Report Reasoning Agent

Irene Bondanza
Matteo Emanuele
Pietro Manganelli Conforti

November 2021

1 Introduction

When facing complex tasks, plenty of different solutions may be proposed through the usage of artificial intelligence: from the exploration of an environment to a classification task, from an optimization of an industrial process to a smart behavior into a software, almost everything can be modelled and tackled nowadays with a declination of the broad field that is Artificial Intelligence.
One of the most exciting branches of the machine learning is the reinforcement learning: this consists in a simplistic point of view, in making an agent learn how to solve a task only giving feedback on the goodness of specific action executed. A reinforcement learning task has three cardinal points that need to be taken always under consideration: an environment in which the task takes place, an agent that execute the task, and actions that can be conducted by the agent based on the "state" in which it can be identified. The purpose of reinforcement learning is for the agent to learn an optimal, or nearly-optimal, policy that maximizes the "reward function" or other user-provided reinforcement signal that accumulates from the immediate rewards.
In the background section we will go a little more in depth into important concepts for the reinforcement learning as the Markov Decision Process(MDP), a general modelling for a reinforcement learning settings.
MDPs can be extended to better embed informations that help better modelling. One way to do so is to use rewards that are considered unconventional for Markov processes, called "non-markovian rewards".
To extend MDPs with non-Markovian rewards we need a language for specifying such rewards,since an explicit representation is no longer possible, as we have an infinite number of past samples for the future. A paper from 2018[$BGP18$] brought developments in the theory of temporal logic over finite traces to the problem of specifying and solving MDPs with non-Markovian rewards. Such work treated the handling of reward formulas through the transformation of the LTL_f/LDL_f rewards into Deterministic Finite Automatas(DFAs).
This transformation can be done in handable time complexity as demonstrated by different papers, if we exclude the worst case scenarios [$BGP18$][$TV05$].

In the background section we will go through the knowledge needed to comprehend the basics of this theoretical part, talking about temporal logic, and non-Markovian rewards expressed in LTL_f/LDL_f.
With this transformation, the MDP can be extended appending the state of the automaton to the MDP state.
Such augmentation though in some scenarios may backfire, depending on the kind of environment in which the task takes place. A simple yet intuitive example is proposed in [Ant21]. If a reinforcement learning agent is trying to learn how to beat a videogame, it will employ CNN models to process the input,which in this case will be the pixels of the screen. Adding the automaton state to the observation of the environment,may lead the agent to treat this new information as "just another pixel". One possible solution to this problem is to treat separately the different state of the automaton: specifically training portions of the RL agent to learn how to solve the task in a specific state of the automaton is a solution that allows to apply such reward shaping approach for Non Markovian scenarios too.
A slightly different approach was proposed by the previous group [$Ant21$] where instead of training portion of the network of the agent for each automaton state, the same solution was reached with the training of separate networks agent. This, as they demonstrated, led to a satisfactory result but with some major intuible downsides which are the training time required to train three networks separately, as well as memory consumption too.
In this project we propose a solution to a reinforcement learning task applied to the sapientino environment through the usage of non-markovian rewards; this will be conducted with the training of a network for our RL agent by portions that will be selected with a filtering mask to train only some neurons at each automaton state(the specifics will be discussed further in this report).
Before diving into the project, on the implementation we developed and the results we obtained, we will briefly go through some background knowledge reguarding LDL_f/LTL_f formulas and basic reinforcement learning concepts. We will also explain the basics of the environment in which the task is taking place, as well as the algorithm we choose to work with(Double_DQN and PPO).

2 Background Knowledge

In this section we aim at filling the general understanding of some key topics which imbued the project. In the specific, we will talk about temporal logic in general together with some general aspects of LTL_f and LDL_f just to allow the reader to reach a general understaing of those topics. Later on, we will go through the basics of reinforcement learning, stating important concepts like MDP and others. To conclude, we will go also through a relatively general section were we will give a brief overview of the non-Markovian rewards written in LDL_f/LTL_f.

