {}^{D}\mathbf{h}$.

5.4.3 What does it learn

The model of the encoder has been fully defined already. In this section, we look at its operation from a more general perspective. We try to answer these questions:

- Why can latent variables be considered an encoding?
- Why DBN / RBM?
- What does the encoding represent?

We said that the values assumed by the hidden layer of an RBM can be considered an encoding of the input vector. This is not obvious, because it depends on the meaning of those units. We observe the following: training an RBM means to find a mapping $\mathbf{h} \mapsto \mathbf{v}$ such that each training sample $\mathbf{v}^{(i)}$ is a *very probable* observation, given the hidden units associated to $\mathbf{v}^{(i)}$ (the dependency is cyclic, but we focus on the reconstruction $\mathbf{h}^{(i)} \mapsto \mathbf{v}^{(i)}$). Thus, the hidden units allow a reconstruction of the visible units.

So, the latent vector is an encoding, but how is it composed? In unsupervised models, we can never answer to this question in advance. This is the strength of these models, after all: they automatically assign a meaning to those units, as convenient for the optimization. In binary RBMs, we can move one step further. Since each unit is binary, we know it contains exactly one bit of information. In other words, each of them acts as an independent feature detector that is associated to a specific pattern of the input. They become active, assuming a value of 1, when the input respects their constraint. So, each entry of the encoding vector indicates the presence or the absence of some patterns that the model has learnt to recognize.

This leads us to motivate the choice of RBM and DBN rather than other types of encoders: the RBM achieves the maximum level of decoupling in the encoding vector. This allows to understand more easily the meaning of each encoding the model produces (for humans). More importantly, it allows to decide the truth of our fluents by reasoning in terms of *present and absent patterns* on the input vector (for machines). We'll talk about this last possibility in Section 5.5.

The RBM is an universal approximator: any input distribution on $\{0,1\}^V$ can be approximated arbitrarily well with an RBM with enough hidden units [17]. This is just a theoretical result, we're interested in approximate reconstructions, in practice. This discussion seems to motivate just the RBM, not the DBN. However, even though both models are universal approximators, a DBN can produce the same

distribution as an RBM with much less parameters. In fact, there exists cases where a two-layers DBN requires exponentially less parameters than a comparable RBM. So, the DBN is much more efficient. So, by adding layers to a DBN we can reduce the size of the latent vector, which is our encoding, while maintaining the same accuracy. Because, given the same output size, a DBN can recognize much more complex patterns.

The last topic to discuss is the appropriate size of the encoding. Initially, we might think that the number of latent variables should depend on the size of the input space. This is not true, instead. This number only depends on two factors: the complexity of the input distribution, and the desired level of accuracy. The dimensions of the encoding should be *at least* the $\log_2$ of the number of configurations that we want our model to recognize and encode. Such minimal encoding might not exists or may not be reachable during training. So, it's always better to use a larger number of units. This size also determines a boundary between the relevant features and those that are treated as noise and will be ignored[9].

Example 13. Consider the region of the fluents *Full* and *Empty* of Figure 5.5. The minimum size of the encoding is the number of bricks inside that area, because we may use each bit to detect the presence of one brick. This optimal result might not be reachable. So, we could use slightly more units than this minimum. We know that the units will indicate the presence of bricks, because that is the most evident change in that region. With this meaning, the model is able to reconstruct as many pixels as possible. This also means that, regardless of the size of the encoding, we probably won't be able to detect the presence of the ball, passing through the region. The DBN focus on the most evident changes of that group of pixels.

In Machine Learning, the depth and width of the network, which is a DBN in this case, are hyper-parameters that needs to be tuned. The considerations above will only help to start from a reasonable model size.

A final side note. A consequence of using binary RBMs is that the continuous intensity of each pixel needs to be converted to a binary value. In this work, we set a threshold to transform each pixel to either black (0), or white (1). However, we don't consider this a real limitation for our model. We've selected binary RBM for simplicity, but there also exists Gaussian-binary RBMs, which handle continuous values for visible units, and binary for hidden units as usual.

[9]The size determines the number of relevant features, not which are relevant.

5.5 Boolean functions

Each encoder transforms the pixels of one region to a vector of few binary indicators. This sensibly reduces the complexity of the problem by extracting the relevant quantities, but the problem is still there: how do we map inputs (now, encodings) to truth values for the fluents? If we consider an encoding vector $\mathbf{h} \in \mathbb{B}^H$ and a propositional symbol $p \in \mathcal{F}$, this means to find the appropriate Boolean function $f_p : \mathbb{B}^H \to \mathbb{B}$, that captures the desired concept for p.

A solution, which is now viable, is to define it manually. To do so, we first need to understand which pattern of the input each unit of the encoding represents. Let's denote with $\mathbf{e}_i$ a vector of zeros except for a 1 at the i-th position. It's possible to discover the meaning associated to each unit by reconstructing the expected input with $\mathbb{E}[\mathbf{v} \mid \mathbf{h} = \mathbf{e}_i]$. Thanks to the separation of features, we don't need to visualize all the other combinations of hidden units.

5.5.1 Searching monomials

Since each element of the vector $\mathbf{h}$ detects a feature of the input, we may now define the truth of a symbol p as: "p is true when the feature h_1 is present and h_3 is not". We will conveniently express such Boolean functions with with formulae of Propositional logic: $h_1 \wedge \neg h_3$. As usual, a formula is a concise definition for the set of inputs that satisfy it. So, by writing this expression, we've successfully defined those images in which the symbol should be assigned to true (we only talk about the set of positive inputs, because we assume it negative otherwise).

We may be quite satisfied with this solution already. Writing a propositional formula is much easier than collecting a dataset of input-output samples for supervised learning. Yet, in the following part of this section, we'll investigate how we could learn even this final valuation function.

Our goal is to find the appropriate propositional formula among the set of all formulae over H atoms. Unfortunately, this search space is huge: there are 2^H interpretations for H propositions, among which we should choose the set of those associated to the true fluent value. There are 2^{2^H} of those sets. Since we train from very weak temporal constraints, we'll reduce the search space with the following assumption: we assume that the target concept[10] of the fluent can be expressed with a conjunction of literals, which are positive or negative atoms. This type of formula is called a *monomial*. Since in this formula each symbol h_i can be positive, negative or absent, there are at most 3^H different monomials from H atoms. This is

[10]The word "concept" is used as a synonym for the desired fluent valuation, that is the set of inputs the symbol should represent.

a much more tractable search space.

By applying a sequence of logical equivalences, any formula of propositional logic can be converted in *Disjunctive Normal Form* (DNF). A formula is in DNF if it is a disjunction of monomials. Thanks to this property, any complex concept can be split into a group of monomials $M_1, \ldots, M_m$, each respecting our assumption. So, we can define m fluents A_i whose valuation functions are M_i. Once these monomials have been learnt, we can write $(A_1 \vee \cdots \vee A_m)$ in place of the original symbol, in every place it appears. This shows how this procedure may be generalized. However, as we'll see in Section 5.7 we should restrict to very simple concepts in the first place.

5.5.2 Learning Boolean rules with genetic algorithms

We've defined both a search space, which is the space of monomials over the entries of the encoding vector, and an objective, that is the satisfaction of the temporal constraints. These constraints are expressed in a single LDL_f formula ψ, written from the atoms in $\mathcal{F}$. So, we formulate the learning problem as: finding a set of functions, $f \coloneqq \{f_p \mid p \in \mathcal{F}\}$ with $f_p : \mathbb{B}^H \to \mathbb{B}$, such that the trace of assignments produced from observations of any episode satisfies the temporal constraint ψ. Each function f_p receives in input the encoding vector $\mathbf{h} \in \mathbb{B}^H$ computed from the region where p is defined. A solution for this problem is the entire set of functions, because consistency with ψ is a property of the group, not the single valuation function.

Clearly, we can't verify that the constraint is satisfied for any episode. For the moment, we assume to approximate this test by running a large number of episodes, instead. By checking that ψ is satisfied in each of these traces, we're able to assess whether some f is a solution. However, satisfaction is a binary test: it doesn't give any indication about how to *reach* such solution. On the other hand, an exhaustive search may be unfeasible, for a large H. These considerations lead us to prefer search methods based on sampling: the idea is to repeatedly sample a candidate set of functions from the space of monomials, then verify it against the constraint.

Even random sampling struggles in high-dimensional spaces. For this reason, stochastic search algorithms assume that some heuristic function is available $f_{\mathrm{he}} : \mathcal{X} \to \mathbb{R}^+$. The purpose of this heuristic is to provide a very approximate ranking over candidates, so that most samples are drawn from the most promising portions of the candidates space $\mathcal{X}$. Let's leave aside our definition of f_{he}, for a moment.

Genetic Algorithms

The search algorithm we've selected is a Genetic Algorithm (GA). It is a local search algorithm which repeatedly samples batches of candidates that are modifications of the previous ones. The motivation for this choice is that, with respect to other

local search algorithms, GAs allow to easily control both nondeterminism and parallelization, as we will see. The idea of applying GAs to learn propositional sentences has also been confirmed by previous works, which applied them for Concept Learning [16]. This is, in fact, a concept learning scenario, in which the target concept is described through temporal constraints rather than positive examples.

In the context of GAs, the batch of candidates is called a *population* of individuals, and the heuristic function is called the *fitness function*. We'll denote the population with $\mathcal{P} \coloneqq \{(\mathbf{x}^{(i)})_{i=1}^{N}\}$, for an even size N. Each individual is represented with a sequence of fixed length, $\mathbf{x}^{(i)} = \langle x_1^{(i)}, \dots, x_L^{(i)} \rangle$, composed of symbols x_j called *chromosomes*. A GA starts by sampling an initial population $\mathcal{P}$ of size N. Then, it repeatedly follows these steps:

Fitness Compute the fitness value for each individual: $v_i \leftarrow f_{\text{he}}(\mathbf{x}^{(i)}), \forall\, \mathbf{x}^{(i)} \in \mathcal{P}$. The probability of reproduction for individual i is defined as $r_i \leftarrow v_i / \sum_j v_j$.

Reproduction Sample a new generation, $\mathcal{P} \leftarrow \{(\mathbf{x}'^{(i)})_{i=1}^{N}\}$, where each individual is assigned as: $\mathbf{x}'^{(i)} \leftarrow \mathbf{x}^{(k)}$ for $k \sim \text{Cat}(\mathbf{r})$. The categorical distribution, $\text{Cat}(\mathbf{r})$, generates an index k with probability r_k. So, the fittest individuals are more likely to reproduce.

Crossover Sample $N/2$ crossover points with a uniform discrete distribution: $c_i \sim U(1, L-1)$. For each pair of parents $(\mathbf{x}^{(2i-1)}, \mathbf{x}^{(2i)})$ with $i = 1, \dots, N/2$, apply a crossover with probability p_c. A crossover is an exchange the first c_i chromosomes between $\mathbf{x}^{(2i-1)}$ and $\mathbf{x}^{(2i)}$.

Mutation For every individual $\mathbf{x}^{(i)} \in \mathcal{P}$ and chromosome $x_j^{(i)} \in \mathbf{x}^{(i)}$, apply a mutation to $x_j^{(i)}$ with probability p_m. A mutation is a substitution of a chromosome with a new one, randomly sampled.

We've said that GAs allow to regulate the desired amount of nondeterminism and parallelization. For the latter, we can increase the population size N. The additional time required by each cycle, may be strongly compensated by the minor number of cycles required for convergence. Furthermore, modern parallel computing libraries will show a very little overhead each additional individual.

The randomness of the search can be tuned by selecting the parameters p_c and p_m, which are the crossover and mutation probabilities. High values correspond to a mostly random search, while low values correspond to steady convergences toward the fittest candidates. Some nondeterminism is clearly desirable in presence of very approximated heuristics as those that we'll define.

During initialization, we've asked to randomly sample a population. Since the individuals are composed of chromosomes, it all comes to being able to sample

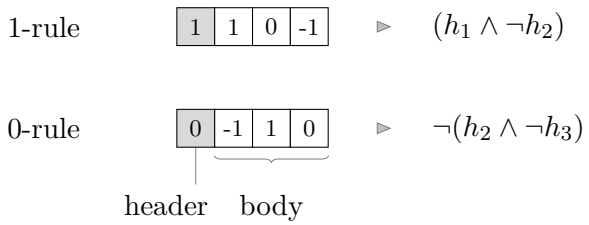


Figure 5.10. A 0-rule negates the output of the conjunction.

chromosomes, just like during the Mutation phase. We assume samples can be easily sampled with an uniform distribution. Usually, this is indeed easy, because, in many problems, the elements of the search space can be formulated as sequences of numbers from small domains. Also, we can note that individuals do not need to be defined with a homogeneous sequence, every chromosome position might span a different domain.

Boolean rules

The learning algorithm is ready. We just need to formulate elements of our search space in terms of fixed-length sequences of symbols. Since our goal is to find a set of functions $f = \{f_p \mid p \in \mathcal{F}\}$, we need to express f with chromosomes. For the moment, let's focus on each f_p.

We represent each function with a propositional sentence over the H Boolean features in the encoding vector. Since we assumed that each target concept is expressible with a monomial, a fixed-length representation is easy to define. Given a monomial M, whose atoms are the entries of a vector $\mathbf{h} \in \mathbb{B}^H$, we define the associated individual as the sequence $\mathbf{x} \in \{-1, 0, 1\}^H$, where each chromosome x_i is 1 if the atom h_i appears positive in M, 0 if h_i is negative, and -1 if it's absent. For example, the sequence $\langle 1, -1, 0, 1\rangle$ represents $h_1 \wedge \neg h_3 \wedge h_4$.

The assumption of monomials was introduced both to reduce the search space and to allow this encoding of fixed length. At a negligible cost, that is by introducing an additional chromosome, we can extend this representation to include both monomials and negations of monomials. We call these new functions *Boolean rules*. We explain them through the examples of Figure 5.10. A 1-rule simply stands for a monomial, as before. Instead, a 0-rule produces the opposite output: it is true, if the constraint on the input $\mathbf{h}$ is *not* satisfied. The negation of a monomial is a clause (a disjunction of literals). So, this extension can be considered a very restricted form of disjunction.

We're now ready to define a fixed-length encoding for the whole set of functions. Assign any order to the fluents in $\mathcal{F}$, so to indicate with f_i the valuation function for a symbol $p_i \in \mathcal{F}$. Given a set $f \coloneqq \{f_i \mid i = 1, \ldots, |\mathcal{F}|\}$ with $f_i : \mathbb{B}^H \to \mathbb{B}$, we

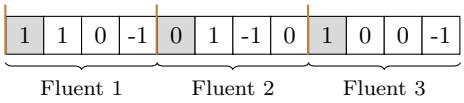


Figure 5.11. The example of an individual for a problem with three fluents and an encoding size of 3.

define the individual associated to f as the sequence $\mathbf{x} \coloneqq \mathbf{x}_1\mathbf{x}_2 \dots \mathbf{x}_{|\mathcal{F}|}$ obtained by concatenation of the Boolean rules $\mathbf{x}_i$ representing each function f_i. This arrangement is shown in Figure 5.11. Each entry in the vector $\mathbf{x}$ is a chromosome. As such, mutations can happen at the level of single digits of the sequence. The head of any rule (gray block) is sampled from $\{0, 1\}$, while positions in the body are sampled from $\{-1, 0, 1\}$.

Genetic algorithms are a type of local search because they modify single chromosomes. This means that how individuals are represented is important because it influences the search. In our case, preliminary experiments have shown that the crossover operation was detrimental to convergence, rather than helpful. This is probably due to Boolean rules: the body of a 1-rule may be completely wrong if joined with the header 0, and vice versa. So, we slightly modify the GA algorithm to restrict the sampling of crossover points to indices that leave rules intact (the brown bars of Figure 5.11). Now, crossovers can only modify candidate sets f by exchanging entire functions f_i between them.

In this discussion, every function f_i assumes a domain with the same number of dimensions H. This means that all regions are encoded to binary vectors with H entries. This restriction is not needed by any part of the algorithm, and it may be convenient to use the appropriate encoding size, based on the complexity of each region. However, the implementation provided assumes a common size H, for simplicity.

Heuristic function and nondeterminism

We conclude this presentation of the learning algorithm by discussing the heuristic function and few other details. The heuristic function $f_{\text{he}} : \mathcal{X} \to \mathbb{R}^+$ is used by the GA to compute the fitness values that determine the probability of survival for each candidate, $f \in \mathcal{X}$. The heuristic should have a global maximum in correspondence of a solution of the problem, so to converge to the desired target. So, in our problem, the maximum value must be returned for a candidate set f whose output always satisfies the temporal constraint ψ. We can't verify "always", but we can test that the trace produced from f satisfies ψ in a large number of episodes.

