
SAPIENZA
UNIVERSITÀ DI ROMA

# LTL2Action: Generalizing LTL Instructions for Multi-Task RL

## Reasoning Agents 2020/21

Roberto Aureli 1757131
Giulia Castro 1742813
Gianmarco Fioretti 1762135

Faculty of Information Engineering, Informatics and Statistics
Department of Computer, Control and Management Engineering

Master's Degree in Artificial Intelligence and Robotics

July 2021

# Contents

# Chapter 1

# Introduction

Reinforcement Learning (RL) algorithms can learn robust policies on specific, well-defined tasks. However, they usually fail when temporally extended behaviors, including conditionals and alternative realizations, are needed to solve the required task. Our work explores the problem of instruct RL agents to learn temporally extended goals in multi-task environments. Indeed, many of the existing methods can be considered *myopic* approaches because they try to decompose instructions into smaller subtasks solved independently without considering the following subtasks and generating policies that are not optimal for the entire desired task. Instead, an optimal policy must consider the whole history of states and actions since the reward function is non-Markovian.

Following the idea introduced in "LTL2Action: Generalizing LTL Instructions for Multi-Task RL" (Vaezipoor et al. 2021) we use the compositional syntax and the semantics of linear temporal logic (LTL) for specifying multiple tasks. Then we exploit a technique called LTL progression to allow the agent to identify the portions of the whole given instruction that remain to be addressed. We use an LTL-Module for embedding LTL progressed instructions while an Env-Module processes the observations, then both of them are fed in input to a *Proximal Policy Optimization* (PPO) based RL-Module, which takes actions in the environment. In addition, we propose a Graph Attention mechanism for embedding LTL formulae, which can provide the same effective results with a lower overhead. We use gym-minigrid environment to show as the proposed RL agent can successfully learn *partially ordered* and *avoidance* tasks and well generalize on unseen instructions.

# Chapter 2

# Linear Temporal Logic (LTL)

In order to instruct RL agents using LTL formulas, we first define a domain-specific set of propositional symbols $\mathcal{P}$, that represent high-level events or properties of the environment the agent can detect, then, we use LTL to compose these events into temporally-extended instructions.

## 2.1 LTL Syntax

LTL is a very understandable formal language, originally invented for describing natural language. Essentially, it extends classical propositional logic with atomic propositions $p \in \mathcal{P}$, i.e. properties of the system that are true at a certain instance of time, and Boolean operators with a set of temporal operators. The Boolean operators are $\neg$ (*not*), $\wedge$ (*and*), $\vee$ (*or*), $\rightarrow$ (*implication*). The temporal operators are the unary operators $\bigcirc$ (next), $\Box$ (always), $\Diamond$ (eventually), and the binary operator U (until).

| | |
|---|---|
| $p$ | (atomic proposition) |
| $\top$ | (true) |
| $\bot$ | (false) |
| $\neg\phi$ | (complement) |
| $\phi \wedge \psi$ | (conjunction) |
| $\phi \vee \psi$ | (disjunction) |
| $\bigcirc\phi$ | (next) |
| $\Box\phi$ | (always) |
| $\Diamond\phi$ | (eventually) |
| $\phi \mathrm{U} \psi$ | (until) |

## 2.2 LTL Semantics

The truth value of an LTL formula is evaluated over an infinite sequence $\sigma = \langle \sigma_0, \sigma_1, \sigma_2, ... \rangle$, of truth assignments for the atomic propositions in $\mathcal{P}$. Each $\sigma_i$ is a propositional interpretation describing which properties are true at a certain current state $i$. We say $p \in \sigma_i$ for a proposition $p \in P$ to indicate that $p$ is true in $\sigma_i$. The semantics for the classical operators is:

- $\langle \sigma, i \rangle \vDash p$ **iff** $p \in \sigma_i$, where $p \in \mathcal{P}$
- $\langle \sigma, i \rangle \vDash \neg\phi$ **iff** $\langle \sigma, i \rangle \nvDash \phi$
- $\langle \sigma, i \rangle \vDash (\phi \wedge \psi)$ **iff** $\langle \sigma, i \rangle \vDash \phi$ and $\langle \sigma, i \rangle \vDash \psi$
- $\langle \sigma, i \rangle \vDash (\phi \vee \psi)$ **iff** $\langle \sigma, i \rangle \vDash \phi$ or $\langle \sigma, i \rangle \vDash \psi$
- $\langle \sigma, i \rangle \vDash (\phi \implies \psi)$ **iff** $\langle \sigma, i \rangle \vDash \phi$ then $\langle \sigma, i \rangle \vDash \psi$

For the temporal operators we have:

- $\langle \sigma, i \rangle \vDash \bigcirc\phi$ **iff** $\langle \sigma, i+1 \rangle \vDash \phi$

  the formula $\bigcirc\phi$ holds at a certain time step $i$ if $\phi$ holds at the next step $(i+1)$

- $\langle \sigma, i \rangle \vDash \Diamond\phi$ **iff** there exist $j$. $(j \geq i) \wedge \langle \sigma, j \rangle \vDash \phi$

  the formula $\Diamond\phi$ holds at a certain time step $i$ if in a finite number of time steps $\phi$ will hold, i.e. soon or later $\phi$ will hold.

- $\langle \sigma, i \rangle \vDash \Box\phi$ **iff** for all $j$. if $(j \geq i)$ then $\langle \sigma, j \rangle \vDash \phi$

  the formula $\Box\phi$ holds at a certain time step $i$ if $\phi$ holds now and $\phi$ holds forever in all the next time steps, i.e. from now on $\phi$ is true.

- $\langle \sigma, i \rangle \vDash \phi U \psi$ **iff** there exist $j$. $(j \geq i) \wedge \langle \sigma, j \rangle \vDash \psi \wedge$
  for all $k$. $(1 \leq k \leq j) \implies \langle \sigma, k \rangle \vDash \phi$

  the formula $\phi U \psi$ holds at a certain time step $i$ if eventually $\psi$ holds and before that $\phi$ will always be true, i.e soon or later $\psi$ will be true and until it $\phi$ will be true.