2.1 Temporal Logic, LTL_f, LDL_f

We will go through a brief introduction of LTL and modal logic in general, in order to introduce important concepts to make easier the explanation of certain informations retrivable from papers available in the references.
In propositional logic we can formalize things as propositional facts that can be either true or false (example: x = Today is raining, y = Tomorrow will rain, etc.). This is fine if we have a short number of conditions.
There are cases though, in which we can't express more complex facts without introducing an infinite number of variables. For example, if we want to say that it will not be sunny forever, we would have to explicitly define that it won't be sunny on all the days. This is a limit case that shows how one of the most famous and used kind of logic, the propositional one, it cannot cover all the use-cases.
Also, facts in propositional logic can be completely uncorrelated and we wouldn't have a way to tell. This is why we can make use of another kind of logic, called "modal logic". Its use is fundamental for those cases in which we would like to model situations where things are not simply going to be "true or false", but that may be "true or false depending on the situation". We can in fact have different possible scenarios not related with respect to time. Then we can express that something is true in one or in multiple situations using the proper semantic.
Temporal logic(TL) is a sub-category of Modal logic where we search for true scenarios in the future, one by one at the time. Going into a sub-category of temporal logic, we find LTL(Linear Temporal Logic), an extension of TL that introduces more operators that are related to time.
In LTL, one can encode formulas about the future of a scenarios, like for example a condition that eventually will be true or a condition that will not be true until another fact becomes true, making it very useful for certain scenarios
LTL is constructed on top of finite set of propositional variables and logical operators like $\neg$. It has at its disposal also the temporal modal operators next(X) and Until(U).
Other than these fundamental operators, there are additional logical and temporal operators defined in terms of the fundamental operators to write LTL formulas. The additional logical operators are $\wedge$, $\rightarrow$,$\leftrightarrow$, true, and false, common also to other kinds of logic. Following are the additional temporal operators:

- G for globally(always): Gx is true if x is true now and forever. This operator in LTL is the correspondent of in TL.
- F for finally: Fx is true if x will be true now or in the future. This operator in LTL is the correspondent of in TL
- R for release
- W for weak until
- M for strong release

We will also introduce the concept of "trace": in LTL a trace is the equivalent of a Kripke model in modal logic. A trace is a sequence of time points $[s_i, s_{i+1}$, ...], which means in simple words that a trace is simply a sequence of states, i.e. a complete description of the evolution of a certain domain over time. Here there is an example of a trace: It is a "complete" description in the sense that

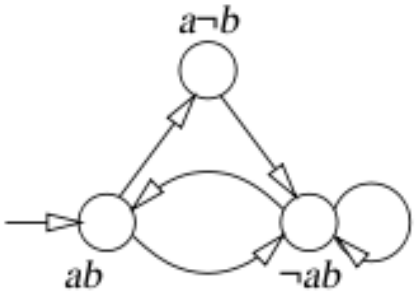


Figure 1: Example of a trace.

at a given time point s_i, we know everything of that point (with everything we mean what is exactly true and what is exactly false).
Traces in the structure are infinite paths: this is because each time point must have a successor.

LTL on finite traces, called LTL_f [DV13], has exactly the same syntax as LTL on infinite traces [Pnu77].
Quite often, especially in the context of temporal constraints and preferences, LTL formulas are used to express properties or constraints on finite traces of actions/states.
LTL_f is an extension of LTL over finite traces that has the same expressive power of other kinds of logic over finite traces, while maintaining the declarative nature and intuitive appeal of LTL.
It is known that LTL_f regular expressions though, are a a kind of formalism for expressing temporal specifications that may bring to some troubles, since, for example, they miss a direct construct for negation and for conjunction.
To overcome this difficulties, in Linear Dynamic Logic of Finite Traces(LDL_f), has been proposed.
Generally speaking,in fact, LDL_f is obtained by merging LTL_f with regular expression through the syntax of PDL(Propositional Dynamic Logic).

2.2 Reinforcement Learning(RL)

Another core knowledge that we felt it was the right fit for a background section was regarding reinforcement learning, since the task we are facing is deep inside such topic.

We will go briefly through the main concept to provide the reader a general understanding of the overall concepts, in order to allow a better understanding of the algorithms explained in a later section.
When we talk about reinforcement learning we talk about a sub-branch of the machine learning.

At the center of every reinforcement learning problem there are an agent and an environment:

- The environment provides information about the state of the system.
- The agent observes these states and interacts with the environment by taking actions.