A first possibility would be to define the heuristic in the following way: over E episodes, the value of $f_{\text{he}}(f)$ is the fraction of episodes in which $\pi^{(i)} \models \psi$, where $\pi^{(i)}$

is the trace produced by f from the sequence of encoded observations of episode i. The maximum value is 1 and, for a very large E, it is assigned to a solution[11].

The heuristics need to be computed at every cycle of the Genetic Algorithm. Clearly, running many episodes each time is unfeasible. So, we'll only test on an much smaller number of episodes. As a consequence of the small E, the fraction of tests passed becomes only an approximation to the true heuristic. What is really important, however, is that *on average* these values will be correct. Since the reproduction step is a stochastic update of the model, to guarantee the convergence of the algorithm, it is sufficient that $\mathbb{E}[f_{\text{he}}(f)] = 1$, if f is a solution. This is a similar concept to Stochastic Gradient Descent, where each update is performed on an approximate gradient computed from a small batch. The expected value of these corrections still point toward the correct direction.

Since each result of the heuristic function is less accurate, we should prefer soft updates of the model. In a GA, we can do this by producing fitness values that are never null. For example, we might define $f_{\text{he}} : f \mapsto [0.3, 1]$ by linearly mapping the output to the appropriate range[12]. This slightly uniforms the reproduction probabilities. As a consequence, convergence becomes slower but the "averaging" effect of different cycles is stronger, which allow to reach better solutions. This is the same effect as that obtained with a low learning rate in supervised learning.

There are still two problems to address with the definition of f_{he} above: discretization and degenerate solutions. Let's discuss discretization first.

An heuristic function defines a partial ordering over candidates. Higher values are associated to better individuals. When this function assumes few discrete values, the ordering becomes very weak and it may not be sufficient to guarantee convergence. This is indeed the case with the definition above, in the case of few episodes E (for one episode, it just returns 0 or 1). It's always better to prefer continuous values, instead. For this purpose we define a metric m_c that we call "consistency". For a candidate f and an episode i, $m_c^{(i)}$ is defined as the fraction of time instants in which the automaton $\mathcal{A}_\psi$ associated to ψ is on a final state. We may now define the heuristic as:

$$f_{\text{he}}(f) \coloneqq \frac{1}{E} \sum_{i=1}^{E} m_c^{(i)} \tag{5.12}$$

This creates a really dense, even though rather arbitrary, ordering among functions. What is important, however, is that the maximum of this function is still associated to a solution. For generic constraints, the answer is no. However, this is the case if

[11] A solution is one that respects the temporal constraint. This do not necessarily mean that the valuations represent the desired concepts.

[12] The global scale factor is irrelevant; only the proportion between the outputs contributes to probabilities.

we restrict to *safety* properties. A safety constraint is any formula that expresses that "something bad never happens": $\psi = [\, true^* \,]\neg\psi'$. If the temporal constraint is a safety property, and the trace of valuations satisfies ψ, then the automaton $\mathcal{A}_\psi$ never reaches a non-final state, and $m_c = 1$. Actually, we can write any prefix-closed property, which is a slightly larger class of properties than safety. Initial conditions are allowed, for example. A formula ψ is prefix-closed if the corresponding minimal automaton $\mathcal{A}_\psi$ has a single non-final state which is a sink.

The second issue to tackle are degenerate solutions. Even without the restriction above, safety properties often arise when talking about constraints and illegal trajectories. Since these are prefix-closed properties, valuation functions that don't predict any change of the fluent at all always respect the constraint. We'll illustrate this with an example.

Example 14. Let's consider Example 11 on page 51. The associated automaton is shown in Figure 5.3. Since the only rejecting state is a sink, the temporal constraint ψ is a prefix-closed property. As we've already pointed out, a set of valuation functions $f(\mathbf{h}) \coloneqq \{\}$, which predicts that both fluents are always false, satisfies ψ. The constraint excludes illegal trajectories but doesn't say that something should eventually happen (the agent might think that the door is always closed, so to avoid mistakes).

Constraints intentionally do not force specific trajectories, in fact. Because the agent might not be able to achieve the task yet. So, to address this issue, we define a second metric, the "sensitivity" m_s. We define $m_s^{(i)}$ as the fraction of visited final states of the minimal automaton $\mathcal{A}_\psi$ during episode i. As the name suggests, this metric is an incentive to be sensitive to input changes, and reflect those variations to the output. Clearly, we're not guaranteed that the transitions of $\mathcal{A}_\psi$ will be traversed in response to meaningful changes of the input, but we argue that from a restricted input space and the appropriate constraint, suboptimal valuations functions may not be able to explore the whole graph without never reaching a rejecting state. Consistency and sensitivity work in opposition: the latter is an incentive for exploration, while the former is a restriction. The only restriction due to the sensitivity metric is that we shouldn't try to learn fluents which remain constant, for example because it cannot influence them, even by change. Because the valuation functions have incentives to induce a change in all of them.

The final definition of the heuristic function is:

$$f_{\text{he}}(f) \coloneqq \frac{1}{E}\sum_{i=1}^{E} m_c^{(i)} \cdot \max_{i=1,\dots,E} m_s^{(i)} + \text{const} \tag{5.13}$$

where const is a minimum constant value that allows a minimum probability of

reproduction. Consistency is not averaged but maximized, because its purpose is just to exclude degenerate solutions: we only want to observe a change in at least one of the E episode. This is not the true goal to achieve, which is indicated by consistency.

So, the algorithm proceeds as follows. Initially, E is set to a low number, as required by the speed of the machine. We'll start to observe that the distribution of heuristic values for each individual slowly drift toward the maximum. At the latest stages of the algorithm, we increase the number E of episodes, so that tiny corrections can be applied to refine the solution. At convergence, we select as best individual the set of functions that achieve 100% of consistency with the highest sensitivity. This is taken as final solution f_*.

In order to validate the solution extracted, we analyze whether each function $f_p \in f_*$ represents the desired concept for $p \in \mathcal{F}$. To do so, we define the set of encoding vectors in which the fluent is true: $\mathcal{H} \coloneqq \{\mathbf{h} \mid \mathbf{h} \models f_p\}$[13]. Then, we reconstruct the input images by computing their approximate expectation with n samples:

$$\mathbb{E}[\mathbf{v} \mid \mathbf{h} \models f_p] \simeq \frac{1}{n} \sum_{i=1}^{n} \mathbf{v}^{(i)} \qquad \begin{array}{c} \text{with} \quad \mathbf{v}^{(i)} \sim p(\mathbf{v} \mid \mathbf{h}^{(i)}) \\ \text{and} \quad \mathbf{h}^{(i)} \sim U(\mathcal{H}) \end{array} \tag{5.14}$$

$U(\mathcal{H})$ represent the uniform distribution over the elements of $\mathcal{H}$.

We won't discuss the remaining parameters of the algorithm: the mutation probability p_m, the crossover probability p_c, and the population size. The requirements on these are not really strict, and we can consider them as hyper-parameters simple to tune. For a detailed discussion about their effect in GAs, the reader should refer to [12].

5.6 Training and incremental learning

The model has been described completely already. We can now take a broader view on the various steps of learning, going from an inexpert agent to a complete restrained Deep RL agent with learnt features. Thus, combining every technique we've discussed until now. The general steps of the learning procedures are shown in Figure 5.12. Although "agent" in its general meaning refers to the complete artificial entity interacting with the environment, for the moment, we'll use it to indicate just the decision maker, so to discuss the features extractor separately.

[13] We're using the notation with some flexibility here. By ignoring the specific representations, we can consider $\mathbf{h}$ as an interpretation for the propositional symbols h_i that appear in f_p.

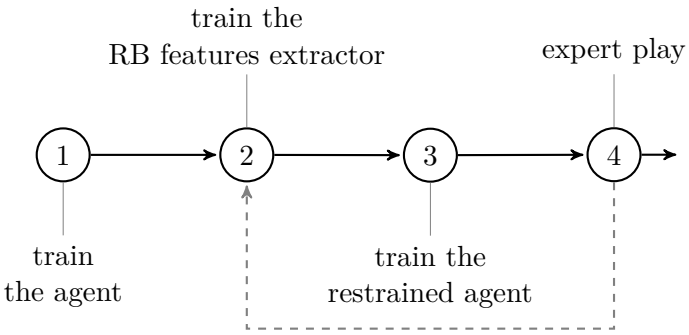


Figure 5.12. The general steps of the training procedure.

.

1 – Train the agent

The first step is to train an agent with Deep RL. The environment is one of the Atari games, the training algorithm is Double DQN [28], described in Section 3.2.2, and the agent model is the Deep Q-Network defined in Section 4.3.1.

During this phase, the agent only receives observations and rewards generated from the environment. The outcome of this training is an agent that, with some success, tries to maximize the environment rewards. As we've discussed in the previous chapter, the agent's performances may be really low in presence of partial observations and, in some environments, the agent may not learn anything at all (this is the case of the game Montezuma's Revenge, for example). These low performances are equally fine, as we don't need specific capabilities to proceed to the next step.

2 – Train the RB features extractor

Either if the agent should achieve some temporally-extended goal (Section 4.1.2) or should improve its performances on partial observations (Section 4.1.1), we'll apply the Restraining Bolt. First, we need to select a set of fluents $\mathcal{F}$ that are useful in the definition of some $\text{LTL}_f/\text{LDL}_f$ temporal specification.

In the definition of $\mathcal{F}$, we must restrict to symbols that respect the assumptions listed in this chapter. In fact, their intended valuation function must be learnable by the complete model we've described here. One of the assumptions, for example, is that the truth of each of these propositions should change, at least sometimes. This means that we cannot learn symbols that the agent obtained from step 1 is not able to influence at all, even by chance (consider for example a symbol *AgentAtEnd*, where *End* is a position in a maze, that the agent is unable to reach, yet).

Now that the fluents have been selected, we can train the complete model of the RB feature extractor that we developed in this Chapter. This is done incrementally: we train all the encoders first, then the Boolean functions.

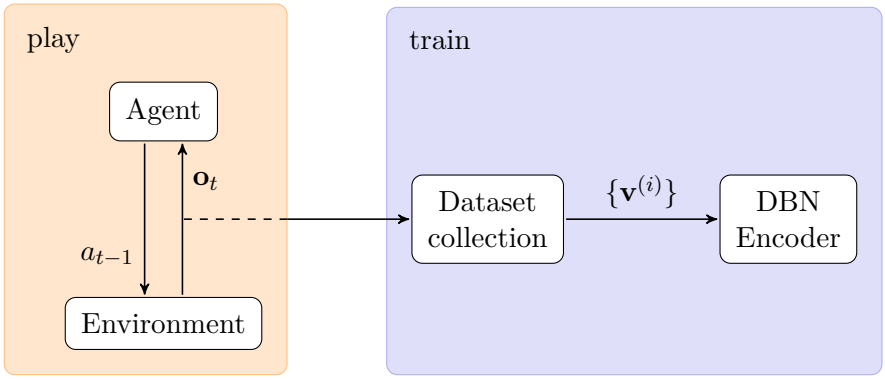


Figure 5.13. Training step 2 – the encoder.

Every symbol $p \in \mathcal{F}$ has an associated region from which it is valuated. For each of these regions, we train an encoder as shown in Figure 5.13. On the left, the agent obtained from the previous phase continuously interacts (plays) with the environment. The agent is not trained here, its only purpose is to produce the stream of observations $\mathbf{o}_t$. These observations are cropped to the region of each encoder and collected in a dataset. This is then used as training set for the Deep Belief Network. For efficiency, we don't need to store every observation. Instead, the dataset contains just the most recent n samples, for a n large enough. Then, at each training step, we sample a random batch $\{\mathbf{v}^{(i)}\}$ from it.

The role of the agent is not to achieve high rewards. Instead, it should explore the environment states as thoroughly as possible. In fact, to train a good encoder we need varied observations, because every image received for prediction should have a meaningful encoding vector assigned. For this reason, it's always better to prefer exploration policies during training, because they usually include a good amount of stochastic actions. We use a combination of classic and custom exploration policies from Section 3.1.3.

Once we've trained all the encoders, we can move on to the Boolean functions. As we know, they are learnt with a Genetic Algorithm which learns all the valuation functions at the same time. We take as result of this training the set f_* that is fully consistent with the specification, with the highest "sensitivity" metric. The main difference with the training shown in Figure 5.13 is that the heuristic function computes the metrics on episodes, not single observations. So, instead of collecting an unordered dataset of images, now we need to receive the exact sequence produced. Also in this case, exploration policies are needed to observe as many different behaviours as possible.

3 – Train the restrained agent

Step 2 produces a trained features extractor, able to valuate all fluents that we use in the temporal specification. It predicts the Boolean value that they assume in each observation.

The next step is to train a "restrained" agent, that is an agent with the Restraining Bolt applied. We use the Double DQN algorithm, just like in step 1. However, the Q-Network we use here has been defined in Section 4.3.2, which is slightly different. The general setup of a restrained agent has been shown in Figure 4.4 on page 36. At the end of this step, we obtain a Deep RL agent that achieves the goal induced by the rewards generated by environment and the Restraining Bolt. Its actions are decided from observations $(\mathbf{o}_t, \vec{q}_t)$.

4 – Expert play

If, with the fluents available, we've been able to write a temporal specification that guides the agent to the final goal, the process should be now complete. The agent obtained at the previous step is an expert player, able to achieve the temporal specification.

However, it is possible that we don't have a satisfactory set of fluents available, especially if the agent's performances from step 1 were poor. In this case, it's possible to train an agent for a partial goal. For example, in an exploration game, we might train the agent just to leave the first room of a labyrinth because we can't say anything about parts of the environment that have never been reached.

A possibility is to proceed iteratively. We can use the agent obtained from the previous step to train a new feature extractor for a more complete set of fluents. As shown by the dashed arrow in Figure 5.12, we can continue from step 2, and repeat as needed. Every time, the agent will be more capable, allowing to observe new regions of the state space.

5.7 Limitations and improvements

In this whole chapter, we've proposed an innovative way to train an extractor of Boolean features whose meaning is assigned by the designer. The approach has some interesting properties, such as defining the desired valuations with temporal logics, and avoiding the manual work required by labelled datasets. However, this thesis is a first study in about this possibility. Due to the simplifications and assumptions we've taken in this work, we'll be able to apply these ideas just to a specific class of fluents and observations. The purpose of this section is to review many of these limitations and to suggest how to overcome them.

Fixed regions

The first limitation is the assumption that the truth of each fluent can be decided from a fixed portion of the input image. This implicitly assumes that the relevant region of the input is always in the same position. This is only applicable to environment with observations of the scene that don't move. In very simple games, such as Pong, Breakout, Mr.Pac Man this is the case, but in other games this may not be the case.

Regions are a reasonable simplification; their fixed position may be not. Instead of defining regions by their position, we may find the relevant location based on some other informations. For example, we could associate visual feature to each region. By applying object detection techniques, when that feature is found inside the image, the portion of pixels surrounding it is selected as input for the encoding.

Weak constraints

Many assumptions and heuristics aim at solving a fundamental issue with temporal constraints: they are very weak descriptions of the desired concepts. As we've already observed in Section 5.1, there may exist many sets of functions which are related by the same temporal behaviour but don't represent the desired meanings of the fluents. This is due to the weak input–output association these constraints create. We can imagine two possible ways to create a stronger grounding of the symbols to the intended meanings.

The first is an hybrid approach. We collect a small set of diverse input images, and we manually label them with the desired truth value for all fluents. This creates a dataset for supervised learning, $\mathcal{D}_{\mathcal{F}} = \{(\mathbf{o}^{(i)}, v^{(i)})\}$ with $v^{(i)} \in 2^{\mathcal{F}}$. Encoders are trained in an unsupervised way, as usual. Instead, the Boolean functions can be trained from a conjunction of both the temporal constraints and prediction error on this dataset. They may be combined into a single heuristic function, for example. We argue that even few labelled samples can be a huge hint about some important associations. Also, thanks to temporal constraints, this dataset can be really small, which would be impossible with supervised learning.

A second possibility is the following. Suppose that the meaning of some fluents is already known and don't need to be leant. These constitute are set of grounded symbols. By relating the fluents to learn with those already grounded, we can write much more meaningful temporal constraints. Grounded symbols may arise from conditions that are easy to detect, or maybe because we have some ad-hoc sensor or trained model already available. Other grounded fluents are those that the agent knows as facts. For example, we could define the set of symbols a_j which are valuated true if the agent has just performed action j. This may allow to encode in temporal

constraints how the environment evolves as a consequence of actions.

A final issue with temporal constraints is that, because of the consistency metric, we had to limit ourselves to prefix-closed properties. This class of temporal properties may be enough in most cases, but if we want to exploit the full expressiveness of the logic, we should abandon the consistency metric. With some additional hints, such as those discussed here, we may use only satisfaction as criterion, instead of some other heuristic.