In our experiments we are particularly interested in using this logic as a specification language for checking two particular types of dynamic properties evolving over time: (1) *Liveness* and (2) *Safety*. (1) *Liveness* means that eventually, soon or later at a certain point something good will happen. For instance, properties such as "eventually reach this point" or "visit these points in a particular sequential order". (2) *Safety* properties describe something good that always

reamains true. It is usually used in the negative fashion to express that something bad will never happen. For example "always stay safe" or "never touch a blue square".

## 2.3 Co-safe LTL

While LTL is interpreted over infinite traces, many of the task specified as LTL formulas can be verified in a finite number of steps. We refer at them as *co-safe* LTL formulae. For instance, the formula ◊G (*eventually* green) is satisfied by any infinite sequence where G is true at a certain point. Therefore, as soon as G occurs in a finite number of steps, we know that ◊G is satisfied. Instead, a formula like □¬B (*always not* blue) or ¬◊B (*not eventually* blue) which means "it is never the case that B will hold" is not *co-safe* because it can only be satisfied if B is never true over an infinite sequence. Despite of this, we can immediately verify its negation because the formula can be determined as unsatisfied over a finite number of steps as soon as B occurs, regardless the following time steps which became irrelevant. Another possibility is to define a *co-safe* task as ¬B $U$ G, which means "B must always false until G is true". In this way a *co-safe* formula can always be used to define properties that should be avoided at each time steps while performing a given task.

# Chapter 3

# LTL Progression for temporally extended goal

An LTL formula can be progressed along a given sequence of truth assignments. Indeed, given a RL agent, the LTL instruction can be updated at each step of the episode to identify aspects of the formula that still need to be accomplished. For instance the formula $\Diamond$(B $\wedge\Diamond$G) which means eventually go to blue square and then eventually go to green square is progressed to $\Diamond$G as soon the agent reaches a square where B is true.

**Definition.** Given an LTL formula $\phi$ and a truth assignment $\sigma_i$ over $\mathcal{P}$, $prog(\sigma_i, \phi)$ is defined as follows:

- $prog(\sigma_i, p) =$ true if $p \in \sigma_i$, where $p \in \mathcal{P}$
- $prog(\sigma_i, p) =$ false if $p \notin \sigma_i$, where $p \in \mathcal{P}$
- $prog(\sigma_i, \neg\phi) = \neg prog(\sigma_i, \phi)$
- $prog(\sigma_i, \phi_i \wedge \phi_2) = prog(\sigma_i, \phi_1) \wedge prog(\sigma_i, \phi_2)$
- $prog(\sigma_i, \bigcirc\phi) = \phi$
- $prog(\sigma_i, \phi_i \mathrm{U} \phi_2) = prog(\sigma_i, \phi_2) \vee (prog(\sigma_i, \phi_1) \wedge \phi_1 \mathrm{U} \phi_2)$

Note that from the binary operator U we can also define $\Diamond\phi = true\mathrm{U}\phi$ and $\Box\phi = \neg\Diamond\neg\phi$

# Chapter 4

# Reinforcement Learning

## 4.1 Markov Decision Processes (MDPs)

A Markov Decision Process is a discrete-time stochastic control model. It's defined by the tuple $\mathcal{M} = \langle S, T, A, \mathbb{P}, R, \gamma, \mu \rangle$ where S represents a finite set of *states*, $T \subseteq S$ is a set of *terminal states*, A is a finite set of *actions*, $\mathbb{P}(s'|s, a)$ is the *transition probability distribution* (i.e. given an action $a$ in a state $s$, it's the probability to reach the state $s'$), $R : S \times A \times S \to \mathbb{R}$ is the *reward function* used to encourage (or punish) the agent in performing a certain action, $\gamma$ is the *discount factor* and $\mu$ is the *initial state distribution*, that defines the probability of a state $s \subseteq S$ to be the first state of an episode.

## 4.2 Non-Markovian Rewards

The structure of the reward is built in such a way to return a value that is only dependent on the state $s'$ reached after the transition $\mathbb{P}(s'|s, a)$ caused by the action $a$ applied in the state $s$. The reward is then an immediate feedback from the model, unable to reflect the consequences of actions undertaken more than one step before.
A typical example could be described by an agent that must take a key to open a door reachable with more than one step: a Markovian reward cannot express this situation without considering "taking the key" and "opening the door" as two separate, independent events.

## 4.3 From LTL Instructions to Rewards

To overcome the limitations imposed by the Markovian rewards, the authors proposed a novel and extended definition of MDP called Taskable MDP.

To describe a sequence of events an LTL instruction built over a set of propositions $\mathcal{P}$ can be used to guide the agent: if the instruction is satisfied the agent will receive a reward of 1 and if it's falsified then the agent will be punished by returning a reward of -1. The episode ends as soon as a reward is received. To detect the changes in the environment and to assign truth values to the propositions in $\mathcal{P}$ as soon as the agent reaches certain states a labelling function $L : S \times A \to 2^{\mathcal{P}}$ is introduced in the model.
Formally, given an LTL instruction $\varphi$ over $\mathcal{P}$, a labelling function $L : S \times A \to 2^{\mathcal{P}}$ and the sequence of states and actions seen so far in the episode : $s_1, a_1, ..., s_t, a_t$ the reward function is defined as follows:

$$R_\varphi(s_1, a_1, ..., s_t, a_t) = \begin{cases} 1 & \text{if } \sigma_1, ..., \sigma_t \models \varphi \\ -1 & \text{if } \sigma_1, ..., \sigma_t \models \neg\varphi \\ 0 & \text{otherwise} \end{cases}$$