Actions can lie in a discrete world or in a continuous world, depending on the modelling of the task itself. Indeed,modelling is an important part of the reinforcement learning task, as we will see later.
These actions cause the environment to transition to a new state. And based on whether the new state is relevant to the goal of the system, the agent receives a reward.
The reward is a key concept for every RL task: in fact,it is the reward that guides the agent towards the goal, and will do so learning a so called "policy" that is basically telling the agent which actions are the more convenient to reach the goal, since they are capable of providing the maximum reward possible. This highlight one of the most crucial aspect: defining a reward is an important step that has to be faced and thought through every time a task is decided to be tackled with reinforcement learning. About specific definition of the rewards, we will discuss it later in a further section in this background.
In general speaking, a reinforcement learning agent interacts with its environment in discrete time steps. At each time t, the agent receives the current state s_t and reward r_t. It then chooses an action a_t from the set of available actions, which is subsequently sent to the environment. The reward that receives guide the whole training process: based on such value, the agent will learn what actions are effective to reach the goal, and which other actions are not effective and needs to be discarded.
Talking of basic reinforcement learning settings, we can introduce the concept of "Markov Decision Process"(MDP). An MDP is a stochastic process that respect the Markov rule, which says that the probability to move from a state to another depends only on the directly precedent state, and not from the whole past of the agent.

It can be modelled as:

- a set of environment and agent states, S;
- a set of actions, A, of the agent;

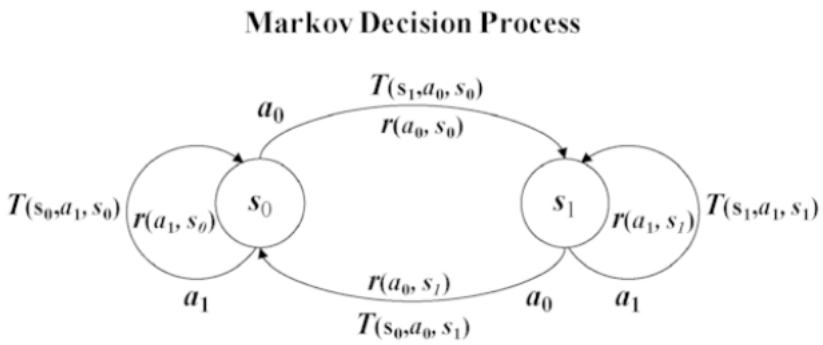


Figure 2: Markov Decision Process cycle modelling.

- $\mathrm{P}_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability of transition (at time t) from state s to state s' under action a.

- $R_a(a, s, s')$ is the immediate reward after transition from s to s' with action a.

In the general scene though, it is not rare to have a goal which can be identified as "temporal", which means that the past does play a role on the goal itself. In that case our process cannot be identified as a Markovian one, but we will talk of Non-Markovian scenarios in this case.
In fig.3 we report a graphical representation of the different sub categories for reinforcement learning approaches. As we can see, RL algorithms can generally

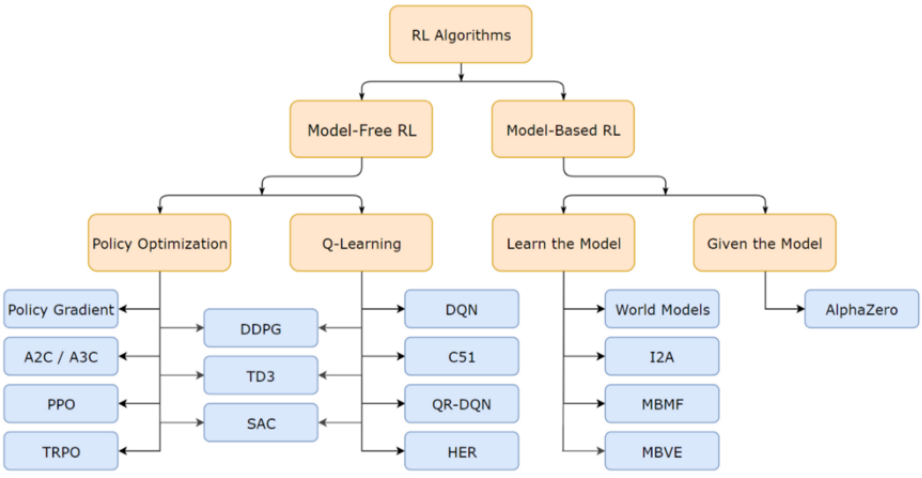


Figure 3: Reinforcement Learning taxonomy as defined by OpenAI.

be divided into two categories: Model based and Model Free.