Guarantees of learnable fluents

It is perfectly normal that Machine Learning models give little convergence guarantees. Provided that the architecture is expressive enough, convergence depends on many other factors, such as initialization, learner biases, training input. Just like the results of supervised learning depend on the dataset, the features extractor we learn depends on the observation it has received for training from the player. Fortunately, thanks to the compression operated by the encoder, it's really easy to visualize the result of training, either by predicting the fluents on test data, or by visualizing the input pattern that the function detects (see Equation 5.14).

Instead, a more serious uncertainty comes from the fact that the target concept may not be representable with a Boolean rule over the entries of the encoding vector. If we knew that some symbol cannot be expressed with a monomial, we could simply divide it into simpler fluents. Unfortunately, we don't know if that is the case. The encoder is an unsupervised model that is free to assign any meaning to the units of the latent vector. We can't know in advance whether the set of encoding vectors we're interested in is representable with a monomial. Fortunately, when the intended meaning of a fluent is to to detect a single input image (regardless of some small noise filtered by the encoder), there is just one encoding vector associated, which is certainly expressible as a conjunction of terms.

Chapter 6

Tool for learning symbols through RL: the AtariEyes package

This chapter describes the software realized in this thesis, that we called "AtariEyes". Its purpose is both to implement the ideas that have been presented until now and to generate the experiments shown in Chapter 7.

Apart from being a realization of the ideas proposed, this software has some interesting qualities:

Clarity Every method and structure has been documented.

Efficiency Thanks to an heavy use of parallel computing libraries, it benefits from GPU acceleration.

User friendly The extensive command line interface allows to experiment with the package as it is, or, thanks to its modular design, individual structures can be reused in future developments.

Regarding its general functionality, through the commands provided, the user can:

- Choose any *environment* from the Atari 2600 collection.
- Train a Deep Reinforcement Learning *agent*. The algorithm is Double DQN and the agent's model can be either the original Atari model (Section 4.3.1) or the restricted agent (Section 4.3.2).
- Train the *feature extractor*, because it implements every model and algorithm presented in Chapter 5.

- *Play*, *visualize* and *record* any of these agents while they interact with the environment.

This chapter is divided in two sections: Section 6.1 documents how the software can be used from a user perspective; while Section 6.2 explains some of the most interesting details about the implementation.

6.1 How to use the software

6.1.1 Tools and setup

The software `atarieyes` is a Python package. It is publicly available at the GitHub repository: `cipollone/atarieyes`. It can be installed with the `pip` command as any other Python package, we just need to point to this git repository. The installation command is:

```
pip install git+https://github.com/cipollone/atarieyes
```

This command installs this package from the master branch. If we need to work on some specific revision, for example on the `develop` branch, we can append `@develop` or any other commit to the previous address.

Dependencies are automatically installed by `pip`. In Python, however, it is common to run applications inside virtual environments. Just run this installation command within a container, to avoid dependency conflicts with other applications. One rather particular dependency is TensorFlow, which is a famous Machine Learning library that we use for parallel computing. Following the instructions of the specific container application, we can reuse some preexisting system installation, if we need. Currently, the supported TensorFlow version is only 2.1, but future 2.x versions might also be compatible.

The package is written in Python 3 and the minimum version required for the interpreter is 3.7. This requirement should be met by most modern operating systems. If that's not the case, we suggest to use `pyenv`, which allows environment-specific Python installations.

Once installed, we can use the `atarieyes` package. As we will see, we often use this module through its command line interface. However, if we want just to include some structures and algorithms in other applications, we can `import atarieyes`. For development, it may be also useful to look at the source code. We can clone the repository:

```
git clone https://github.com/cipollone/atarieyes.git
```

This is also useful if, for any reason, some updated dependency is no longer compatible with this package. What we can do is to `cd` to this cloned directory, then run `poetry install`. Poetry is the container application that we use. This command will install the exact dependency versions that have been tested during development and are guaranteed to work.

6.1.2 Execution

To run this package as a script we run the following command from the same environment where we've installed it:

```
python3 -m atarieyes
```

The reader can assume that any `atarieyes` command that we'll see, is executed through `python3 -m`.

Getting help

The package provides a compete command line interface, with many options that control the training process. In these sections, we look at the most important commands. For any doubt, or to list the arguments not discussed, we can use the `--help` option, abbreviated as `-h`. When added, it prints the arguments that are supported by any command. For example, running `atarieyes -h` produces the following message[1]:

```
usage: __main__.py [-h] [--list] [--from FROM] {agent,features} ...

Feature extraction and RL on Atari Games

positional arguments:
  {agent,features}  Choose group
    agent           Reinforcement Learning agent
    features        Features extraction

optional arguments:
  -h, --help        show this help message and exit
  --list            List all environments, then exit
  --from FROM       Load arguments from file
```

[1] We use the `argparse` library for parsing and generating these messages. The file `__main__.py` file also acts as reference for these commands.

The `--list` option prints the unique identifier of every Atari game. To use any of these games as environments, we pass its name to the `--env`/`-e` option, where appropriate.

The `--from` option allows to execute the commands and options stored some JSON file. The JSON must be a dictionary of parameter names and values. After any "train" command, an `args.json` is automatically saved. The purpose of this option is allowing to repeat, resume or slightly modify a command that was previously used.

All commands are divided in two groups. Those starting with `agent` regard the RL agent, while the `features` commands are related to the features extractor.

Agents

Three operations can be performed for the agent: `train`, `play`, and `watch`.

To train an agent we run:

```
atarieyes agent train      # ...
```

This command has many options, some of which control the parameters of the Double DQN algorithm. We show here just the most relevant:

`-e`/`--env` Selects the environment to use, among the list of Atari games.

`-b`/`--batch` Each update of the Q-Network is computed from a cumulative gradient of this number of samples.

`-r`/`--rate` Chooses the learning rate associated to each gradient update.

`-g`/`--gamma` Selects a discount factor.

`-c`/`--continue` Resumes training from any checkpoint. Checkpoints are saved in regular intervals, according to the `--save` option, or when a training is interrupted with CTRL-C (SIGINT).

`--rb` Trains an agent with the Restraining Bolt applied. When this option is absent, the agent Q-Network is that of Section 4.3.1. When `--rb` is added, the restrained model from Section 4.3.2 is used. The argument of this command is the IP of a running Restraining Bolt; often, just `localhost`.

There are many other options which we didn't list here.

The second command related to agents is `play`. Its purpose is to load an agent previously trained and let it interact with the environment. This is certainly useful to assess the performances reached. Most importantly, this continuous play generates the stream of observations that we need in order to train a features extractor. Some options are:

`args_file` This mandatory argument is the path of the JSON file containing the exact training command of the agent.

`-c/--continue` It is a mandatory argument that says which checkpoint to load.

`--rand-eps` and `--explore-policy` These two options allow to use the two custom exploration policies that were defined in Section 3.1.3.

`-w/--watch` Used to visualize the frames of the game. Allowed arguments are `render`, to watch the images on screen while the agent plays, or `stream` to send them to another running instance. The receiver could be another instance training the features extractor model.

`--record` To save a video of the agent's performance.

Features

Commands that start with `atarieyes features` are related to the features extractor. We can train the model that was developed in Chapter 5 and use it for prediction in combination with a Restraining Bolt.

The first step is to define a set of fluents to learn, and their associated regions. The `select` command allows to easily select the regions for an environment. For example, to define regions in the Pong game, we run:

```
atarieyes features select -e Pong-v4
```

where `Pong-v4` is the precise name of the environment. After this command, a frame of the game is shown. With the mouse we can do one or more selections (press Enter to accept). The first selection is the portion of the image where the agent should be trained (allowing to cut irrelevant parts). Then, every following selection is a definition of a new region. After each selection, we insert at the terminal an unique name and abbreviation for that region.

The output generated is a JSON file containing our definitions that we can now modify and integrate. We could have written this file manually, but `select` is a convenient way to start. The file is saved at `definitions/`<env-name>`.json`. Now we can fill all the `"fluents"` fields with the list of symbols that we want to define in each region. In our example on the Pong environment, the output is shown in Listing 6.1 which contain each region name, abbreviation, coordinates, and fluents defined.

The other two empty fields are `"constraints"` and `"restraining_bolt"`. Here we write the LDL_f formulae for the temporal constraints (Section 5.1) and for the Restraining Bolt temporal specification (Section 4.2.2), respectively. The atomic

```
{
  "_frame": [ 0, 33, 160, 195 ],
  "regions": {
    "paddle_right": {
      "abbrev": "pr",
      "region": [ 131, 34, 151, 196 ],
      "fluents": []
    },
    "bottom": {
      "abbrev": "bot",
      "region": [ 0, 184, 160, 194 ],
      "fluents": []
    }
  },
  "constraints": [],
  "restraining_bolt": []
}
```

Listing 6.1. Example for the Pong game; file `definitions/Pong-v4.json`

symbols of both formulae must be among the fluents we've defined above. In this file, they are stored as lists of formulae, just to improve readability. The expressions each list will be joined through conjunction in a single LDL_f formula. Since the `constraints` are always satisfied, we consider as restraining specification the conjunction of both of these formulae.

After the definitions, we're ready to train the valuation functions. This is achieved by the `atarieyes features train`, much like we've done for the agent. Few of the many options of this command are:

`--stream` Sets the IP address of a running instance of `agent play --watch stream` (default is `localhost`). The frames received are used to train this model.

`--shuffle` Sets the size of the dataset composed by the most recent observations.

`-c/--continue` Resumes an interrupted training from a checkpoint.

`-i/--init` Starts a new training but initializes the parameters from a checkpoint.

`--train` The two arguments that follow are the name and the depth of the layer that should be trained by this command. The other parameters of the model are not modified.

`--network` Specifies the structure of the encoders. The argument of this command is a list of natural numbers. The i-th number indicates of how many hidden

units composed the $i + 1$-th layer of each DBN. Instead, the first layer has always as many visible units as the number of pixels of each region.

The features extractor that we've defined in Chapter 5 is composed by one DBN for each region (the encoders) and the Boolean functions, shared by all regions. The encoders, in turn, contain a stack of RBMs, which are organized in layers. For each region, we need to train the shallow layers first. For example as:

```
atarieyes features train -e Pong-v4 --network 20 3 --train bottom 0
```

Then, we proceed to the next layer just below (in this example, `bottom 1` is the next and last layer of this encoder). After each encoder is trained, we can proceed to train the Boolean functions with `--train all -1`. Each time we proceed to a different part of the model, we should initialize the parameters from the previous result via the `--init` option.

Many other options, which we didn't list here, allow to personalize both Persistent CD and the Genetic Algorithm. For example, we can tune how many episodes are executed when computing the fitness function.

Once every part of the features extractor is trained, we can use it to make predictions. In particular, we pass the predicted fluents values to the Restraining Bolt. With the command `features rb` we can execute a RB from the features extractor just trained. Some arguments are:

`args_file` Mandatory path of the JSON file of arguments that generated the features extractor.

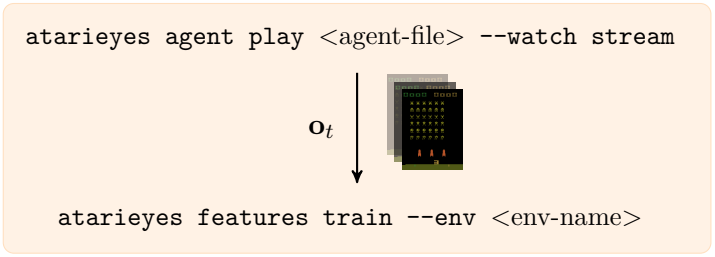
`-i/--init` Model checkpoint to load.

`--stream` IP address of the running agent to which this Restraining Bolt should be applied.

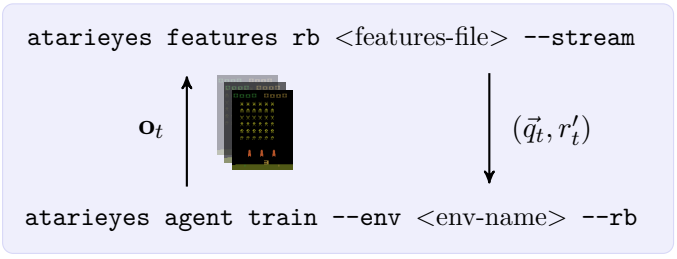
Instances

As we can understand from the arguments of the various commands, often we need to run more than one instance at the time. Figure 6.1 shows how the instances interact in each situation.

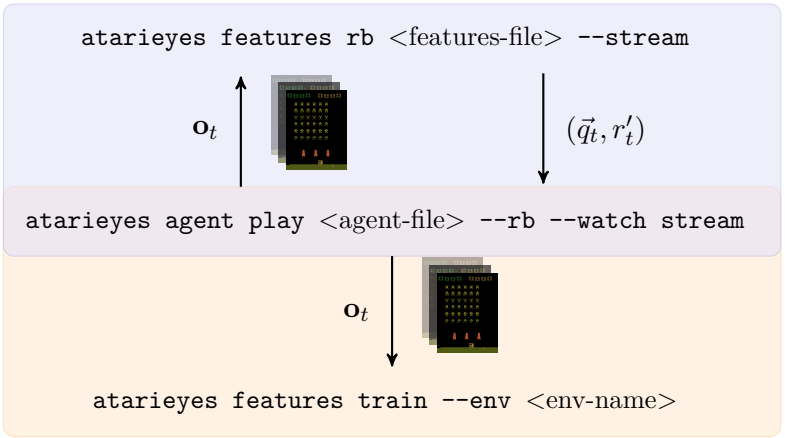
These instances use sockets to exchange observations, states and rewards. The reason for this complete separation is that the main purpose of the `atarieyes` package is to implement the Restraining Bolt and the features extractor of Chapter 5, not a RL agent. In fact, there are many modern and stable libraries implementing Deep RL agents. Thanks to this separation, we can substitute `atarieyes agent` commands with any software implementing a Deep RL agent. We don't need to



(a) Training a features extractor.



(b) Training a restrained agent.



(c) Training a new features extractor from a restrained agent.

Figure 6.1. How the various instances interact in each case.

restrict ourselves to our implementation, not even to Double DQN. In fact, it's sufficient that the agent's instance respects the interface of the socket messages. Since we also provide Client–Server classes, the integration should be immediate.

Output

To conclude this overview of the user interface, we look at the output of the various commands. As we've seen from `select`, the definitions for each environment are stored in JSON files inside the `definitions/` directory. Training commands, instead, store their result inside `runs/`. For example, `atarieyes agent train -e Pong-v4` generates the following directories:

```
runs/agent/Pong-v4/logs/0/
runs/agent/Pong-v4/models/0/
```

Every training generates "logs" and "models" directories in unique paths composed with increasing numbers. After the example above, a new output would be saved at `logs/1` and `models/1`.

Inside the "models" directory we can find all the checkpoints saved during training. These files can be passed as arguments to a `--continue` option, to load a saved agent. "logs" directory, instead, contains `args.json` and a file of TensorBoard logs. The JSON of arguments `args.json` is used for the `play` command or for the `--from` option, if we want to repeat a similar training.

The remaining files in the logs directory contain various metrics that allow to follow the training process. These logs can be visualized with TensorBoard, the TensorFlow visualization tool. In the example, we would run:

```
poetry run tensorboard --logdir runs/agent/Pong-v4/logs/
```

For each episode, we could read in TensorFlow: the number of steps, the cumulative reward, the distribution of RB states, the distribution of selected actions, and metrics related to DQN.

The output of a `features train` command is really similar to that for the agent: "logs" and "models" directories are saved under `runs/features` that contain the JSON of arguments, checkpoint files, and TensorBoard logs. Intead, the main difference is the content of the log files. Some informations that we save and can be visualized are the following:

Scalar metrics These are relevant scalar quantities that allow to follow the training algorithm. For RBM training, we store: free energy, reconstruction error, sparsity and normalization loss. Instead, for the Boolean functions, we can read: average and maximum values of the sensitivity, consistency metrics, and fitness.

Images Reconstructed most probable input images.

Graph We can inspect the graph of computations for each training command. Each inner model of the features extractor has a different training graph. For the Boolean functions, for example, we observe the four steps of the genetic algorithm.

Distributions We can observe how the model parameters are distributed on the real axis. This helps to investigate under/over-fitting and other issues. For the genetic algorithm, we visualize the population fitness values, a projected representation of the individuals, and the fluents predictions.

LDL_f library: `flloat`

Temporal logic is used for two purposes: the RB restraining specification, which indicates the agent's behaviour to reward, and the temporal constraint, used to learn the valuation functions. They are LDL_f formulae, written inside each environment JSON file of definitions.