where $\sigma_i = L(s_i, a_i)$

## 4.4 Taskable MDP

It's now possible to include in an MDP the concept of LTL instructions. Given an MDP without a reward function $\mathcal{M}_e = \langle S, T, A, \mathbb{P}, \gamma, \mu \rangle$, a set of propositional symbols $\mathcal{P}$, a labelling function $L : S \times A \to 2^{\mathcal{P}}$, a finite set of LTL formulas $\Phi$ and a probability distribution $\tau$ over $\Phi$, a Taskable MDP is defined as follows: $\mathcal{M}_\Phi = \langle S', T', A, \mathbb{P}', R', \gamma, \mu' \rangle$ where $S' = S \times cl(\Phi)$, $T' = \{\langle s, \varphi \rangle, s \in T \text{ or } \varphi \in \{True, False\}\}$, $\mathbb{P}'(\langle s', \varphi' \rangle | \langle s, \varphi \rangle, a) = \mathbb{P}(s'|s, a)$ if $\varphi' = \text{prog}(L(s, a), \varphi)$ (zero otherwise), $\mu'(\langle s, \varphi \rangle) = \mu(s) \cdot \tau(\varphi)$ and

$$R'(\langle s, \varphi \rangle, a) = \begin{cases} 1 & \text{if } prog(L(s, a), \varphi) = true \\ -1 & \text{if } prog(L(s, a), \varphi) = false \\ 0 & \text{otherwise} \end{cases}$$

where $cl(\Phi)$ denotes the *progression closure* of $\Phi$, i.e. the smallest set containing $\Phi$ that is closed under progression.

# Chapter 5

# Model Architecture

Following the proposed definition of *Taskable MDP*, a new RL framework has been created to include the LTL formulas in it. While two modules are inherited from the standard RL models, one is unseen.

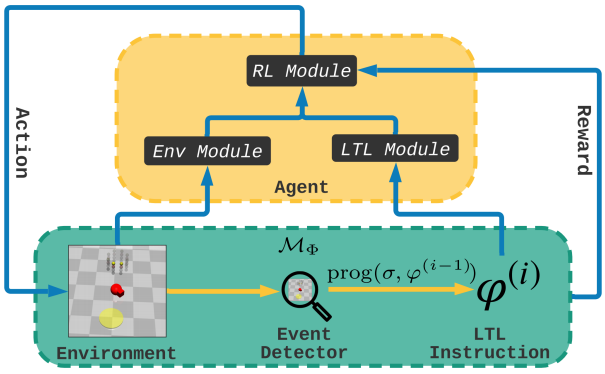



Figure 5.1: RL framework used in the paper. Credits to the authors.

The environment needs a wrapper as well (green module in the Figure 5.1), including the *Event Detector* (i.e. labelling function) and a set of LTL instructions that will determine the reward of the episode. Practically speaking, the *Event Detector* is nothing more than a series of controls made on certain parts of the environment, returning the truth value for the given proposition (e.g the agent is in a specific position in the grid).

The state of the environment will be embedded by the **Env Module**, creating a reduced latent representation through a CNN architecture.

The progressed LTL formula with respect to the output of the *Event Detector* is embedded by the **LTL Module**, producing a compact, numeric representation of a natural language formula. When dealing with natural language, RNN architectures are commonly adopted for their capacity to capture the sequence dependencies between the tokens.

Finally, the concatenation of the two embeddings will be fed to the **RL Module** (i.e. PPO or every other RL algorithm) that will treat it as an observation.
It's important to note that once the LTL instructions and the Event Detector are added to the environment, its nature changes from a classic MDP to a *Taskable MDP*, admitting non-Markovian rewards.
The original reward $R$ is then replaced by the $R'$ reward, computed by progressing the LTL formulas on the *Event Detector* truth assignments.

## 5.1 Env Module

The environment (Section 6.1) returns an observation composed by a 7x7 colored grid (3 channels) and a single value representing the orientation of the agent with respect to the world. The image is embedded by two convolutional layers with ReLU activation function and 3x3 kernels. To obtain a linear representation the output of the final convolutional layer is flattened.

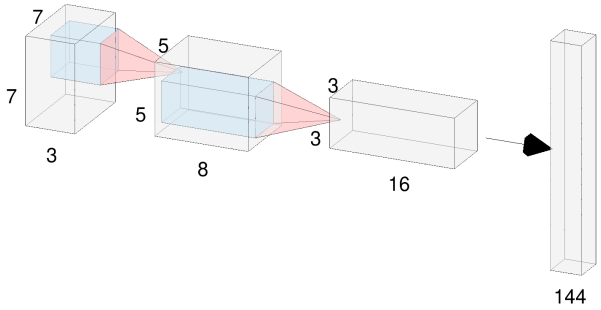



Figure 5.2: *Env Module* adopted in the experiments.

The directional value is not embedded and treated as it is.

## 5.2 LTL Module

To ease the LTL formulas processing, they are first of all converted in their *Prefix Notation* (or *Polish Notation*), e.g.:

$$\begin{gathered} \psi : \square\neg b \wedge \Diamond r \\ \Downarrow \\ pre(\psi) : [\wedge, [\square, [\neg, b]], [\Diamond, r]] \end{gathered} \tag{5.1}$$

Therefore they are made up by tokens (representing operators and propositions) and brackets to represent the relations between them. Naturally, the order has its own importance in the

representation, suggesting the use of a RNN.

All the formula's tokens are embedded using a trainable *Embedding Layer*, mapping the representation of the single token from a single integer into a real-valued hidden vector. The length of all the formulae is kept constant by using a zero padding. By an empirical choice, a *Gated Recurrent Unit* (GRU) performs better than a LSTM layer. The former is then used with a dual layer, bidirectional architecture. To keep a constant embedding dimension, the content of the last hidden layers is exploited and used as an embedding for the formula.

### 5.2.1 Graph Attention

In alternative, we designed a new architecture for this module (our main novelty with respect to Vaezipoor et al. 2021) able to effectively capture all the dependencies between each proposition/operator taking advantage of the type of relations among them. In other words, this module is capable to learn how each relation affect both the two elements involved, through a kind of multi-head attention mechanism.

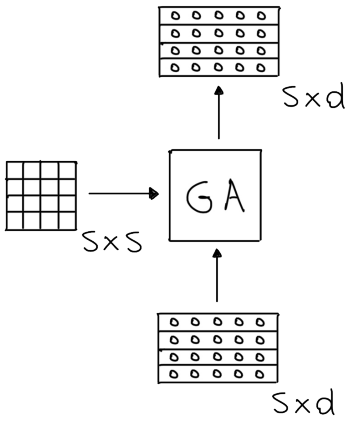



Figure 5.3: Graph Attention module which takes in inputs a sequence (of length $s$) of $d$-dimensional latent representations (bottom) and a $s \times s$ relation matrix (left-hand side) which specifies the relations between the elements of the sequence: the element in position $(i, j)$ specifies what relation the $i$-th element of the sequence has with respect to the $j$-th one.