Quoting indirectly from [DN08], we can propose a brief explanation and comparison between these two families of algorithms.
Model-based algorithms use experience to construct a model of the transitions and immediate outcomes in the environment. Appropriate actions are then chosen by searching or planning in this world model.
Model-free algorithms on the other side, use experience to learn directly one or both of two simpler quantities (state/ action values or policies) which can achieve the same optimal behavior but without estimation or use of a world model. Given a policy, a state has a value, defined in terms of the future utility that is expected to accure starting from that state.
Model-free methods are statistically less efficient than model-based methods, because information from the environment is combined with previous, and possibly wrong sometimes, estimates or beliefs about state values, rather than being used directly.
In this project we addressed the task with two different algorithms, Double DQN and PPO, two model-Free RL algorithms that we will discuss later in this theoretical work.
To conclude this section about reinforcement learning, we will state a couple of additional facts about rewards.
It has long been observed by different papers(LTL_f/LDL_f Non-Markovian Rewards paper reference here @@@) that In many scenarios it urges the need to define more sophisticated reward functions that depends on more than one state. For example, we may want to reward for a robot that eventually delivers a package for a logistic task each time that it gets a request; again we may have a robot that has to reach three different locations in a specific order. Such rewards are non-Markovian.

2.3 LTL_f/LDL_f non-markovian Rewards

In Markov Decision Processes (MDPs), the reward obtained in a state is Markovian,which means that depends on the last state and action. This dependency makes it difficult to reward specific long-term behaviors, such as always closing a door after it has been opened, or providing an item every time it is requested by a customer.
A paper from 2018[BGP18] addressed a solution to this problem passing through the concept of "MDP extension".
As we have discussed in the previous section, The Markov assumption is a must for this model, since it states that the effects of an action depend only on the current state in which is taking place and to the previous action executed.
This problem may be tackled through the usage of non-Markovian rewards, that are capable of encapsulating the temporal dependency of the task. We will call this a Non Markovian Reward Decision Process(NMRDP).
In general, an NMRDP is represented as a tuple exactly as it works also for the basic MDP, but with the added extention of the reward: M = ⟨S,A,Tr,R⟩, where each element inside this tuple is as follow:

- S are the states
- A are the set of actions
- Tr is the transition function
- R defined as R $(S \times A)^* \rightarrow R$

As we can see,the difference with the vanilla MDP is the last point, the reward, as we have already mentioned.The reward is now a real valued function over finite state-action sequences. Given now a trace π, the value of π is in general equal to:

$$v(\pi) = \sum_{i=1}^{|\pi|} \gamma^{i1} R(<(1), (2), ..., (i)$$

Where gamma is the discount factor and the general value of the trace (i) denotes the pairs composed by the previous state and the action executed. A single policy then now, will induce a distribution over the set of all the possible traces(which are infinite). This allows us to define the value of a policy for a given state s_0 as :

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

Now that we have thrown the common ground to further proceed in our disclosure about non markovian rewards, we can finally add an additional information, given by a starting intuitive notion. Specifying a non-Markovian reward function explicitly is problematic in general, and this is because of the infinite nature of the traces. For starters, it would be troublesome also if the traces were finite. But typically though, we want to reward behaviors that correspond to specific patterns, which means that we try to reward those patterns instead of a set/subset of actions. And here is where LDL_f, LTL_f comes in handy. LDLf provides an intuitive and convenient language for specifying R implicitly, using a set of pairs $(\Phi_i, r_i)_{i=1}^m$.
Intuitively, if the current partial trace is $\pi = <s_0, a_1, ..., s_{n1}, a_n>$, the agent receives at s_n a reward r_i for every formula Φ_i satisfied by π.
So to conclude this part and move on to the next section, we will state in a couple of sentences what we will do in this project.