The library that we use for parsing and transforming these formulae to DFA is called `flloat` (GitHub `whitemech/flloat`). The purpose of the package is to implement the DFA transformation for two temporal logics: LTL_f and LDL_f. In fact, we might alternatively use LTL_f with minor modifications to the source.

The initial author of this library, Favorito [8], started this project as a Python port of an homonymous software that was developed in Java[2]. Then, development has continued and, during this thesis work, we contributed to the advancements of the library. We helped to improve the overall stability of the software and to write a more efficient parsing of the input languages (the version used here is 0.3.0).

The fields `constraints` and `restraining_bolt` both contain LDL_f expressions in a string format accepted by `flloat`. We can refer to this library documentation to understand what format is accepted. Since `flloat` gets installed with `atarieyes`, we can also experiment interactively with it. For example:

```
from flloat.parser.ldlf import LDLfParser as Parser

expression = "A & [true*; A]B & <true*; ?B>tt"
formula = Parser()(expression)
automa = formula.to_automaton()
```

This is useful to check that the `expression` has the correct syntax, and that it represents the intended temporal property (it's possible to visualize the automaton

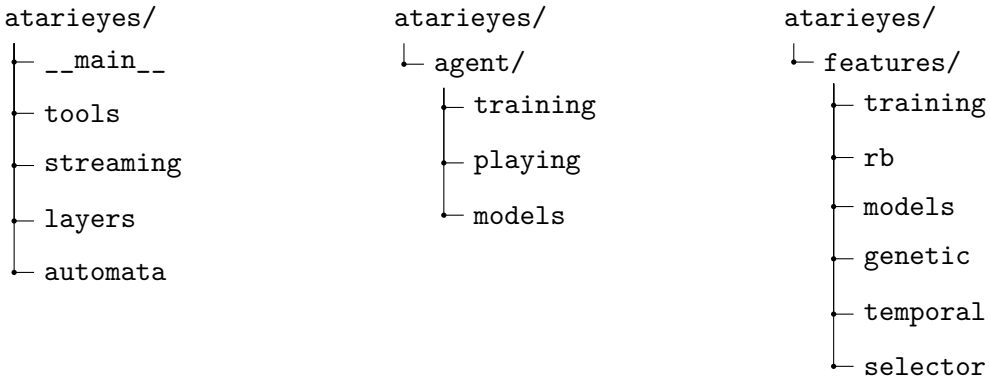
[2] GitHub `RiccardoDeMasellis/FLLOAT`.

`automa` with Graphviz). In this example, `expression` is correctly parsed as:

$$A \wedge [\, true^*; A \,]B \wedge \langle\, true^*; B? \,\rangle tt$$

6.2 Implementation

The package has a modular and comprehensible design, as we can see from the following file structure:



Each of these files is an importable Python module (extension omitted), with a separate functionality. We won't discuss all of them, as we only want to look at the most interesting details of the software.

6.2.1 `atarieyes` package

There are 5 modules inside the outer scope (left column of the file hierarchy above). `__main__.py` only realizes the command line interface and `tools.py` contains generic utilities. Instead, the remaining modules are the most interesting.

`streaming` module

This module allows the various instances of the program to communicate. It defines a communication protocol and the format of the messages to exchange. The instances communicate through sockets. So, the RL agent and the Restraining Bolt could even be on separate machines. Furthermore, this allows our implementation of the features extractor and the Restraining Bolt to communicate with any kind of RL agent, even implemented with some other library. It's sufficient that the agent program does `import atarieyes.streaming` to use this module interface for exchanging messages with the Restraining Bolt.

This file defines two base classes: `Sender` and `Receiver`. The sender is a TCP server that waits for an incoming connection from the receiver. They only realize the

basic functionality, because the specific messages format is defined in their subclasses.

`AtariFramesSender` and `AtariFramesReceiver` is a pair of subclasses that are used to send images of the Atari games when the `--stream` option is present. Similarly, `StateRewardSender` and `StateRewardReceiver` transmit the pair $(\vec{q}, r')$ containing an automaton state and a reward, from the RB back to the agent.

Users can send and receive data with `send` and `receive` methods of the respective instances in each pair. The base classes also provide transmit and receive buffers for an asynchronous exchange.

`automata` module

This module only contains one class that realizes a DFA. There are many automata libraries available. The motivation behind this class is that we need to be very efficient in our specific use case: running many copies of the same automaton. In the following, let's denote with $\mathcal{A}_\psi$ the DFA associated to the temporal constraint.

The final layer of the features extractor model is composed by an array of Boolean functions. As we've seen, this part is trained with a Genetic Algorithm, which maintains a population of candidates (each individual is an array of functions). Computing the fitness function for each individual requires to do predictions with all of them and check the generated traces against the temporal constraint. To do so, we continuously predict the fluents values with each candidate, and we move each copy of $\mathcal{A}_\psi$ accordingly. At the end of the episodes we combine the metrics to compute the total fitness function for each candidate. Since the population can contain thousands of candidates, it's important to have an implementation that allows an efficient execution of thousands of parallel copies of the same DFA.

The class `TfSymbolicAutomaton` stores the edges of the graph into a Hash table. Each key is a pair containing the current state and the input symbol; each value is the next state for that key. Moving through the automaton is a simple lookup from this table. To realize both the table and the lookup mechanism we've used the vectorial calculus of the TensorFlow library. So, with just one call, it's possible to receive the next states for any number of state–symbol input pairs.

The methods are:

```
def initial_states(self, n_instances): # ...

def is_final(self, states): # ...

def successors(self, states, symbols):
    # Lookup
    keys = self._to_keys(states, symbols)
```

```
    next_states = self.transitions.lookup(keys)

    return next_states
```

The automaton instance is state-less. The caller should maintain a vector of states, created from `initial_states` and transform it each time with `successors`.

The `symbols` argument is a vector of predictions, one for each candidate. The symbols alphabet is the set of Boolean interpretations of the fluents. This means that the space occupied by the lookup table is exponential in the number of fluents. This is not an issue, because the LDL_f-to-DFA conversion has a double-exponential time cost on the length of the constraints, which is often a much stronger requirement on the number of usable fluents.

`layers` module

In Keras, TensorFlow and any modern library, Neural Networks are implemented as a composition of layers. Although we're not required to use them, they allow a better organization of our models. This module defines the basic functionality related to layers and the specific definitions that will be used in our networks.

`BaseLayer` is the base class of any layer that will be defined. Its main role is to set some defaults and enclose the subclasses' operations in isolated namespaces. Every layer also appears as an isolated block in the TensorBoard graph visualization.

This module also contains an utility called `layerize`. This is a function decorator that can be applied to functions of TensorFlow computations. The result is a new layer class that executes the same function. This is a very quick way of converting simple state-less computations to layers. For example, suppose we have a simple function `scale_to(inputs, in_range, out_range)` that linearly scales the values `inputs` from the `in_range` to `out_range`. Applying the decorator,

```
@layerize("ScaleTo")
def scale_to(inputs, in_range, out_range):
    # ...
```

defines a new class `ScaleTo`, subclass of `BaseLayer`. We can now instantiate from this layer class:

```
rescaling = ScaleTo(in_range=(0, 255), out_range=(-1, 1))
```

The layer instance, `rescaling`, can be used as an atomic operation inside our models. Now, calling `rescaling(x)` is equivalent to `scale_to(x, (0, 255), (-1, 1))`.

The outer module also define:

- Image preprocessing layer: scaling, cropping, resizing.

- Convolution block: optional padding, 2D convolution, activation function.

6.2.2 `agent` package

The content of the `atarieyes.agent` package is used by any `atarieyes agent` command. It contains three modules: `training`, `playing` and `models`.

The Deep RL library we've selected is called Keras-RL (GitHub `keras-rl/keras-rl`) which implements the most common algorithms using the Keras deep learning library. The specific version used here is a slightly modified version (GitHub `cipollone/keras-rl`) that uses TensorFlow 2 instead of Keras[3].

`models` package

This file contains all the necessary definitions to create a `keras-rl` agent. Since we'll instantiate a DQN agent, the most important part is the definition of the agent's Q-Network.

The `QAgentDef` is just an abstract class that represents a definition of a DQN agent. Subclasses should instantiate two attributes: `model`, which is the Q-Network of that agent, and `processor`, which allows to transform the data exchanged between the agent and the environment. Subclasses also define a list of metrics that allow to follow the training process.

Two agents are defined in this module:

```
class AtariAgent(QAgentDef): # ...

class RestrainedAtariAgent(AtariAgent): # ...
```

The first of the two, `AtariAgent`, has the same structure as that of the original DQN paper [20]. The `model` is a TensorFlow `Model` with the composition of layers that we've illustrated in Section 4.3.1. Essentially, it contains by a stack of convolutional layers, followed by a final dense layer.

The `processor` is a function that can transform the observations produced by the environment, before the agent processes it. This agent uses it to combine a stack of the 4 most recent images into a single observation (see the discussion on non-Markovian rewards due to moving bodies). It also apples reward clipping and terminates episodes when the first life is lost.

The second agent, `RestrainedAtariAgent`, is used in place of `AtariAgent` when a RB is applied (`--rb` option). The `model` of the restrained agent is a Q-Network of two inputs: the image and the RB state. The structure of the net has been provided

[3]In this version, only DQN algorithm was completely migrated to TensorFlow. The other algorithms are not supported.

in Section 4.3.2. The `processor` of the restrained agent, other than performing the same operations as that of `AtariAgent`, sends and receives data from the Restraining Bolt. Every time the environment produces a new image, it sends this observation to the RB and receives a new pair of state and reward in response. These inputs are then combined with those of the environment, before passing them to the agent.

The last definitions of this module are the three exploration policies of Section 3.1.3: `RepeatEpsPolicy`, `EpisodeRandomEpsPolicy` and `ExplorationPolicy`.

`training` and `playing` packages

The `Trainer` class, inside the `training` module, is executed for any `agent train` command. On initialization, it collects all the command line parameters and the appropriate agent definition that serve to instantiate a Keras-RL agent. The agent's class is the `DQNAgent`, which is the `keras-rl` implementation of the (Double) DQN algorithm. Then, the method `Trainer.train` enters the main training loop.

Keras uses a very interesting concept: callbacks. A callback can be used to insert additional computations in specific points of the training loop. We define two callbacks: `CheckpointsSaver`, that exports the parameters values at regular intervals (see `--save` option), and `TensorboardLogger` which saves all the metrics to the log directories in a format that can be visualized by TensorBoard (see `--log` option).

The `playing` module follows a similar idea. It contains a `Player` class with a `play` function. This method enters the agent's `test` method, which is the loop of repeated play against the environment. The callbacks defined in this module are `Streamer` to send the images through an `AtariFramesSender`, and `Recorder` to save a video from these frames.

The parts of the whole software regarding features are the most efficient. We've observed that the Atari game simulator and the DQN implementation of Keras-RL are the bottlenecks of the whole training loop. We could investigate the use of other RL libraries in the future.

6.2.3 `features` package

This package is the largest portion of the software, because it implements all the models and algorithms described in Chapter 5. The files in this directory are used for any `atarieyes features` command: `training` is used when training features, `rb` executes a Restraining Bolt from predictions of trained features, and `selector` is the regions selection tool (this last module doesn't need to be discussed further). The other files in this package serve to define the model or participate at the training process.

```
while True:
    # Receive an observation
    frame, _ = self.frames_receiver.receive(wait=True)

    # Make a prediction for all fluents
    predicted = self.fluents.predict(inputs)

    # Update RB
    state, reward = self.rb.step(predicted)

    # Send RB state and reward to the agent
    self.rb_sender.send(self.states_map[state], reward)
```

Listing 6.2. Infinite loop of `Runner.run` in the `rb` module.

`rb` module

When we execute an "`atarieyes rb`" command, we start an instance that executes a RB from trained features that continuously interacts with the agent's instance. The class responsible for this interaction is called `Runner`. Its only method, `run`, enters the infinite loop shown in Listing 6.2. The procedure is simple. At each cycle: receive an observation from the environment (RL agent's instance); predict the fluents values from the image; use these values to move the Restraining Bolt; send back the RB state and reward. As we can see, this executes both the features extractor for prediction and the Restraining Bolt. Of course, in order to predict the fluents values we need to have a valuation function already trained.

The second class in this module is `RestrainingBolt` (this is the type of the `rb` object in Listing 6.2). The only important detail is that it can either create a new automaton or load one previously used. We'll see why this is important.

When the `--new` option is added to an `atarieyes features rb` command, this class creates a new DFA, by combining the JSON file fields `constraints` and `restraining_bolt` into a single formula. Then, it uses the `flloat` library to transform it into the corresponding DFA. This automaton is saved in the current `logs` directory as a file called `rb.pickle`. In future runs, if both formulae didn't change, it's important to let this class load the saved automaton.

The fist reason is efficiency, because LDL_f to DFA is a very costly operation. Even more important is that, if the RL agent has trained with one automaton, it should use exactly the same automaton when testing, or for continued trainings. The reason is that the agent has learnt to associate a state ID to a configuration of the environment. Any other permutation of the automaton IDs would be misinterpreted by the agent's policy.

It's important to precise that the current implementation doesn't perform *reward shaping* [8]. This technique produces additional rewards, even before the temporal goal is completely reached, in order to guide the RL agent (it is possible to do so, without modifying the optimal policy). Since this hasn't been implemented yet, we used `restraining_bolt` to reward the prefixes of the desired trajectory. We won't consider this detail in the following discussion, as it creates more complex temporal specifications and it would be avoided with reward shaping.

`training` module

Just like for the agent, a training command, in this case `atarieyes features train`, executes a `Trainer.train()` method. Its role is to instantiate a features extractor model according to the parameters supplied (net structure, layer to train, etc), optionally initialize the parameters from a checkpoint, and start executing the training loop. At each iteration, it receives a new observation from the agent instance, samples a new random batch and performs one optimization step with the Adam optimizer.

Also, in this training loop, at regular intervals, a `CheckpointsSaver` saves the current model parameters, and a `TensorboardLogger` exports the metrics for visualization. We care this original part more than the agent training loop. So, the metrics are particularly detailed in this case. Also, they only talk about quantities that are relevant for the layer currently trained. When we train the Boolean functions, we can visualize consistency, sensitivity and fitness; instead, when we train a RBM, we visualize the free energy and reconstruction error, for instance.

All the remaining files in this package concur for the training process or the model definition. We'll follow a bottom up description with the few remaining.

`temporal` module

Training the features requires to evaluate "how much" each candidate set of Boolean functions satisfies the temporal constraint. For this purpose, this module computes the "consistency" and "sensitivity" metrics from the received traces of predicted values.

The only class is called `TemporalConstraints`. On initialization, it converts the constraint formula to its equivalent DFA, or loads one previously computed. It's interface is composed of just two other methods: `observe` and `compute`. The former must be called at each time step. As argument it receives a matrix of Boolean values, where each row is one candidate's prediction of the fluents values. This serves to update the relevant quantities for computing the two metrics. The latter, `compute`, must called at the end of each episode. It has no arguments, but returns two arrays,

```
def compute_train_step(self, population, fitness):

    # Compute
    population = self.reproduce((population, fitness))
    population = self.crossover(population)
    population = self.mutate(population)
    fitness = self.compute_fitness(population)

    return population, fitness

def apply(self, population, fitness):

    self.population.assign(population)
    self.fitness.assign(fitness)
    self._update_best()
```

Listing 6.3. The public interface of any `GeneticAlgorithm`.

the value of consistency and sensitivity metrics for each candidate set of Boolean functions.

Both functions have been implemented with TensorFlow operations, so to ensure a parallel computation for each individual. The time required to compute the fitness function depends very weakly from the number of candidate functions, as most time is spent for the Atari simulator.

`genetic` module

This module contains the implementation of the Genetic Algorithm described in Section 5.5.2, i.e. GA applied to Boolean functions. The file is structured as follows. The abstract class `GeneticAlgorithm` implements most of the algorithm. All the subclasses that inherit from it just complete the algorithm with the few missing parts which are problem dependent.