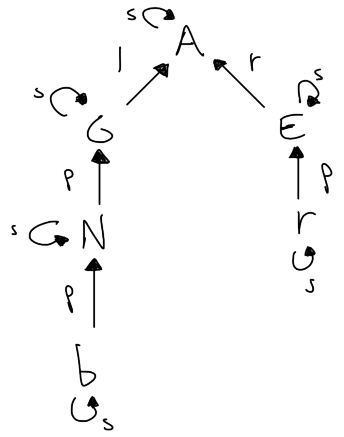
More specifically, it learns how attention it has to pay for each element of the input sequence to each of the others. Let's explain it through an example, given a LTL formula $\psi$ as follows:

$$\psi : \Box\neg b \wedge \Diamond r \tag{5.2}$$

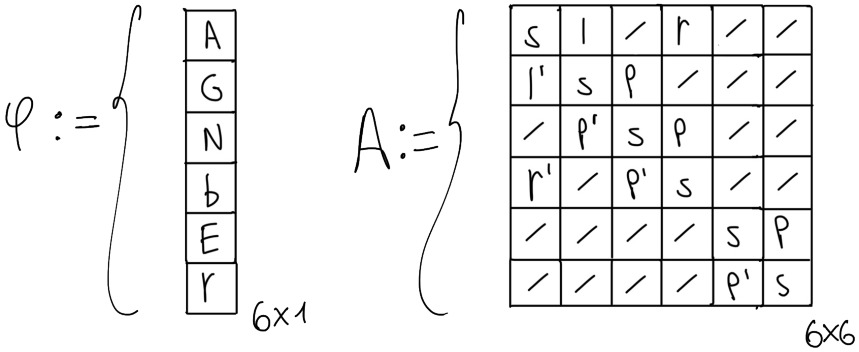
which for easing the computation we codify as:

$$\begin{matrix} [A, [ & G, [ & N, & b]], [ & E, & r]] \\ 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \tag{5.3}$$

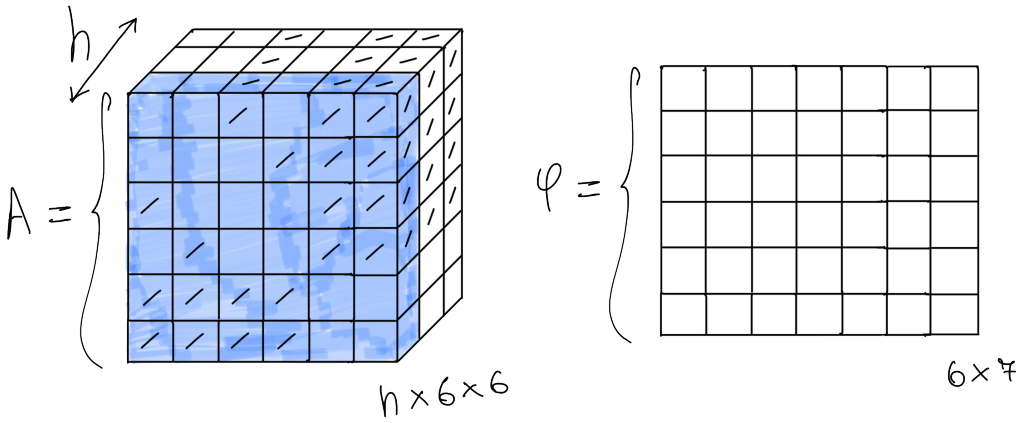
where are reported also the indices of each element for simplicity. Now, we better envision the relationships between each element by putting it on a graph or tree-like form:




where you can see several types of relations: $s$: self-loops, $p$: parent relation (assigned to the argument of every unary operator), $l$, $r$: left and right relations (for the arguments of binary operators) and finally the inverse relations ($p'$, $l'$, $r'$). At this point, we can construct the $6 \times 6$ relation matrix for $\psi$ as follows:

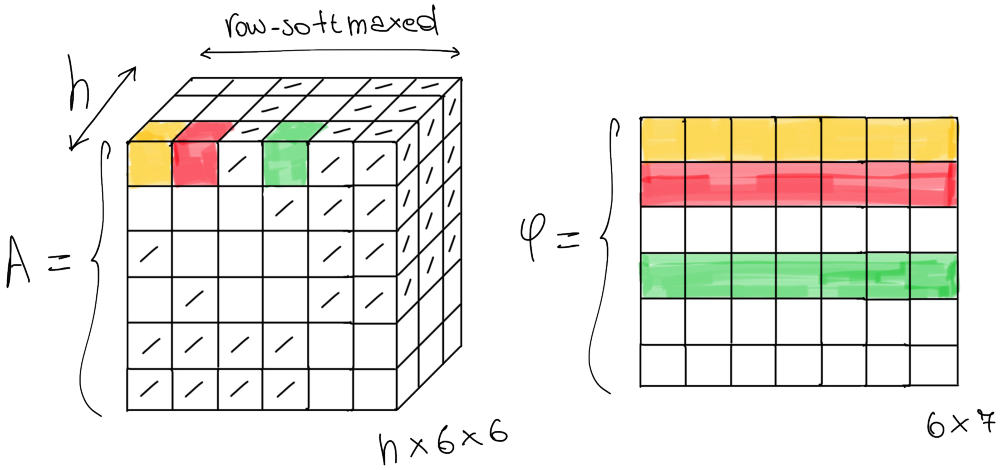



then, passing both to the embedding layers, one specific for the input sequence tokens and one specific for relations we end up with a hidden vector for each element of the input ($\psi$) and for each relation present in ($A$), expanding therefore their dimensions along the width and along the depth for $\psi$ and $A$ respectively:



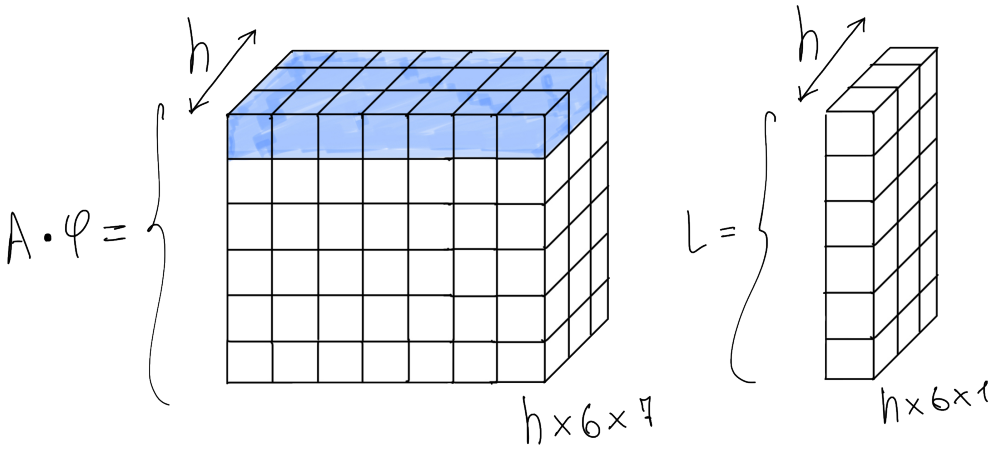

So 7 is an example of the hidden vector dimension of the elements of $\psi$, and $h$ will be the dimension of the hidden vector associated to each relation, and represents also the number of heads of the computation. The heads are represented by the vertical slice (like the one highlighted in blue) of the parallelepiped $A$ in the figure above, and every head will "work" independently from the others. The main idea behind this architecture is to make each head learn a different way to combine the information each element brings by "looking at" the neighbor elements of the tree-like form, i.e. based on their relations.
Notice the padding in $A$ which is the same of the original relation matrix for all the heads, meaning that those elements do not influence the computation of the elements in those position, e.g. the first element ($\wedge$ operator) computation won't be affected by the third, the fifth and the sixth ones, given that it has no relationship with the corresponding elements ($\neg$, $\square$ and $r$ respectively).

Before combining them we softmax the last dimension of $A$ in order to make the model to perform for each element a weighted sum on the values of the others:

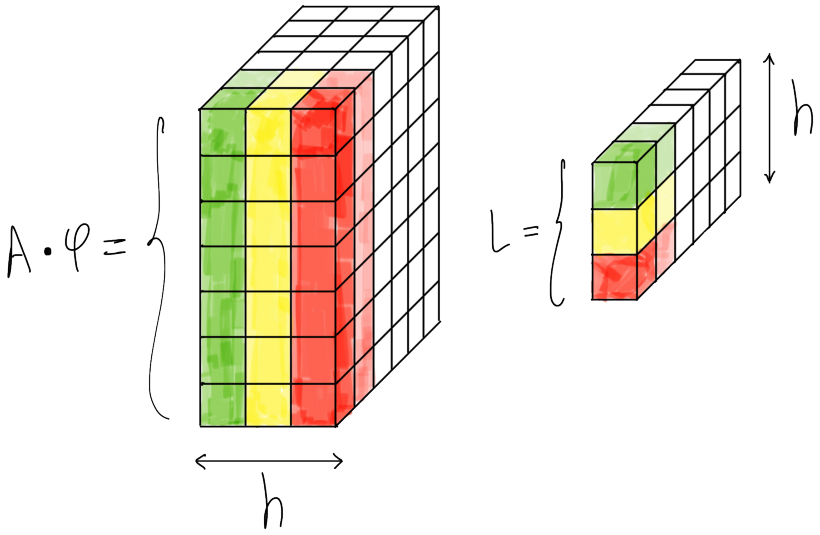



therefore, a matrix multiplication between each head and $\psi$ will be performed imposing that

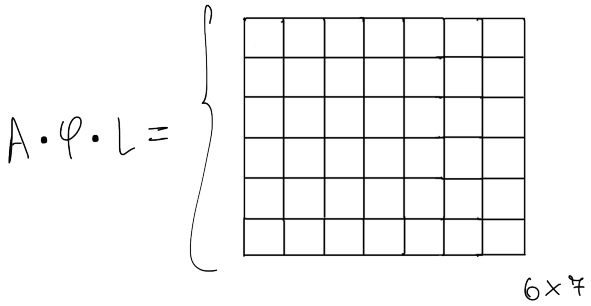
each element has to pay attention just to the elements it has relationships with.




Once we performed the head computations independently from each other, we can properly combine all the heads together through a linear transformation, so now working for each element of the sequence independently taking into account all the heads involved. For instance, all the heads into which the first element ($\wedge$) is involved is highlighted in blue in the figure above. Hence, the two tensors are overturned to better visualize the matrix multiplication involved in the next operation:




Therefore, each "plane" like that one is multiplied with the corresponding $h$-dimensional vector in $L$ as suggested by the colors in the figure above, so, through every such a vector in $L$ the model learns a specific way to combine the heads differently for each element of the sequence. The final result will be a sequence of hidden representations for each element of the same dimensions of the original input (a $s \times d$ matrix, $6 \times 7$ in this case):




## 5.3 RL Module

The RL Module used in the implementation is the PPO version of *stable-baselines3* (Raffin et al. 2019), with the same architecture for policy and actor networks returning a vector of length 64. As observations from the environment (namely, $S'$), this module get the concatenation of the output of the *Env Module* and the formula embedding.

# Chapter 6

# Experiments

In this chapter we report the experiments we have conducted using the LTL-progression based method described above, and investigate whether and how an RL agent can learn to accomplish more and more complex, temporarily-extended tasks encoded in Linear Temporal Logic form. Specifically, we try to prove the effectiveness of this method in comparison with other less powerful baselines, i.e. the one without the LTL-progression and its myopic version. And finally, we demonstrate how this approach generalize to larger LTL instructions than those seen during training and to the same type of tasks but involving different propositions.