2.4 General approach for our task

We want to solve through a reinforcement learning algorithm a task in a continuous environment called "Sapientino". In this environment, an agent represented by a unicycle has to move through this world with the basic movements at a unicycle's disposal in order to reach a specific goal or goals, depending if it has to reach only a single cell in the world represented by a color, or a sequence colors.
When we are facing the simplified version of this task(i.e. one color), we are

facing a simple Markovian problem, while we are facing a non markovian task when we have a sequence of colors. In this latter case, we have inside the task the concept of "temporality" encapsulated inside the task goal.
To solve this more general and complex task, in our case, we will proceed extending our MDP with the state of a Deterministic Finite Automata(DFA), since temporal formulas like the one discussed in this background section allows us to compute a DFA from temporal formulas with which we have written our non-Markovian reward.
This solution will be implemented and further explained in a later section.

3 Algorithms

3.1 Proximal Policy Optimization

Proximal Policy Optimization (PPO) belongs to the family of policy gradient methods for reinforcement learning which optimize a surrogate objective function using stochastic gradient ascent.

Pre-existing methods fails on different problems for example vanilla policy gradient methods are not robust and efficient on poor data and trust region policy optimization (TRPO) is complex and has compatibility problems with some architectures. On the other hand TRPO introduces trust region strategies instead of the line search strategy making sure that the new updates policy is not far away from the old one (the new policy is within the trust region of the old policy) [4].

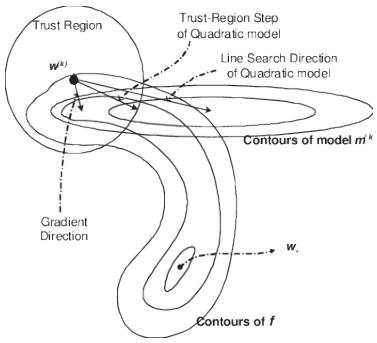


Figure 4: Difference between line search and trust region

PPO aims to combine the advantages of TRPO while using a first-order optimization. In [Sch+17] researchers have shown that PPO strikes a favorable balance between complexity and performance. When performing optimization on this loss the problem may leads to large policy updates.

3.1.1 Background

The basic gradient estimator of policy gradient methods is:

$$\hat{g} = \hat{E}_t[\nabla_\theta \log(\pi_\theta(a_t|s_t))A_t]$$

$$\begin{aligned} where: \quad & \pi_\theta = \text{stochastic policy} \\ & A_t = \text{estimator of the advantage function at timestep t} \end{aligned}$$

For the implementation is constructed an objective function whose gradient is the policy gradient estimator, the estimator $\hat{g}$ is obtained by differentiating the objective function [1]:

$$L^{PG}(\theta) = \hat{E}_t[\log(\pi_\theta(a_t|s_t))\mathsf{A}_t] \tag{1}$$

To improve training stability, it is necessary to avoid parameter updates that change the policy too much at one step, here it comes into play TRPO that enforces a KL divergence [2] constraint, which measures a distance between old and new probability distribution. This contraint ensures that the new updates policy is not far away from the old one.

$$D_{KL}(\pi_{\theta_{old}}||\pi_\theta) = \pi_{\theta_{old}}(a) \cdot log\frac{\pi_{\theta_{old}}(a)}{\pi_\theta(a)} \tag{2}$$

Therefore the (surrogate) objective function of TRPO is defined as follows:

$$\begin{aligned} \max_\theta \quad & \hat{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\mathsf{A}_t] \\ \text{s.t.} \quad & \hat{E}_t[D_{KL}(\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t))] < \delta \end{aligned} \tag{3}$$

Note that the KL constraint adds an overhead to the optimisation process whereby the PPO approach is preferred.

3.1.2 Clipped Surrogate Objective

The objective function is not subject to a constraint but instead it is clipped to remove incentives for the new policy to change too much from the old policy. First of all, let define the probability ratio between the new policy and old policy and we can call it as $r(\theta)$:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, \qquad r(\theta) \in [0, 1)$$

From the TRPO objective function [3], we can define:

$$L^{CPI}(\theta) = \hat{E}_t[r_t(\theta)\hat{\mathsf{A}}_t]$$

Without adding any constraint, this objective function can lead to instability due to excessively large policy update. Instead of adding the KL contraint, PPO impose to the policy to stay within a small interval around 1, thus between $1-\epsilon$ and $1+\epsilon$. Therefore if $r(\theta)$ falls outside the range $[1-\epsilon, 1+\epsilon]$ then the advantage function will be clipped [see figure 5].
The main objective is the following:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{\mathsf{A}}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{\mathsf{A}}_t]$$

As you can see it takes the minimum of the clipped and unclipped objective thus the final objective is a lower bound on the objective.