The outer interface of any `GeneticAlgorithm` is composed of two functions. Since they are relatively simple, we report their code in Listing 6.3. The first, `compute_train_step`, receives the current pair of population of candidates and fitness values, and computes a new pair after one cycle of the algorithm. We can clearly recognize the 4 basic steps: reproduction, crossover, mutation, and fitness. `apply` receives a pair of newly computed population and fitness values and stores them into `self.population` and `self.fitness`. Instead, `_update_best` saves the individual with the highest fitness value to the variable `self.best`. To execute the algorithm, the caller can simply loop over the two instructions:

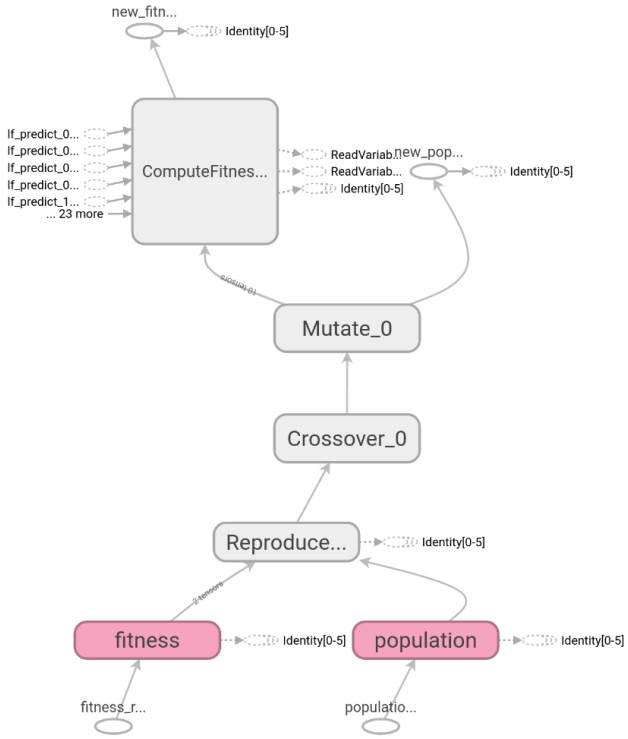


Figure 6.2. The TensorBoard visualization of the computation graph for GA.

```
population, fitness = ga.compute_train_step(ga.population, ga.fitness)
ga.apply(population, fitness)
```

The base class already defines `reproduce`, `crossover` and `mutate`, because most operations can be executed independently of the problem. Instead, subclasses are required to implement three functions: `initial_population`, which returns the initial set of individuals; `compute_fitness`, because fitness is clearly problem-dependent; and `sample_symbols`, that is used when the algorithm needs to sample new symbols (chromosomes).

For the greatest efficiency, every computation of this module has been implemented with TensorFlow operations. In fact, reproduction, sampling, crossovers can be implemented as parallel operations, which leads to a scalable algorithm for a large number of individuals. This is also the reason of the separation into `compute_train_step` and `apply`: separating most computations from assignments leads to a more efficient execution and a clearer visualization.

Every function in this class, even those defined by the subclasses, are transformed to TensorFlow layers, thanks to `layerize`. This keeps the model organized and simplifies the TensorBoard graph. We can see the general structure in Figure 6.2. The blocks we see are layers and their content can be inspected as well.

We can now proceed to apply this algorithm to learning the Boolean functions, just like we've discussed in Section 5.5.2. The class `BooleanFunctionsArrayGA` is a subclass of `GeneticAlgorithm` that defines the missing methods specifically for the problem of learning the Boolean rules.

This class defines, through `initial_population` and `sample_symbols`, a population composed of arrays of Boolean rules. Essentially, each individual is composed by the concatenation of the same Boolean rules we've shown in Figure 5.10 on page 66. We don't need to look much other details.

What is important is that this class defines a function `compute_fitness` which runs a number of episodes; computes the vector of predictions according to each individual; and uses the `TemporalConstraints` class to return the final metrics. Then, the vector of fitness values is computed as:

```
# Combine metrics
avg_consistency = tf.math.reduce_mean(consistencies, axis=0)
max_sensitivity = tf.math.reduce_max(sensitivities, axis=0)

# Compute fitness and scale
fitness = avg_consistency * max_sensitivity
fmin, fmax = self._fitness_range
fitness = fmin + (fmax - fmin) * fitness
```

The class also overrides the `_update_best` function, because the best individual is not that with the highest fitness, but the one with maximum consistency (hard constraint) with the highest sensitivity (soft constraint).

Finally, when we call the public method `predict` the class computes the most likely value for every fluent using just `self.best`. The population is only used for training.

`models` module

We've discussed both the outer training loop in the `features.training` module, and most of the necessary parts such as `temporal` and `genetic` module. We complete this description with `models`, which defines the complete structure of the features extractor model.

This file contains many classes, because the outer model is just a composition of the others inside. This should be the cleanest way to handle such a complex structure. The outer model, the only that we directly train, is called `Fluents`, which realizes the end-to-end behaviour of the features extractor both for prediction and training.

```python
class Model(ABC2):

    # This is the main model
    model = AbstractAttribute()

    # Custom training?
    computed_gradient = AbstractAttribute()
    train_step = AbstractAttribute()

    @abstractmethod
    def predict(self, inputs):
        # ...

    @abstractmethod
    def compute_all(self, inputs):
        # ...
```

Listing 6.4. The interface of every model definition.

Every class in this file is a subclass of `Model`. This is an abstract interface that just indicates what we need to define, much like we've done with `QAgentDef`. We can see its structure in Listing 6.4. On initialization, every subclass must instantiate the three abstract attributes. The most important is `self.model` which stores any container of TensorFlow operations such as `tf.Module` or `tf.keras.Model`. The other attributes can be used if we need to define a custom training loop. `computed_gradient` is a Boolean flag that indicates when the gradient can't be automatically computed from the loss function. Similarly, `train_step` should be assigned to a callable, when we need to redefine the training phase entirely. The function `predict` is used to make a prediction with the model (whatever that means for a specific subclass), while `compute_all` performs all operations required both for prediction and for training (see the class docstring for more details).

The smallest model in the whole hierarchy is the `BinaryRBM`, which represents a Restricted Boltzmann Machine with binary units. Its `model` attribute is just a layer called `BernoulliPair`, which is composed by the visible and hidden units. RBMs are not trained from some loss function, but with Persistent CD. So, `computed_gradient` is set to `true` and `compute_all` returns both a prediction for the binary units, and the gradient computed with the algorithm. More precisely, at each call, it performs a single pass of the "repeat" loop of the Algorithm 1 on page 59[4].

It's now easy to proceed by composition: the `model` of `DeepBeliefNetwork` contains a stack of `BinaryRBM`s, and the class `LocalFeatures` combines a preprocessing

[4]The gradient also includes two optional regularizations: l_2 loss and a sparsity-promoting term.

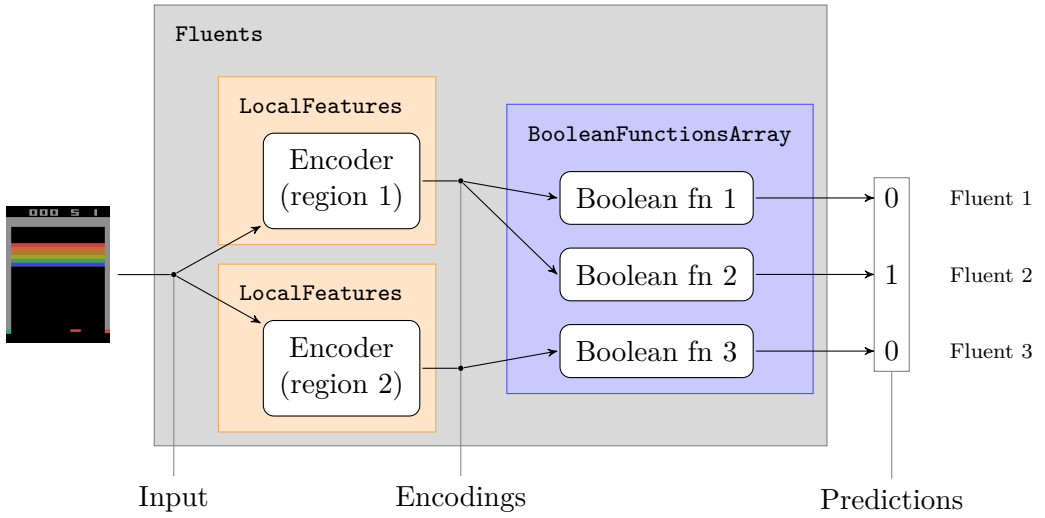


Figure 6.3. The disposition of the various classes corresponding to the general scheme of Figure 5.6 on page 54.

layer and a `DeepBeliefNetwork`. `LocalFeatures` represents the encoder model.

On the other hand, the `GeneticModel` encapsulates any `GeneticAlgorithm`, cast as other model definitions, so to use them together. `self.model` is, in this case, the population of individuals. Since the training procedure of GAs is so different, we override the training loop by defining a simple `train_step` that simply calls the GA's `compute_train_step` and `apply`.

Now that every model has been defined, we can combine them in the final arrangement. `Fluents` is the outer container for all other models. Figure 6.3 shows how the various models are composed inside the `Fluents` class. This figure directly corresponds to the scheme shown in Figure 5.6 on page 54. All the initialization parameters reflect the command line arguments. For instance, the `--network` command chooses the number of units and layers of each encoder. Also, `--train` selects which of these inner models should be trained, when we call the `Fluents`' `compute_all` function. Depending on which model we're working on, both the computation graph and the metrics saved greatly change.

Chapter 7

Esperiments

In this Chapter, we'll use the `atarieyes` software to test the effectiveness of the ideas presented in this thesis. The purpose of these experiments is both to demonstrate that a features extractor can be learnt with the method proposed, and to show how these features can be used in combination with the Restraining Bolt for complex RL tasks.

We'll look at two environments: Breakout and Montezuma's Revenge. For the first we'll test the training process of the features extractor, while the second is used for a more complete demonstration about the possibilities of the complete agent (we proceed from learnt features to the application of the Restraining Bolt).

7.1 Breakout

The first environment we'll see is the famous Atari game "Breakout". A frame of this game is shown in Figure 7.1. The goal is to hit all the bricks with the ball (the small orange dot). Every time one brick is eliminated, the environment produces a positive reward. The agent, through the four actions available, *NoOp*, *Fire*, *Right* and *Left*, can move the paddle at the bottom and direct the ball. Every time the paddle misses the ball, the agent loses a life. However, during training, we terminate and reset the episode at this event.

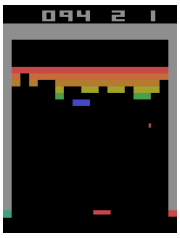


Figure 7.1. A frame from the Atari game "Breakout".

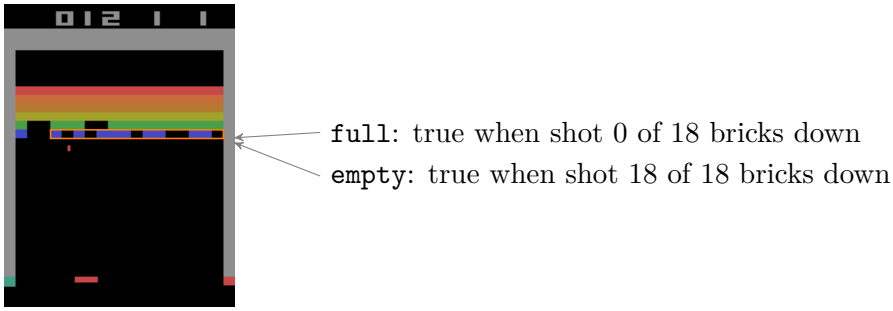


Figure 7.2. In this environment we define two fluents and one region.

The exact environment name for this game is `BreakoutDeterministic-v4`.

7.1.1 Definitions

We now define two propositional symbols, *Full* and *Empty*, and we apply the proposed model and the training procedure to learn their Boolean valuation function. The intended interpretation of these two proposition is:

Full should be true when the area is full of bricks.

Empty should be true when there are no bricks inside the area.

The area which we're talking about is shown in Figure 7.2. The orange rectangle which contains the intended set of bricks is also the fluents region. So, we've defined two symbols in one region of the image.

How could we *describe the behaviour* of these two propositions with temporal logic? When an episode starts, we know that *Full* should be true, because all bricks are present, initially. This initial condition is really helpful. Then, *Full* and *Empty* represent concepts that are always mutually exclusive. Finally, since the bricks cannot reappear, we know that the path is forced: the propositions can't return to a previous configuration. All these descriptions translate to the following LDL_f temporal constraint:

$$
\begin{aligned}
&\mathit{Full} \wedge && \rightarrow \text{initial condition} \\
&[\, \mathit{true}^* \,](\neg \mathit{Full} \vee \neg \mathit{Empty}) \wedge && \rightarrow \text{exclusive propositions} \\
&\neg\langle\, \mathit{true}^*; \neg \mathit{Full}; \mathit{true}^* \,\rangle(\mathit{Full} \wedge \neg \mathit{End}) \wedge && \rightarrow \text{can't reappear} \\
&\neg\langle\, \mathit{true}^*; \mathit{Empty}; \mathit{true}^* \,\rangle(\neg \mathit{Empty} \wedge \neg \mathit{End}) \wedge && \\
&\neg\langle\, \mathit{true}^*; \mathit{Full} \,\rangle \mathit{Empty} && \rightarrow \text{not immediately}
\end{aligned}
\tag{7.1}
$$

The conjunction $\wedge \neg \mathit{End}$ means that we're not referring to the end of the trace. This is only required in the LDL_f semantics for finite traces used here.

The DFA associated to this temporal constraint is shown in Figure 7.3. This is the automaton that the software will use to search among the candidate functions.

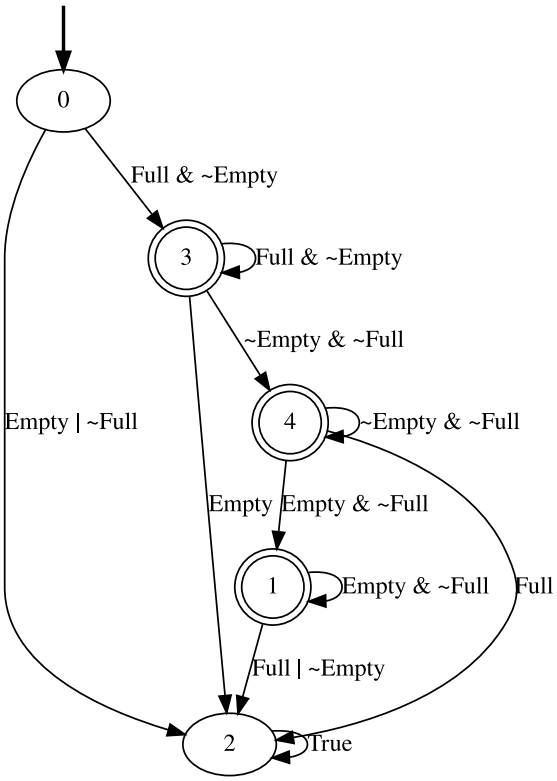


Figure 7.3. The DFA associated to the formula in Equation (7.1).

As we can see from its structure, the fluents dynamics is relatively simple. We could have also used the following equivalent formula:

$$\begin{aligned} &\langle\,(\mathit{Full} \wedge \neg \mathit{Empty})^{+}\,\rangle(\mathit{End} \vee \\ &\quad \langle\,(\neg \mathit{Full} \wedge \neg \mathit{Empty})^{+}\,\rangle(\mathit{End} \vee \\ &\quad\quad \langle\,(\neg \mathit{Full} \wedge \mathit{Empty})^{+}\,\rangle \mathit{End})) \end{aligned} \tag{7.2}$$

where the operator ρ^{+} is an abbreviation of the regular expression $(\rho; \rho^{*})$.

Most states in this automaton are final. In fact, nothing guarantees that the agent will be able to hit the bricks in every episode. The fluents might not evolve at all when the agent loses a play.

To specify all these definitions to our software we run:

```
atarieyes features select -e BreakoutDeterministic-v4
```

and we select the region of Figure 7.2. Then, we complete the generated file with the fluents names and any of the two temporal constraints above. The resulting file is shown in Listing 7.1. "`br`" is just the abbreviation for that region name. Other than that, the file exactly represents what we've defined so far.

```
{
  "_frame": [
    8, 32, 152, 197
  ],
  "regions": {
    "blue_right": {
      "abbrev": "br",
      "fluents": [
        "br_full",
        "br_empty"
      ],
      "region": [
        32, 87, 152, 93
      ]
    }
  },
  "constraints": [
    "br_full",
    "[true*](!br_full | !br_empty)",
    "!<true*; !br_full; true*>(br_full & !end)",
    "!<true*; br_empty; true*>(!br_empty & !end)",
    "!<true*; br_full>br_empty"
  ]
}
```

Listing 7.1. The content of `definitions/BreakoutDeterministic-v4.json`.