## 6.1 Environment

We have run all the experiments on a single environment, which is in our opinion not so demanding, so that the experiments can be reproduced by anyone, and at the same time enough intrinsically complex. In particular we have done it in the *Minigrid* environment, shown on the Figure 6.1 below:

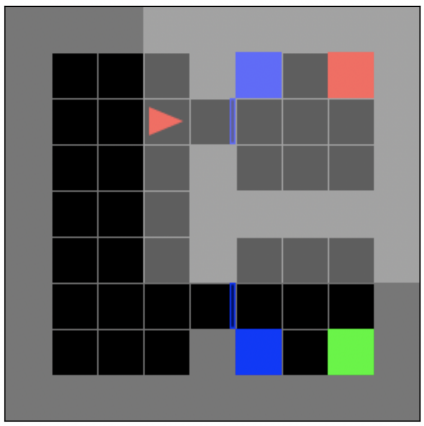


Figure 6.1: Minigrid environment.

That is a quite simple environment, roughly speaking, composed of a 7x7 grid into which not all the cells can be reached by the agent, the red triangle.
The "navigable" cells are not randomized meaning that are always the same, and are arranged in a such a way that two sort of rooms are created. Each of this room will have a blue square and one more square that could be red or green: the room into which the red square will be placed is chosen randomly, and therefore also the green one.
Accordingly, the agent can reach one of the navigable cell, be oriented along one of the four cardinal directions and move forward, i.e. the direction it is currently pointing to.
Moreover, the environment is not fully-observable, indeed the agent can "see" only a square of certain side placed right in front of it (the highlighted square in Figure 6.1), so that the agent could be required to explore a bit before taking a decision.
The real complexity of this environment stems from the fact once the agent entered a given room it will not be able to escape it, therefore it has a sort of "one shot" attempt to solve the task assigned, otherwise it will fail. That's the reason why a policy which cannot take into account temporarily-extended task (myopic) can't obtain good results here.

## 6.2 Tasks

Using the LTL language a wide range of tasks can be formalized and therefore assigned to our RL agent. The main types are the following:

- Single goal: $\Diamond R$ (finally reach the red square)
- Goal sequences: $\Diamond(R \wedge \Diamond G)$ (reach the red square, then the green one)
- Disjunctive goals: $\Diamond R \vee \Diamond G$ (reach the red square or the green one)
- Conjunctive goals: $\Diamond R \wedge \Diamond G$ (reach the red square and the green one in any order)
- Safety constraints: $\Box \neg B$ (never touch the blue square)

Such type of tasks can be also combined together to form new more complex types of them, e.g. $\Diamond(R \wedge \Diamond G) \wedge \Box \neg B$: "never touch the blue square while going to the red square and then to the green one".

It's important to notice unlike LTL-f, the formal language LTL is interpreted over infinite traces, however, we have to deal with finite times, therefore we set a maximum number of steps the agent can take within the same episode, beyond which it would receive a negative reward, i.e. $-1$. Anyway, not all the LTL formulas can be verified or falsified in finite times, therefore it

will receive a meaningful reward signal only in the case the given LTL task is actually verified or falsified within the "timeout", i.e. the maximum number of steps.
For instance, the *eventually* operator ($\Diamond$) cannot be falsified in finite traces, just verified if and only if its argument actually occurs within the timeout. Whereas the *always* operator ($\Box$) cannot be verified in finite traces (since no one can actually assured it will be valid to infinity), but only falsified if the opposite of its argument occurs within the timeout.

Based on that, we set up several tasks to be accomplished within the timeout in the environment defined above. As well as Vaezipoor et al. 2021 the **LTL Module** of all the models tested is first pre-trained on a simplified *gym* environment (referred to as **LTLBootcamp**) which represents a new *Taskable MDP* defined by a single state into which every action has the sole effect of making a given proposition true. That is done in order to facilitate the training on the main environment and to later better generalize on other downstream tasks. In other words, by doing so, the model starts to learn *what* to do (no matter what the environment it is), and then during the main training it will learn *how* (in a specific way for the environment at hand).

## 6.3 Toygrid Tasks

The very first test we conducted is the one the *Toy-minigrid* env is born for, namely the one composed just of the following couple of tasks, in order to prove the effectiveness of this method over the myopic variants:

1. $\Diamond(B \wedge \Diamond R)$: "*go to blue then red*".
2. $\Diamond(B \wedge \Diamond G)$: "*go to blue then green*".

As you can see in Figure 6.1 the blue square is present in both the rooms, therefore to accomplish the first requirement of the task the agent could go in either room, however if it would be the wrong one it was not able to escape it, that's why an agent which cannot take into consideration what it has to do next before accomplishing the current requirement would fail the half of the times on average. These considerations are readily proved by the results of the two training performed and reported in the Figure 6.2: one for the myopic agent and one for the agent equipped with the LTL progression.

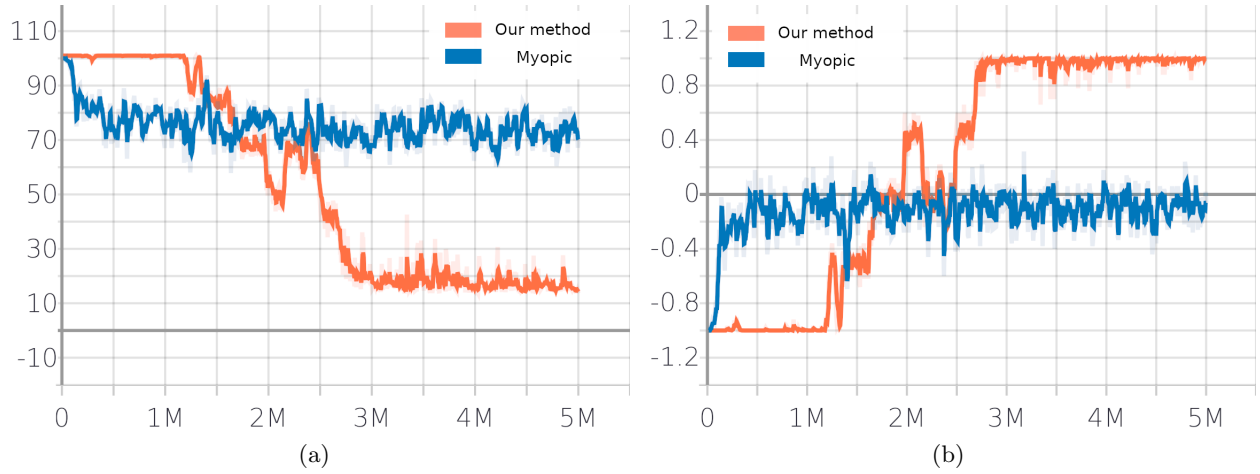



Figure 6.2: (a) mean episode length over the 5 million steps elapsed during training (b) mean episode reward during training.