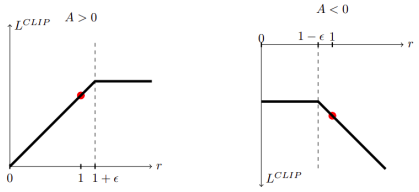


Figure 5: L^{CLIP} as a function of r, for positive (left) and negative advantages (right).

3.1.3 Adaptive KL Penalty Coefficient

An alternative to the clipped surrogate objective is to use the KL divergence as a penalty instead of using it as a constraint (like TRPO), hence it penalizes the objective if the new policy is different from the old policy.

$$L^{KLPEN}(\theta) = \hat{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{\mathsf{A}}_t - \beta D_{KL}(\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t))]$$

3.1.4 PPO

If the purpose is to use PPO with a neural network architecture for which there are shared parameters between the policy and a value function, we must use a loss function that combines the policy and a value function error term. Moreover we can also add an entropy bonus to ensure the exploration. The combination of these three terms lead to the objective:

$$L_t^{CLIP+VF+S}(\theta) = \hat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]$$

$$c_1, c_2 = \text{coefficients}$$
$$S = \text{entropy bonus}$$
$$L_t^{VF} = (V_\theta(s_t) - V_t^{targ})^2 \text{ [squared-error loss]}$$

Here it is shown the pseudocode of PPO:

Algorithm 1 PPO, Actor-Critic Style

for *iteration* $= 1, 2, \ldots$ **do**
  for *actor* $= 1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T time steps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt. θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end for

3.2 DDQN

As we have seen in the taxonomy above in the figure 3, the DQN algorithm is model-free RL-algorithm which falls into the Q-learning category. This algorithm develops the Q-learning through a neural network and is further developed in the DDQN case by separating the main network underlying the latter into two different ones.
Before diving into the main aspect of this, let's introduce first the non-double approach by describing the Q-learning concept behind, and then its DDQN variant, which is one among the other possible declination.
The DQN, or Deep Q-Network, is a RL algorithm which approximates a state-value function in a Q-Learning fashion. As the name suggests, this algorithm develops the Q-learning through the usage of a neural network. Let's see briefly how it works: As we have seen above, a basic representation of a RL scenario can be generalized with:

- an agent
- an environment within the agent takes actions
- the state or observation produced by executing a specific action inside the environment and a related reward

In Q-learning, given a new state (or observation) of the environment, the agent chooses the next action to execute by following a table. This table maps the states and the actions to a value, the $Q - value$, which is an estimation of the actual and possible future rewards that an action brings with it if executed considering the actual state. Therefore, the table can be simply implemented as a matrix with the rows as the states (or observation) of the environment which can be assumed and the columns as the actions that can be taken in each specific state. Each cell is filled with the Q-values, initially zeros, and a single row of the matrix represents the possible q-values which can be obtained executing the corresponding action from the respective column. When the agent is in a specific state it can take a new action by going into the related row and selecting the column with the highest Q-value. After executing the action at the time t, the agent gets a reward that is stored in the memory inside a tuple together with the other information as the following:

$$e_t =< s_t, s_{t+1}, a_t, r_t >$$

The elements inside of it are: the state at the time t, the action performed and the state reached at the time t+1, the reward obtained and sometimes a flag representing if the termination has been reached or not. This tuple (together with the others from the others timesteps) becomes part of the memory which is used later on by the algorithm during the learning phase in order to improve the state-action table.
After terminating the task, so when it is completed successfully or a maximum threshold of timesteps has been reached, the agent starts replaying with batches

of the experiences it gathered in its memory in order to improve the Q-values in its table. This is called *Experience Replay*, and is done in order to deal with the instability of the training. The main advantages of doing that are two:

- A more efficient use of previous experience, by learning with it multiple times. This can be really useful when gaining the experience is difficult or expensive (i.e. specific real world data).
- An overall better convergence behaviour and stability during training, due to the fact that batches of samples can be used. In particular batches of random experiences which are shuffled in the memory in order to keep enough diversity in the training data. This allows the network to learn meaningful weights that generalize better the estimate.