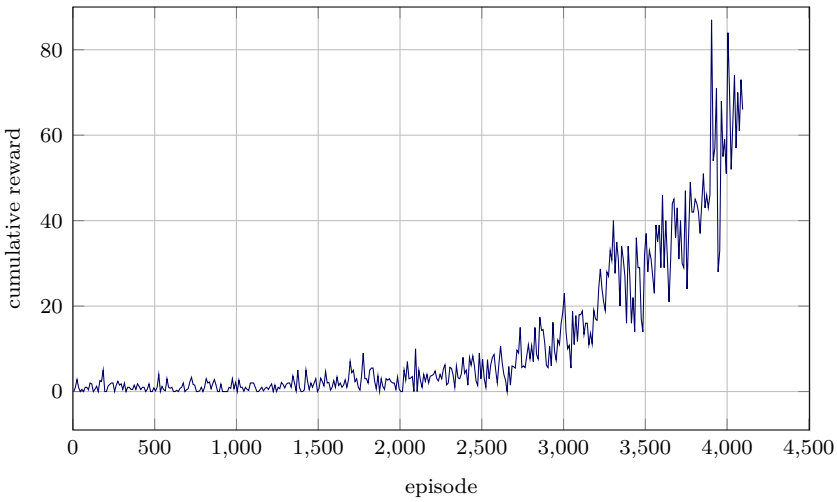


Figure 7.4. Training the RL agent. Plot of the cumulative reward in each episode.

7.1.2 Training

The first step is to train a RL agent on the environment as it is. Some of the options we've used for the `agent train` command are: a batch size of 32, learning rate of 0.0001, and double Q-Network update every 10000 steps. To follow the training progress, we run TensorBoard on the agent's log directory. The results obtained are shown in Figure 7.4.

This plot is the cumulative reward achieved in each episode. At the end of the training, the agent achieves a reward above 70, which means that is able to hit 70 bricks without ever losing the ball. This was an expected result, because at this stage, we just want to replicate the results of previous studies. The neural network needs time to adapt to the new observations reached (a frame without bricks is really different for the Q-Network). However, we stop this training at 4100 episodes, as this is not the main purpose of this experiment. These performances are, in fact, sufficient to reach states in which *Empty* becomes true. So, we're ready to train the features extractor.

The fluents are trained from a dataset generated online by a running agent. So, we start the trained agent from the previous step:

```
atarieyes agent play <args-file> -c <checkpoint> --rand-test 0.1
```

where <checkpoint> is any saved agent from the previous training. When training the feature extractor, the agent should avoid repetitive behaviours, and try to thoroughly explore the environment. This command selects a 0.1-greedy policy, but we've also experimented with many others, such as `--rand-eps`. We let this command run and focus on the receiving side.

The network that we use for the encoder is a DBN of size $(N, 50, 20)$. N is the number of input pixels in our region, but this is computed by the software and we don't need to specify it. This network contains two RBMs and generates an encoding of 20 binary units. At first sight, this encoding could seem quite large. However, we must consider that there are 18 bricks in the selected region. If we want to learn a representation for these bricks, there must be at least 18 units. 20 constitutes a nearly-optimal size for this encoding (we use 20, because the optimal might not be reachable during training). Also, we've observed that the shallow network $(N, 20)$ is not able to achieve the same performances as the deep architecture selected here.

First, we train the layer number 0, that is of size $(N, 50)$, with the following command:

```
atarieyes features train --network 50 20 --train blue_right 0
```

Other omitted arguments are: the environment, learning rate of 0.001, batch size of 50 and regularization factors.

The output of this training is shown by the plots on the left in Figure 7.5. The most important is the free energy, shown in the top left plot, that we've defined in Equation (5.10) at page 58. A low free energy means that the model is recognizing the training dataset, because a low energy is associated to a high probability for the input batches. The second metric, the reconstruction error, shows the L_1 distance between the input images and the reconstructed images (reconstructions are the most probable images under the encoding assigned for the true inputs). Minimizing the reconstruction error is not the training objective of Persistent CD, but it's useful to visualize it anyway, because, unlike the free energy, we know its correct scale and lower bound.

We can also appreciate the trained model, by looking at the quality of its reconstructions. The top row of Figure 7.6 shows three input images in our region. Each of these inputs has a different configuration of bricks. On the bottom row, we see the expected input images, given the encoding that the model associates to the inputs above[1].

Now that the first layer is trained correctly, we proceed to the second and final layer of this encoder. With the commands `--train` and `--init` we can train the next layer (the index is 1) from the weights obtained at the previous step. The results are shown in the right-hand column of Figure 7.5. We have now trained an encoder that transforms the input image for this region in a vector of 20 binary units.

[1]The expected input has a likelihood term, that is generated from the encoding, and a prior expectation, which derives from the most frequent input patterns. So, the expected input is a probabilistic prediction and it shouldn't be properly considered an input reconstruction.

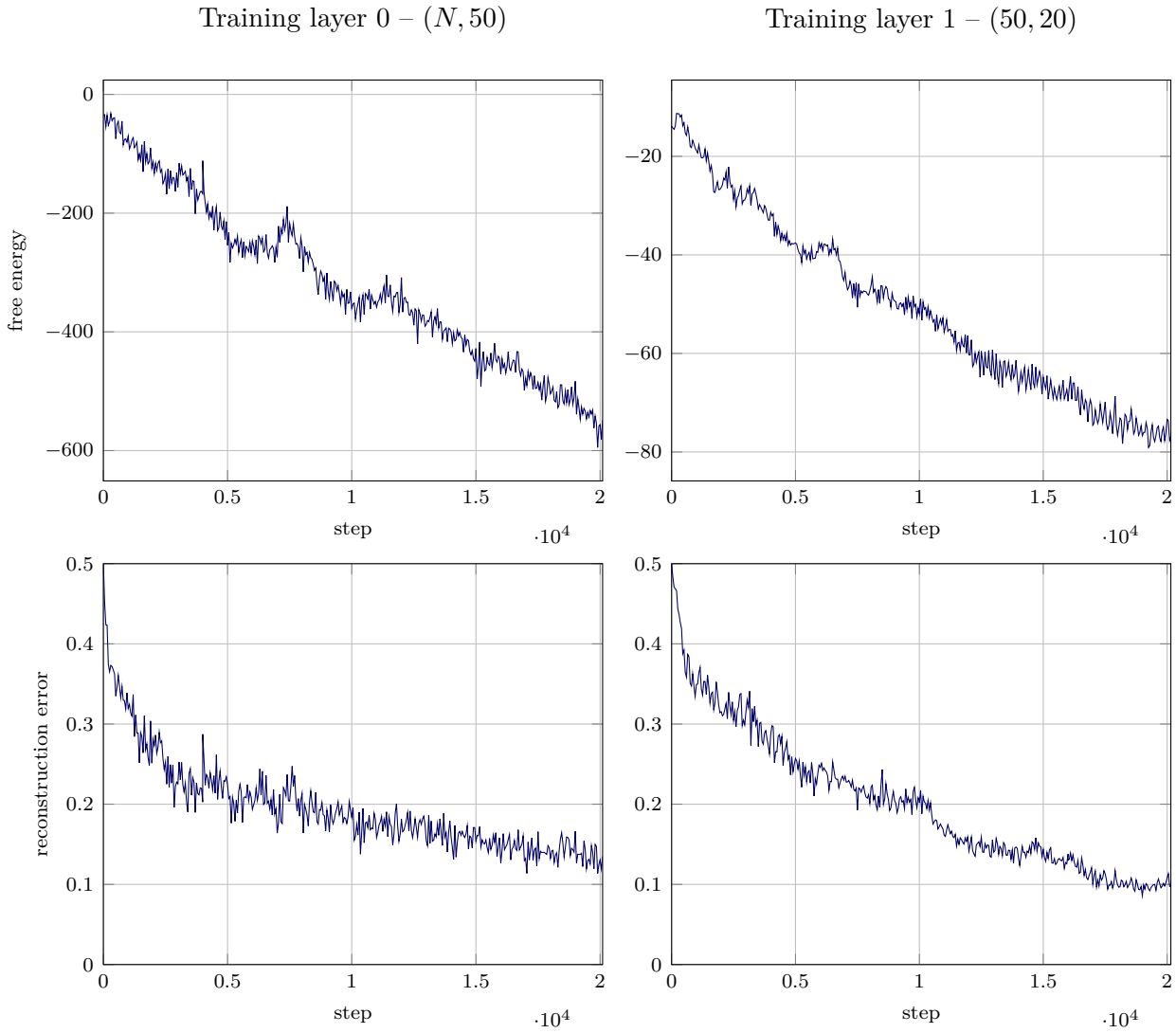


Figure 7.5. Training metrics of the encoder model.

Figure 7.6. True inputs (top row) and expected input images (bottom row). Reconstructions generated with layer 0.

Since this is the only encoder, we can now pass to discuss the Boolean functions. To train this final "layer", we issue a similar command:

```
atarieyes features train --network 50 20 --train all 2
```

For the moment, let's ignore the specific parameters used in this case. We can directly look at the training outcome in the plots of Figure 7.7. From top to bottom, they show the values of "consistency", "sensitivity" and fitness, averaged over all candidate functions. These metrics are computed with respect to the automaton in Figure 7.3. Only the fitness value contributes to the reproduction probability of each individual. The other two metrics are shown just to get a better understanding.

As we can see from the first peak in the consistency plot, the genetic algorithm, just by eliminating all the candidates that do not predict $\{Full\}$ for the initial configuration, is able to be consistent with the constraint. Then, to further improve the fitness, the algorithm looks for functions that are also able to explore the automaton states. Of course, this comes at a risk of falling into rejecting states. It seems that better candidates are found at step 220 and 260, where predictions visit all the automaton final states, always satisfying the constraint, at the end of the trace.

The colors and the vertical lines in Figure 7.7 separate different training commands. After each interruption we resume from the previous state with the `--cont` option. This detail is relevant because, as training progresses, the most appropriate hyper-parameters change. From left to right, the algorithm parameters for each run are the following:

`--fitness-episodes`	2	5	12	20
`--fitness`	(30, 100)	(30, 100)	(10, 100)	(10, 100)
`--mutation-p`	0.02	0.02	0.005	0.002
`--crossover-p`	0.02	0.02	0.005	0.002

The most important is `--fitness-episodes`, which determines how many episodes are observed in order to compute the fitness function. As training progresses, we should increase this number, because the target function should be ideally consistent with any trace. A good nondeterminism from the agent's side, helps to generate diverse test episodes and to keep this number limited. Similarly, the other parameters follow a similar idea: training should slow down and be more accurate over time. At the end of this process, a best candidate is selected according to the criterion described in the previous chapter: the individual with maximum consistency and highest sensitivity.

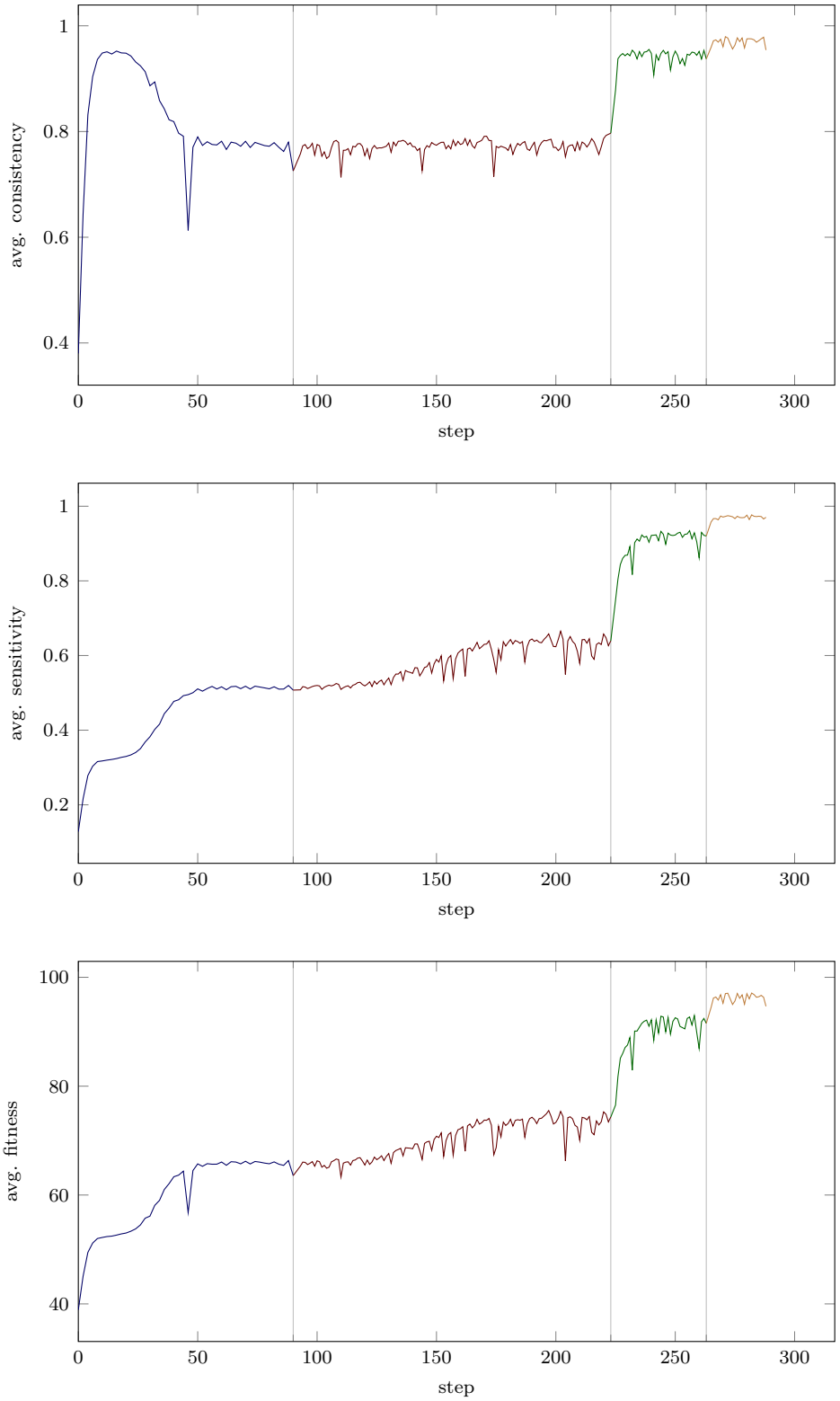


Figure 7.7. Training metrics generated by the GA for the Boolean functions: population average values for consistency, sensitivity and fitness.

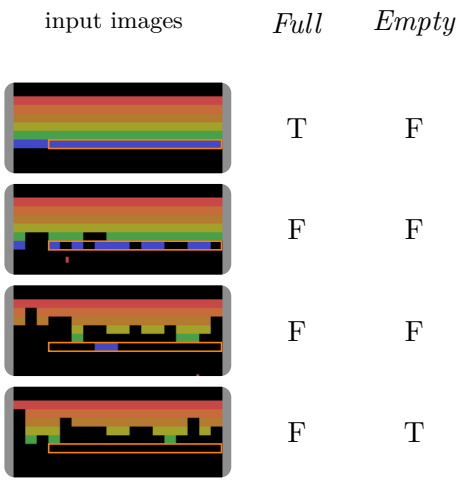
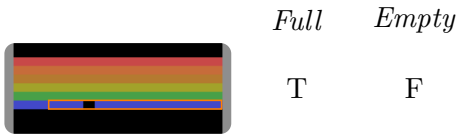


Figure 7.8. Predictions on Breakout with the trained model.

7.1.3 Comments

We've now trained a complete features extractor. From an image of the Breakout environment, it's able to predict a Boolean value for *Full* and *Empty*. To verify the quality of the result, we can just make predictions with this model. We can see few predictions in Figure 7.8. For compactness, we show just a portion of the entire frame, and the input region is highlighted in the rectangle.

In these, and many more cases that we tested, the model is always correct. The only wrong prediction we could find is the following:



where it mistakenly predicts that the region is still full of bricks. To understand this small error, we looked at the agent's plays and we discovered that it has learnt to always hit that brick at the very first touch with the ball. Let's see what this repetitive behaviour had caused.

One great advantage of working with Boolean functions from a compact encoding space is that we can inspect and understand what the model has learnt to recognize; i.e. which input patterns the model associates to a true output. To do so, we need to visualize both the meaning of the Boolean features and the Boolean rules deciding from such features.