## 6.4 Ordered and Avoidance Tasks

The latter experiment regards a way more complete test into which the agent has to perform all the possible tasks we have described so far: both *ordered tasks* (single goals, disjunctive and conjunctive goals and sequence goals) and *avoidance tasks* (avoid violating safety constraints while achieving other goals).

Specifically, the task distribution $\Phi$ from which a new LTL formula is uniformly drawn at every episode is composed of the following:

- $\Diamond R$ (single goal)
- $\Diamond G$ (single goal)
- $\Diamond B$ (single goal)
- $\Diamond B \wedge \Diamond R$ (conjunctive goal)
- $\Diamond B \wedge \Diamond G$ (conjunctive goal)
- $\Diamond B \vee \Diamond R$ (disjunctive goal)
- $\Diamond B \vee \Diamond G$ (disjunctive goal)
- $\Diamond(R \wedge \Diamond B)$ (sequence goal)
- $\Diamond(B \wedge \Diamond R)$ (sequence goal)
- $\Diamond(G \wedge \Diamond B)$ (sequence goal)
- $\Diamond(B \wedge \Diamond G)$ (sequence goal)

- $\Diamond(R \wedge \Diamond(B \wedge \Diamond R))$ (sequence goal)
- $\Diamond(G \wedge \Diamond(B \wedge \Diamond G))$ (sequence goal)
- $\Diamond(B \wedge \Diamond(R \wedge \Diamond B))$ (sequence goal)
- $\Diamond(B \wedge \Diamond(G \wedge \Diamond B))$ (sequence goal)
- $\Box\neg B \wedge \Diamond R$ (single goal with safety constraint)
- $\Box\neg B \wedge \Diamond G$ (single goal with safety constraint)
- $\Box\neg R \wedge \Diamond B$ (single goal with safety constraint)
- $\Box\neg G \wedge \Diamond B$ (single goal with safety constraint)

And according to the results obtained during training and reported on the Figure 6.3 below, our agent effectively learn to solve all the tasks assigned.

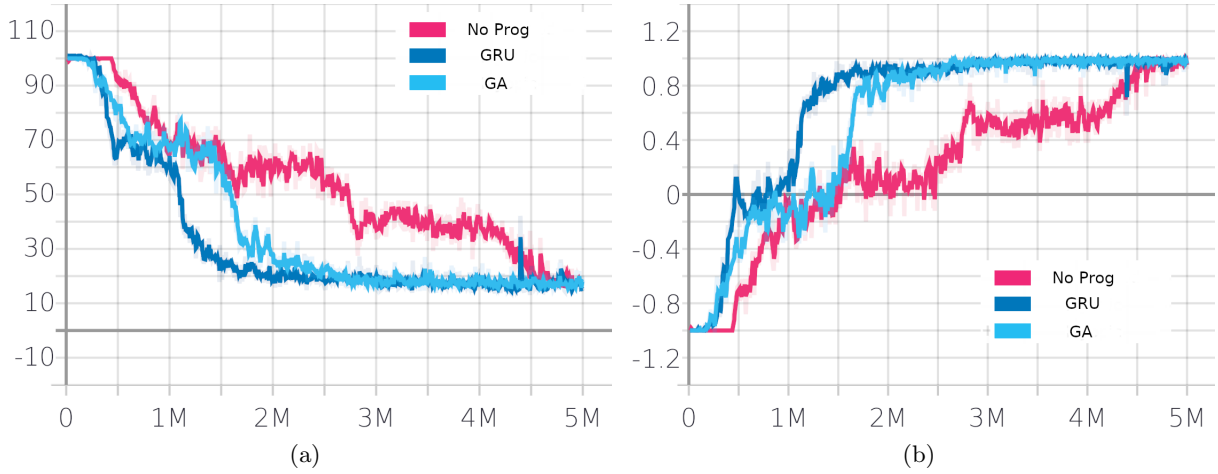



Figure 6.3: (a) mean episode length over the 5 million steps elapsed during training (b) mean episode reward during training.

In the Figure 6.3 above we reported the training results of three models, *No Prog* is the one into which the agent cannot make use of the *prog* function e therefore it observes always the complete task assigned, and that's to prove the utility of the progression method which effectively make the reward non-Markovian. Then, are reported the two main models: the one equipped with the recurrent neural network to extract the features from the current LTL instruction, and finally the one equipped with our novel architecture based on multi-head attention mechanism to extract the features from the same instruction as well. As you can see both the two models obtain the maximum mean reward therefore learning to solve all the tasks we defined. Although the GRU version seems to learn quicker (it's indeed more sample-efficient) the GA version is way

more time-efficient because of the nature of its computation, which can be highly parallelized, obtaining virtually the very same results while saving some training hours.

The sequence of figures below represents an example of complex task accomplishment once the agent has been trained with our implementation of the method.