A key concept of Q-learning are the Q-values, $Q(s, a)$, which are stored inside the state-action table and can differ a lot from the rewards $R(s, a)$ given by a specific action from the same state. This because even if a reward from an action leads to a temporary higher total reward, the estimated future reward isn't accounted. Therefore, the task could be never be completed, for example if a local maximum is reached in the training, so the Q-values are used. A possible definition is that Q(s,a) is equal to the expected utility starting in s, performing the action a and thereafter performing optimally.
In the algorithm, the Q-values are updated, given a state action pair (s,a), a new state s' and an old estimate Q_k, by doing:

$$\begin{gathered} target = R(s, a, s') + \gamma \ max_{a'} \ Q_k(s', a') \\ Q_{k+1}(s, a) = (1 - \alpha)Q_k(s, a) + \alpha(target) \\ => Q_{k+1}(s, a) = (1 - \alpha)Q_k(s, a) + \alpha[R(s, a, s') + \gamma \ max_{a'} \ Q_k(s', a')] \\ = \mathrm{Q}_k(s, a) + \alpha[R(s, a, s') + \gamma \ max_{a'} \ Q_k(s', a') - Q_k(s, a)] \end{gathered}$$

Where the immediate reward $R(s, a, s')$ is summed(if s' is not a terminal state, otherwise the target value is only composed by the reward) to the discounted maximum of the Q value, obtained by taking the maximum value over all the possible future action a' in the state s'. The discounting factor gamma (a number between 0 and 1, usually 0.99) is used to balance the importance given by the algorithm to future rewards in respect to the actual ones. Then we have a weighted sum of the current and the target estimate, where the old Q_k value is updated by summing it to the target one, both weighted by the hyperparameter $\alpha \in [0, 1]$.

In **DQN**, the state-action table used in Q-learning is not a matrix but is derived through a deep neural network, so the name "Deep Q-Network". In particular, the input will be the state (or observation) of the environment and the output dimension will be of the same number of actions the agent can take. Doing so, the Q-value is estimated by a parametrized Q function $Q_{\theta k}(s, a)$ and the update step becomes:

$$\begin{gathered} target = R_t(s, a, s') + \gamma \ max_{a'} \ Q_{\theta k}(s', a') \\ \theta_{k+1}(s, a) = \theta_k - \alpha \nabla_\theta [\tfrac{1}{2}(Q_{\theta k}(s, a) - (target))^2]|_{\theta=\theta_k} \end{gathered}$$

Where we can see that the same network is used in order to estimate the target value and to evaluate the square loss function for the update of the parameters. While using this standard DQN approach, a possible problem of overestimation my occur. This because the target changes during the learning and . A possible way to overcome this is to use the **DDQN**. The Double Deep Q-Network algorithm uses two identical neural network models, one which learns through the experience replay, as described above, the other one is a copy of the first model but "freezed" at the last learning. While computing the Q-value used to choose the next action, isn't used the network which is being actually updated but instead its previous version. In this way, the overestimation problem described above is dampened.
This approach can be found in the paper "Deep Reinforcement Learning with Double Q-learning" [HGS15], where are used a model Q' for action selection and another model Q for action evaluation.
Finally, we can see how the Q-value function is updated in DDQN:

$$Q^*(s,a) = R_t + \gamma\ Q(s',\ max_{a'}\ Q(s',a'))$$

Where, as before: s, s' are the states at time t,t+1, a is the action taken at time t, R the relative reward, Q the Q-value of the state s' and the action a' which generates the highest Q-value for the main model (the one still untrained on the new samples). From an implementation point of view we have the main network slowly copying the parameters of the other, by periodically hard-copying all the parameters. Another possibility is to copy the updated weights θ' through averaging, i.e. by using the following formula:

$$\theta' = \tau\theta + (1-\tau)\theta'$$

where the rate of averaging τ is usually set to 0.01 (*Polyak averaging*). Finally, to sum what we have sen so far the pseudocode is as shown in fig. 6.
As we can see, after the initialization of the various structures the experience of the exploration of the agent is stored inside the replay buffer as a set of tuples. Action sampled as a_t are selected by an ϵ-greedy policy, therefore are random with a probability ϵ or the one which maximize the Q-value with probability of $1-\epsilon$. During the update step, a sample is extracted from the memory and used to compute the Q-value as described previously. In the end, the weights of the target network are updated using the ones from the main one, by using the *Polyk* averaging(otherwise they can be directly copied each n steps).
Other possible development of the DQN approaches are available in the literature, as the clipped DQN, the noisy DQN and the DQN with Prioritized Experience Replay.