The large image in Figure 7.9 contains 20 "rows". The i-th row in this image is the *expected input* region that the DBN predicts, given an encoding vector of zeros except for a 1 at the i-th position (we can do this backpropagation because

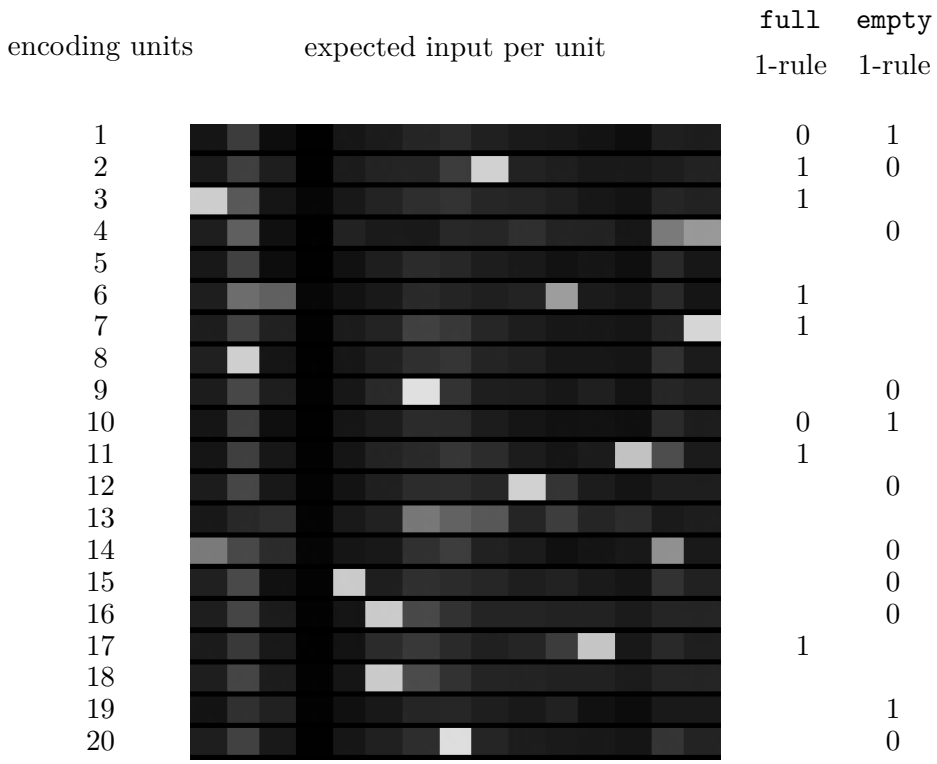


Figure 7.9. Visualization of the trained encoder and Boolean functions. Each row represents an input feature. Details are explained in the main text.

the encoder is a probabilistic model). So, each line contains the input that is most correlated with each encoding unit; essentially, what each unit represents. It emerges an interesting result: each unit is correlated to the presence of one, or at most two bricks. For example, when the left-most brick is present, the third unit of the encoder is 1, and vice versa.

The most probable inputs shown here are also affected by a prior probability. In fact, the fourth brick, which is almost always down, is the same that the agent has learnt to hit first (see the dark column). Since the fourth brick is so rare, the model didn't associate any unit to its precence, but it has only assigned a strong prior probability, instead. So, the wrong prediction we've seen above is caused by a too coarse encoding. Clearly, everything that the encoder considers of the same class cannot be distinguished by the Boolean functions.

On the right of Figure 7.9, there are two columns of 0s and 1s. These are the Boolean rules (1-rules) of the best candidate that the model has selected. We can recognize a reasonable pattern: the rule for *Full* requires the absence of the features 1 and 10, which detect the empty line, and the presence of many other features, which are associated to the presence of each brick. The rule for *Empty* does almost the opposite.

We also understand that the concept for *Full* is not learnt exactly, because the rule doesn't require anything about the second brick (feature number 8). This is not necessarily an error: we want the features extractor to give correct valuations just for the trajectories visited by the agent, not for any input. In fact, the training dataset are the observations produced by the agent, and we want the valuations function to be correct on those. In this example, the light colour of the second column suggests that the second brick is the last to be hit. At that point, the model has many other evidences that *Full* must be false (other bricks required by the rule are down, already). So, it didn't need to link the concept of the fluent to the 8th feature. In general, this is the reason why the agent's policy should have a strong stochastic component: to encounter as many possible trajectories during the training phase.

To conclude, this game is not considered a complex environment for Deep RL, because it has been solved as one of the first games. Instead, it can be a much more challenging environment, if we consider the features extraction problem. The difficulty of learning some propositions doesn't reside in the cumulative reward, but on the complexity of the observations and on the desired meaning of the propositions. This problem quickly becomes hard when we want to learn very complex concepts.

Even though our observations had very little noise (the ball passing was the only noise, in fact), we selected a region with many meaningful configurations. Due to the combinatory nature of the bricks, we've asked the features extractor to detect one in 2^{18} configurations, which is not an easy task. As we've seen, this goal as been achieved almost perfectly.

Due to the way temporal constrains and metrics work, it's better to learn some correlated fluents at the same time. Even if we would only need *Full* for some temporal goal, we wouldn't be able to learn one of the two symbols in isolation.

7.2 Montezuma's Revenge

With the previous game, we've shown that it's indeed possible to learn Boolean propositions of moderate complexity. Here, instead, we demonstrate the complete training procedure described in Section 5.6: from a completely inexpert agent, we'll obtain a capable one, that bases its decisions on the features learnt.

We'll use a second game from the Atari collection, called Montezuma's Revenge (the precise environment name is `MontezumaRevengeDeterministic-v4`). Figure 7.10 shows the initial image of this game. The agent's goal is to move the character inside the many rooms of the labyrinth, collect the items required to advance, and avoid enemies. The action space is composed by 18 actions. They

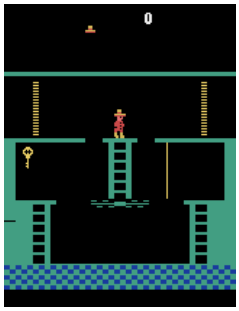


Figure 7.10. A frame from the Atari game "Montezuma's Revenge".

allow to move the agent to remain still, move left-right, up-down (and combinations of the two), optionally while performing the `fire` action.

As we can imagine from the many actions and the more complex map, this game is much harder for a RL agent with respect to Breakout. In fact, it's arguably the most difficult among all games in the Atari 2600 collection. We can see, for example, that in [19] the DQN agent achieves no rewards at all. This difficulty arise from two factors: partial observations (because of the many rooms), and sparse rewards. Luckily, both issues can be addressed with the Restraining Bolt, because it can be used to provide additional rewards when specific trajectories or simple conditions are reached.

7.2.1 Definitions

The problem of sparse rewards is the first difficulty the agent faces. In fact, reaching the first rewarded state, which is taking the key of Figure 7.10, requires a long sequence of correct actions that is almost impossible to achieve at random.

The fluents that we'll define serve for this purpose: correctly navigating this first room. Four of them, *AtStart*, *AtStairs1*, *AtStairs1bot*, *AtStairs2*, should be valuated to true if the character is at each of these positions. We can see the regions to which they refer, in Figure 7.11. The last one, *KeyTook*, should be true when the key has been taken by the agent.

We can clearly use these symbols to guide the agent through the room, rewarding it at each spot reached. When it grabs the key, the environment sends a reward, so that of the RB is redundant. However, in order to progress, the agent has to get the key and come back to the start position. So, *KeyTook* is required in order to write the correct temporal specification.

As we've seen in Section 5.6 "Training and incremental learning", we can't directly learn the fluents that the agent is not able to influence. In this case, we can't learn what it means *KeyTook* until the agent is able to grab it. So, only *AtStart* and *AtStairs1* will be learnt, initially. Then, every time the agent becomes able to reach

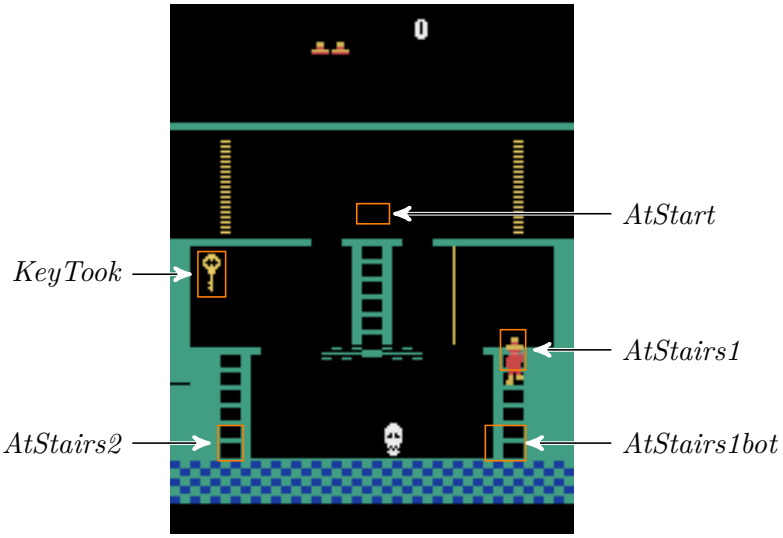


Figure 7.11. Definition of five fluents and their associated regions. The correct valuation for this frame would be: {*AtStairs1*}.

a new position, we can expand our definitions to include a new symbol to learn.

We'll now show the complete JSON file of definitions, but, for brevity, we'll omit the symbol *AtStairs1bot*. The file is shown in Listing 7.2. The `restraining_bolt` section is empty here, but it will be filled later on. Other than the initial condition, this constraint states that the "*At*" propositions are mutually exclusive, because the agent can be only in one place at the time. Also, at least one instant is needed to pass from one place to the other.

7.2.2 Training

Following the same procedure as in the previous game, we should train the RL agent alone on the environment rewards. However, as previously studies had found, the agent can't reach any reward at all, here. This is fine. The first agent is used just to train the features extractor. The random policy will be enough to learn the first two symbols: *AtStart* and *AtStairs1*. So, we run:

```
atarieyes agent play <args-file> -c <checkpoint> --rand-test 1
```

which plays completely with random actions.

The encoder network we chose is a DBN with size $(N, 10, 1)$. This means that each input region is encoded into a single Boolean value. Let's discuss this choice: a scalar encoding means that the fluent value will be exactly the extracted Boolean feature. Therefore, the Boolean functions play no role at all in the selection of the desired concept (they can only negate the received value). This is intentional. As we've previously noted, the encoding size should be the smallest number of units that

```
{
  "_frame": [ 0, 47, 160, 180 ],
  "regions": {
    "start": {
      "abbrev": "start",
      "region": [ 74, 79, 87, 87 ],
      "fluents": ["start_at"]
    },
    "stairs1": {
      "abbrev": "stairs1",
      "region": [ 131, 129, 141, 145 ],
      "fluents": ["stairs1_at"]
    },
    "stairs2": {
      "abbrev": "stairs2",
      "region": [ 19, 167, 29, 181 ],
      "fluents": ["stairs2_at"]
    },
    "key0": {
      "abbrev": "key0",
      "region": [ 11, 98, 22, 116 ],
      "fluents": ["key0_took"]
    }
  },
  "constraints": [
    "start_at & !key0_took",
    "[true*](start_at -> !(stairs1_at | stairs2_at))",
    "[true*](stairs1_at -> !(start_at | stairs2_at))",
    "[true*](stairs2_at -> !(start_at | stairs1_at))",
    "[true*; start_at]!(stairs1_at | stairs2_at)",
    "[true*; stairs1_at]!(start_at | stairs2_at)",
    "[true*; stairs2_at]!(start_at | stairs1_at)"
  ],
  "restraining_bolt": []
}
```

Listing 7.2. The content of `definitions/BreakoutDeterministic-v4.json` (symbol `stairs1bot_at` omitted for brevity).

are sufficient to determine the truth of the propositions. If, with a single unit, the encoder is able to isolate the required feature by itself, there's no point in transfering this task to the Boolean functions, where the separation might be more complex.

The training command for the first layer of the encoder at the "start" region is:

```
atarieyes features train --network 10 1 --train start 0
```

We repeat this process for layer 1 and the encoder at region "stairs1". All these runs are much faster than in the Breakout case, because the input images show little diversity. On the other end, the event we're trying to catch is very rare (the agent needs to pass on each region by chance), so we use a slightly larger batch size, with 100 images. Figure 7.12 shows the reconstruction error for each of these four trainings. We omit plots for the last layer, the Boolean functions.

The features extractor obtained correctly valuates both symbols, *AtStart* and *AtStairs1*, in all images that we tested. Let's see what the model has learnt. In Figure 7.13, we visualize the expected input images given the output encodings, in a similar way to Figure 7.9. In this case, the encoding is composed of just one Boolean unit. So, we show the input associated to both values.

The left images are the expected inputs when the encodings are 0, while the right are associated to 1. Since the character is rarely inside the regions, there is a strong bias that tend to uniform all these images. So, we've slightly emphasized the differences between the two columns to improve this visualization. As we can now understand, the encoders have learnt to recognize when the agent is *not* inside each region. The slight gray tones in the left column mean that there is some possibility for the agent to be there, when the encoding is 0; while it is certainly not there for the output 1. In fact, if we then look at the Boolean rules, we would see that this encoding is negated to produce the fluent valuations.

The now complete features extractor can be used to guide the agent with additional rewards. We write the following simple temporal goal:

$$\langle\, (\neg \mathit{AtStairs1})^* \,\rangle (\mathit{AtStairs1} \wedge \mathit{Last}) \tag{7.3}$$

which sends a single reward when the agent reaches the stairs on the right (see the regions in Figure 7.11). We write this formula on the field `restraining_bolt` in the file of definitions. If we remember that the agent wasn't able to reach any reward at all, this initial goal is a way to start.

We now train the RL agent on this goal by executing both `agent train` and `features rb`. Once we see that the agent has learnt to reach this first goal, we can repeat the previous process for a new symbol. In this case, we would let the trained agent play, and train the features extractor for the symbol *AtStairs1bot*, which is

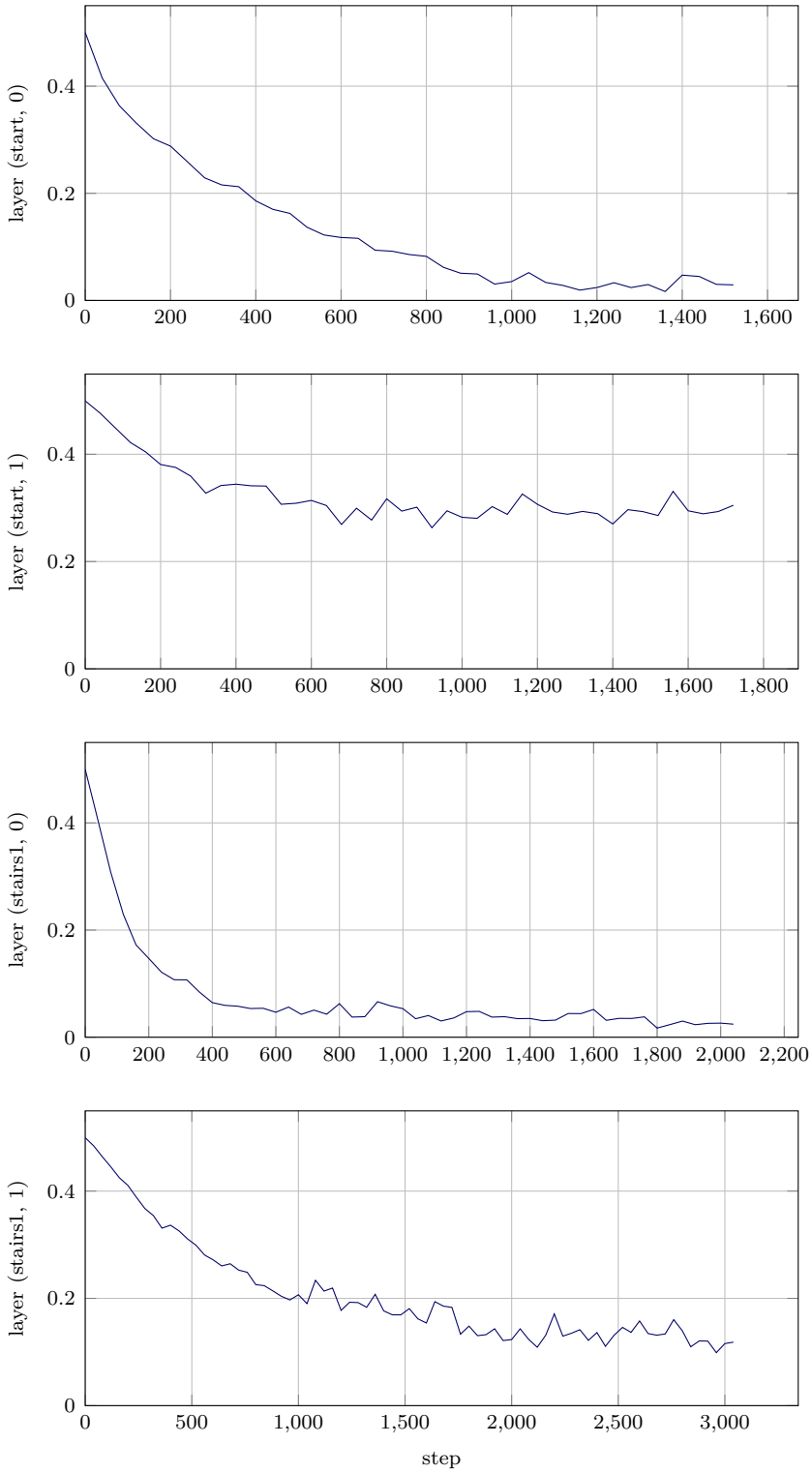


Figure 7.12. Training metric for each encoder and layer. The x-axis is the training step, the y-axis is the reconstruction error.