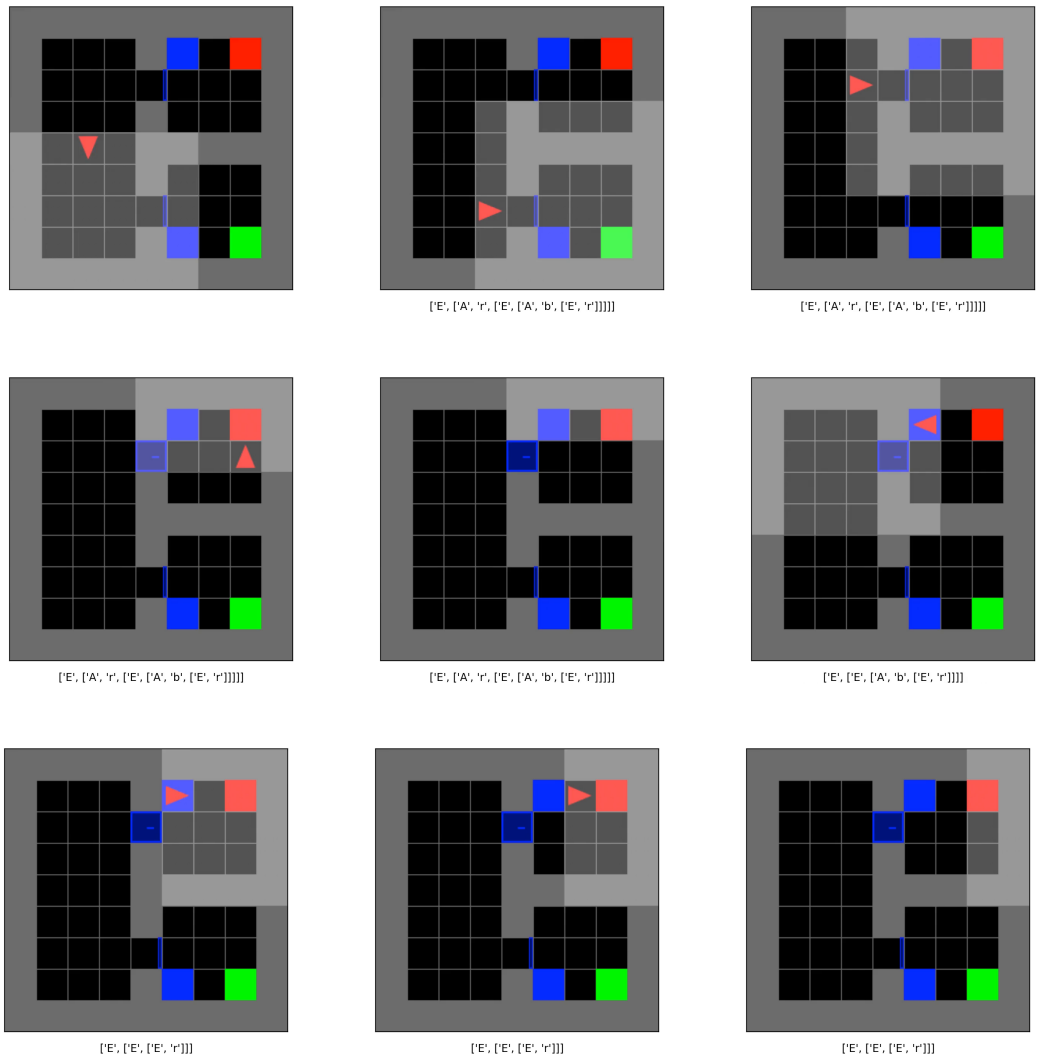


Figure 6.4: Sequence of frames of an episode. From top-left to bottom-right. Since the grid is partially observable, the agent first needs to explore it fully (imgs. 2 and 3). Once the right colored goal is found, the agent can accomplish the task by entering the room. The progressed formula is on the bottom of each frame.

## 6.5 Task Generalization

The last experiment involves testing the generalization capabilities of the resulting agent. In particular, a training has been performed on a subset of the tasks defined in section 6.4.

At test time, the hidden tasks has been proposed to the agent, recording its performances.

Hence, two types of generalization can be tested, based on what has been hidden in train time:

**Propositions generalization**: made by training the agent over a set of formulae without including, for every formula, all the propositions. For example, it can be trained over "go to blue then go to red" and "go to blue then go to green" and tested over "go to green then go to blue". It turns out that none of our models behave so well on this type of unseen situation. Indeed, they all can't go beyond the 55% of success rate on average.

**Formulae generalization**: made by training the agent over a set of formulae and testing it over a set of deeper formulae, like training on "go to blue then go to red" and testing over "go to blue the go to red then go to blue". It turns out that the GRU version of our model can't go over the 55% of success rate on average on this type of situation as well, whereas the GA variant manages to reach the 82.5% of success rate on average on the very same unseen task set.

# Chapter 7

# Conclusions

The approach proposed by the authors in the paper has revealed a great effectiveness in defining and solving multi-task RL problems. *Taskable MDPs* are easy to use yet very powerful, tackling in a mathematical way the non-Markovian rewards problem. The high modularity suggests also a great compatibility and freedom on the applications, permitting 2D and 3D environments, discrete and continuous action spaces and so on. Given the presence of the *Env Module*, the observation space can be virtually everything.
Various techniques have also been used to enhance this approach, like pretraining the *LTL Module* or progressing the formulae, making the agent temporal-aware of what is happening in real time with respect to the propositions.

However, the complexity introduced by the *LTL Module* adds a relevant overhead required to solve such complicated problems.
Our novelty (Graph Attention) tackled this problem by increasing the FPS (environment's frames per seconds during training) from 280 to 365 in the same computer architecture, without altering the effectiveness of the model. This is caused by the different operations used, permitting a better parallelization of the computation. Moreover, it can effectively better generalize on previously unseen LTL tasks with respect to the variants based on recurrent neural networks.

# References

De Giacomo, G. (2021). "Linear Temporal Logic (LTL)". In: *Slides from Reasoning Agents course.*

Icarte, Rodrigo Toro, Klassen, Toryn Q., Valenzano, Richard Anthony, and McIlraith, Sheila A. (2018). "Teaching Multiple Tasks to an RL Agent using LTL." In: *International Conference on Autonomous Agents and Multiagent. Systems (AAMAS)*, pp. 452–461.

Raffin, Antonin, Hill, Ashley, Ernestus, Maximilian, Gleave, Adam, Kanervisto, Anssi, and Dormann, Noah (2019). *Stable Baselines3.* `https://github.com/DLR-RM/stable-baselines3`.

Vaezipoor, Pashootan, Li, Andrew, Icarte, Rodrigo Toro, and McIlraith, Sheila (2021). "LTL2Action: Generalizing LTL Instructions for Multi-Task RL". In: *International Conference on Machine Learning (ICML).*