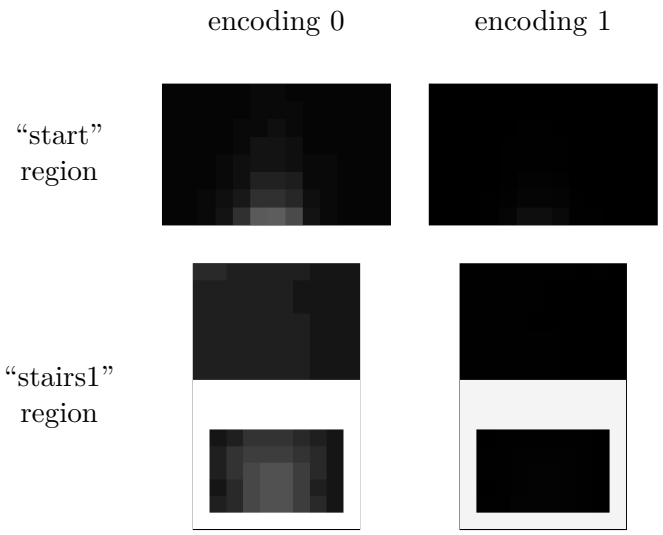


Figure 7.13. Expected input images for each region and encoding output. The differences between columns have been emphasized.

the next encountered the path. With this iterative procedure we've been also able to learn *AtStairs2*.

To this point, we've mainly addressed the problem of sparse rewards, because the Restraining Bolt has been used to send rewards upon the detection of some events. However, we can also use it to solve non-Markovian tasks. Suppose we want the agent to repeatedly jump from “start” to “stairs1” and vice versa. This apparently simple problem cannot be solved just providing the appropriate rewards, because, from one input frame, the policy hasn't enough information about where to direct the player. Instead, the RB is able to provide a consistent information about the next position to reach.

In the `restraining_bolt` field, we now write a LDL_f formula, that, combined with the temporal constraints, corresponds to the automaton in Figure 7.14. What this specification says is that the agent can accumulate rewards, represented by the final states, by following the cycle of states: $3 \rightarrow 1 \rightarrow 4 \rightarrow 5 \rightarrow 3$. So, it is induced to pass from “start” to “stairs1” and vice-versa. This is only possible, because the agent also receives from the RB in which of these states it is.

Seen as an image-to-action function, the agent's policy corresponding to states 4, 7 and 5 will tend toward the position “start”, while for 1, 6 and 4, it will point in the opposite direction. We run the usual `agent`–`features` pair of instances to train the agent on this new goal. In Figure 7.15, we show the the cumulative reward during this training. The increase we see means that the agent is learning to get from one position to the other, multiple times, in the same episode. The maximum of 14 means that it touches both positions 7 times. In Figure 7.16, we show the

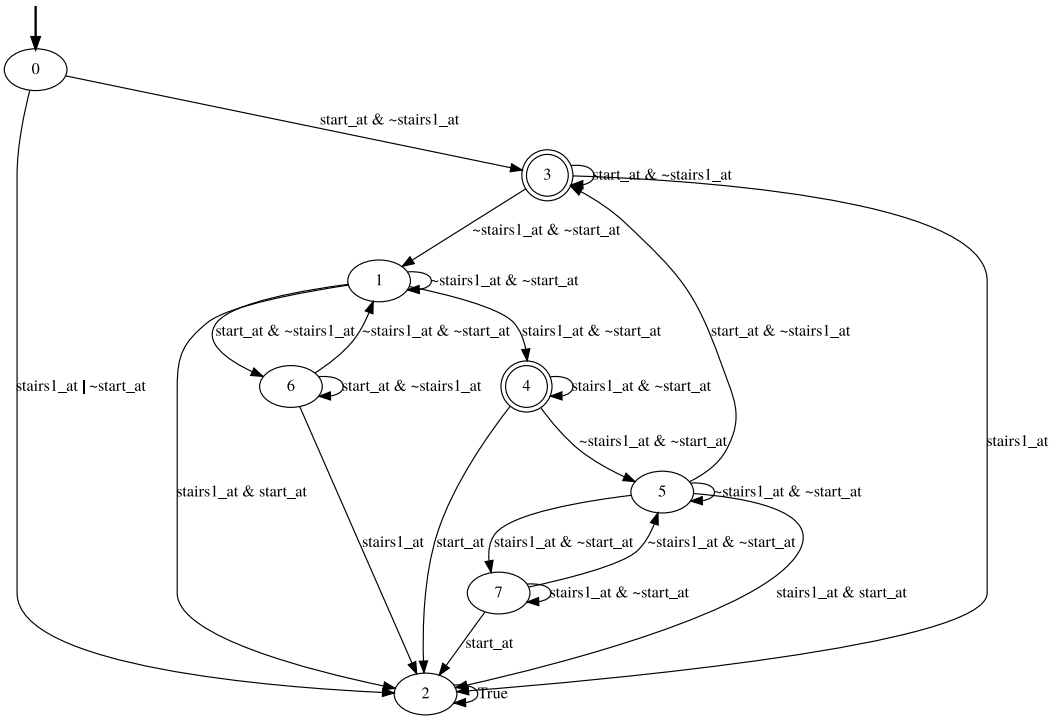


Figure 7.14. Automaton associated to the last temporal goal (back and forth from "start" to "stairs1").

.

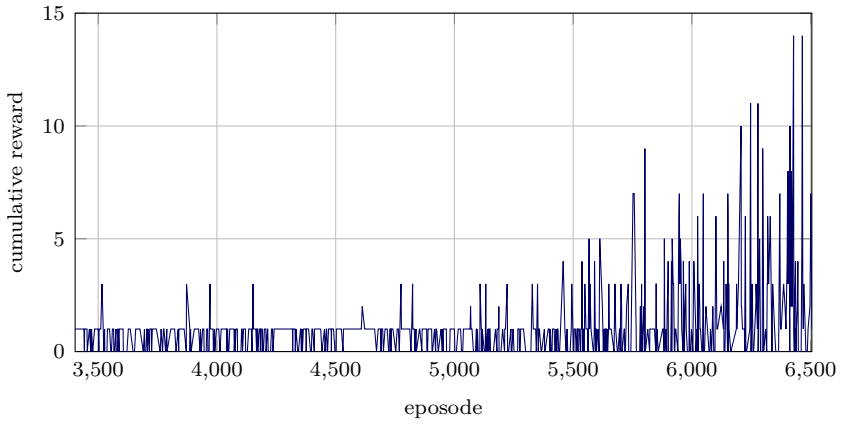


Figure 7.15. Training the RL agent on the "start"–"stairs1" goal. Plot of the cumulative reward in each episode.

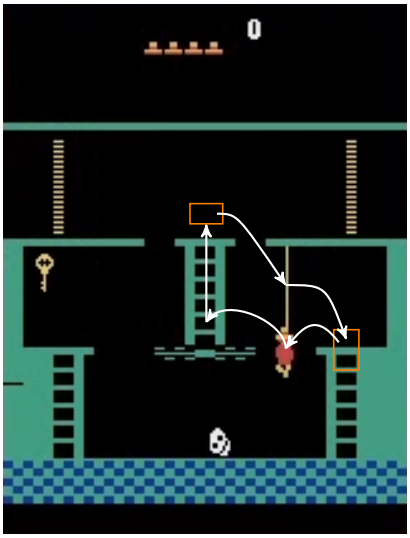


Figure 7.16. Path followed for the "start"–"stairs1" goal.

approximate path that it usually follows.

7.2.3 Comments

In this second environment, we've shown how to interleave two processes: training the RL agent and training the features extractor. We gradually improved the agent's performance until it had been possible to reach a satisfactory set of symbols. Then, we used this set of fluents to declare a non-Markovian task, and we've shown a successful training and play with the Restraining Bolt.

In this small section, we'll make few observations about these experiments on the Montezuma's Revenge game. First, we highlight some interesting details:

Rare events Although the symbols represented very simple conditions, we requested encoder to recognize very rare events. This has been possible thanks to Persistent CD, because it focuses on encoding the input distribution, not finding the accurate input reconstruction.

Long training Restrained RL agents require longer trainings with respect to the agent without the Restraining Bolt. Often, this effect isn't caused by the additional parameters that are required. Instead, this is needed by the more complex policy that we want to learn. Q-values have now a more complex landscape that requires to sample pairs of (input images–output values) for each automaton state, because the best action can be radically different for each of those. Unfortunately, sampling cannot be performed uniformly, since the agent has to reach those state. So, rare configurations will slow down the training.

Regarding the necessary improvements to this example:

Too many actions With the same training time, as the number of action increases, the quality of the policy decreases, because sampling becomes less efficient. 18 distinct and independent actions may be too much. Instead, we could try to exploit the fact that some of them are just a composition of a smaller group of actions, such as: *Left* + *Up* + *Fire*.

Explore the map! We have all the symbols we need to write a temporal specification that asks the agent to reach the key, then leave this first room of the game. This would be a more interesting challenge to try. Instead, in order to navigate the other rooms of the maze, we would need to slightly modify the implementation to remove fixed regions. This idea is introduced in Section 8.3 of "Future works".

Chapter 8

Conclusions

8.1 Overview

With this thesis, we addressed the problem of applying a method, called the "Restraining Bolt" [6], to Deep Reinforcement learning. This necessity is caused by the known limitations of RL, that Deep RL directly inherits: the Markovian assumptions on rewards an observations. Full observability, in particular, is always falsified in the real world, and even in some video games of moderate complexity. Instead, this logic construction, inspired by [1], allowed us to solve RL environments with sparse rewards, and a large class of tasks with partial observations and non-Markovian goals. This means that we're starting to address the group of problems where RL and Deep RL always struggled.

However, any form of logic reasoning works by manipulating an abstraction. In the case of the temporal logics used by Restraining Bolt, we're talking about a set of atomic propositions, that represent the state of some meaningful conditions in the outside world. Creating this connection, between environment observations and true atomic propositions, is clearly not an easy task. The more complex the observations become, the harder is to *ground* the truth of such symbols to the environment states. This is the major limitation that impede the application of some high-level reasoning to Deep RL.

By slightly extending the concepts of "observation" and "proposition", any decision problem can be stated as the issue addressed here: the problem of deciding the truth of a propositional atom. So, after the general frame in which this problem has been introduced, we must remember that we've only considered a limited class of observations and propositions.

Our observations are images of the video games of the collection Atari 2600. Propositions are conditions on visual features on the image, such as the presence of an object at some location, or a certain pattern of a visible feature. Limitations and assumptions regarding propositions have been listed thorough Chapter 5.

8.2 Main contributions

In this section, we summarize the main contributions of this thesis. Most of them have been motivated by the need of a successful application of the Restraining Bolt to Deep Reinforcement Learning. The Restraining Bolt has been previously demonstrated only in combination with RL. Moving to more complex observations required to devise many original solutions that we list here:

Model for a deep restrained agent

Deep RL agents trained with value-based algorithms rely on Q-Networks. When the agent plays in combination with a Restraining Bolt, it receives the image from the environment and the RB state. How should be combine such different inputs in the agent's network? The model proposed in Section 4.3.2 uses the RB state to select the appropriate parameters to use among a set. Although only a portion of the network has been separated, this allow to produce radically different policies for distinct RB states.

Learning fluents with temporal constraints

Chapter 5 proposed original ideas to solve the problem of learning valuations for some propositions. We used formulae of temporal logics to express the expected temporal behaviour of the desired valuations. The known limitations of this approach have been discussed in Section 5.7, but we also provided experiments to shown that these ideas can be effective in the class of problems under consideration.

Models for the features extractor

With "features extractor" we called the complete function that from an image of the game returns the fluents Boolean values. This thesis also proposed a model for this function, which combines a part of unsupervised learning, a DBN network, and an array of Boolean functions. This last part is inspired by the area of concept learning. This double arrangement addresses the known limitations of temporal constraints. Furthermore, as the visualizations in the Experiments chapter have shown, the output encoding of the DBN can be understood by the human observer. The final part of the model might also be easily be defined by hand, if we desire.

The `atarieyes` package

We provide an implementation for all the ideas presented in this thesis. The software `atarieyes` is a Python package, documented in a clear way. Most operations can be performed just from the command line interface it provides, with many tunable training parameters. Its internal structures can be also

adopted in future applications, thanks to the modular design of the package. The most interesting feature is that the functionalities related to the fluents are completely separated from the RL agent. We could even train a feature extractor or apply the Restraining Bolt with agents realized with other Deep RL libraries.

We conclude with two minor contributions that are worth mentioning. First, we worked to the recent improvements to the `flloat` Python library, together with the original author [8]. This is package is currently used by researchers for LDL_f-to-DFA translations, and it's the one we adopted in this thesis. Together, we worked for a more stable version of the software and an improved language parser for LDL_f.

Lastly, we'd like to emphasize that Chapter 4 presented RL problems with partial observations and temporally extended goals, with a unified approach. We've shown some similarities between the two, and started to illustrate why it is possible to adopt the Restraining Bolt, which is devised for the latter group, also for some problems of the former class. We've discussed that if it can be successfully applied, the RB is a clever way to define the extended state space which include the relevant history-related informations.

8.3 Future works

Features extraction is clearly a central topic of this work. We believe that temporal logics can be also used to describe the desired meaning of our symbols. Still, we don't think to have satisfactorily demonstrated this possibility. In the future, we'd like to remove some of the most limiting assumptions taken in this thesis, so to allow more interesting possibilities. Aware of the current limitations of this work, we conclude the thesis by suggesting some interesting directions for future research:

Removing regions One simplification we took is valuating fluents on fixed portions of the input image. Clearly, this strongly restricts the class of observations that can be handled, since it requires mostly static elements in the scene. We should definitely avoid this. One possibility is to define regions not on fixed locations, but on specific visual features of the image. The idea is first to find the relevant feature, then evaluate the symbol on a portion of the input at that moving position.

Removing metrics The "consistency" and "sensitivity" metrics have been used to rank candidate function and pick a solution. However, both of them induce a rather arbitrary ordering over candidates, which might not suitable for all problems. They are a form of heuristics, biasing our search. Instead, we

should only check for satisfaction between the traces and the constraint. The reason for their adoption is that, as we've discussed in Section 5.7, temporal constraints are very weak indications when used alone. We should search alternative and more correct ways to *ground* the model predictions to the desired valuations.

Strong grounding We believe that this last issue is the most important to discuss. Temporal constraints are just weak indications of the desired input-output associations. Just by talking in some abstract language, we can't hope to reliably bind the abstraction we're using to the external world states. These states must appear inside the specification, somehow. We propose to investigate two possibilities:

- The first is to include in the temporal specification some *grounded* fluents. Grounded fluents are propositional symbols which are exactly valuated. They might come from events that the agent already knows, such as the action just performed, or from conditions reliably valuated by some ad-hoc Neural Network.
- The second is to include a very small dataset of few input-output labelled samples. These very few examples can specify desired valuations which would be hard to describe otherwise.

We strongly believe that merging temporal constraints with more classic sources of informations like the two mentioned above would create very more specific and effective descriptions of the desired concepts.

Experiments The last, and more obvious, necessity is to test these improvements to see what can be achieved. We believe that these changes would open many possibilities. Also the original idea hasn't been satisfactorily demonstrated just from the experiments shown in this thesis.

The problem we tried to addressed in this work is a central topic. This isn't something related just to the Restraining Bolt method, but with any form of logic reasoning on Deep RL agents. Even in presence of a Markovian task, which might be solved with a pure Deep RL agent, there might be the necessity to look into the agent policy. Policies on images are really hard to visualize and understand, because they don't process structured representations, which are much clearer for humans. Being able to create reliable abstractions is an important goal that would also help to us to better understand the agent's purposes and strategies.

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 and Giuseppe De Giacomo. "Regular Decision Processes: A Model for Non-Markovian Domains". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. July 2019, pp. 5516–5522. DOI: 10.24963/ijcai.2019/766.

[4] 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.

[5] 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.

[6] 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.

[7] Marco Favorito. *FLLOAT*. Version 0.3.0. URL: https://github.com/whitemech/flloat.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets". In: *Neural computation* 18.7 (2006), pp. 1527–1554.

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

[16] 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.

[17] Nicolas Le Roux and Yoshua Bengio. "Representational power of restricted boltzmann machines and deep belief networks". In: *Neural Computation* 20.6 (2008), pp. 1631–1649. ISSN: 08997667. DOI: 10.1162/neco.2008.04-07-510.

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

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. 2018. ISBN: 9780262039246.

[26] 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.

[27] 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.

[28] 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.

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