4 Implementation

4.1 Sapientino

Gym sapientino is a OpenAI Gym environment inspired by a kids game called Sapientino [Gia+19]. The environment consists of a grid where a robot moves

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figure 6: DDQN algorithm

on [7], where each cell can be coloured or not, if the robot is on a coloured cell, then it can visit the cell by run a beep.

This is a RL discrete environment and both actions performed by the agent and the observations received by the environment are continuous.

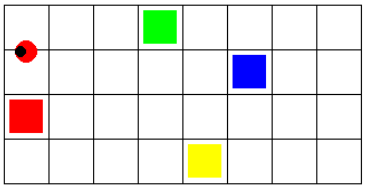


Figure 7: Sample of gym sapientino

The observation state is an array which contains the *position* of the agent (x and y coordinates, sine and cosine of the orientation angle), its *linear* and *angular* velocity. The action space is composed by the actions: *turn left*, *turn right*, *forward*, *null* and *beep*. The first three actions indicate where the agent is moving, the beep action is the agent that is visiting a cell and the null is simply an action that indicate the robot is doing nothing.
When the agent performs an action, the environment returns an observation on the state and on the automaton state and a reward. The reward is different from zero if and only if the agent has successfully visited the last colored cell and all cells in the right order. In section 4.4.3 is described how we change the reward function.

4.2 Parameters configuration

We configure a *config.cfg* file to store all the parameters before starting our training. Here is shown an example:

```
[ENVIRONMENT]
map_file = MAP.txt
reward_per_step = 0.0
reward_outside_grid = 0.0
reward_duplicate_beep = 0.0
tg_reward = 500.0
reward_ldlf =  <!red*; red; !yellow*; yellow; !blue*; blue>end
max_timesteps = 400
name_dir_experiment = case4

[AGENT]
algorithm = PPO
initial_position_x = 0
initial_position_y = 0
max_velocity = 0.4
min_velocity = 0.0
acceleration = 0.2
angular_acceleration = 10.0

[TENSORFORCE]
batch_size = 128
memory = 128
learning_rate_initial_value = 0.001
learning_rate_final_value = 0.00001
exploration_initial_value = 0.8
exploration_final_value = 0.005
learning_rate_decay_value = 0.0000025
exploration_decay_value = 0.00
entropy_bonus = 0.0
hidden_size = 64
discount = 0.99
update_frequency = 20
target_sync_frequency = 3
target_update_weights = 0.8

[RUNNER]
episodes = 1500
goal_reward_reduction_rate = 1.00
```

Listing 1: Example of a config file for a map of three colors

As you can see the file is divided into four sections:

* Environment: are stored information including the reward function and the map chosen.

* Agent: which algorithm is chosen between PPO and DDPG, and information on initial position and velocity of the agent.

* Tensorforce: as we used the tensorforce library here we sum up all the parameters that are necessary for using the Environment and Agent class of tensorforce.

* Runner: are the parameters used in Runner class that will be explained in section 4.4

4.3 Model

As said previously we decided to use the tensorforce library that implements RL Agents, the class Agent gives the possibility too customize a network by giving a dictionary that describes the NN as argument. In figure 8 is shown the structure of the model, The network is very simple formed by two dense layers followed by a tahn activation function. Before feeding the input in the dense layer a mask is applied depending on the automaton state of the agent. In the figure 9 for instance the map has three color: red, yellow and blue to be visited in this order, until the agent doesn't reach the red cell the mask will have a 1 on the red circles of the image otherwise 0, similarly if it has visited the red and now it has to reach the yellow cell the mask will have 1 on yellow circles and on the other 0 (same reasoning for blue cell).

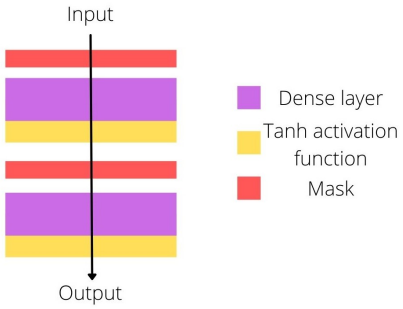


Figure 8: Structure of the model

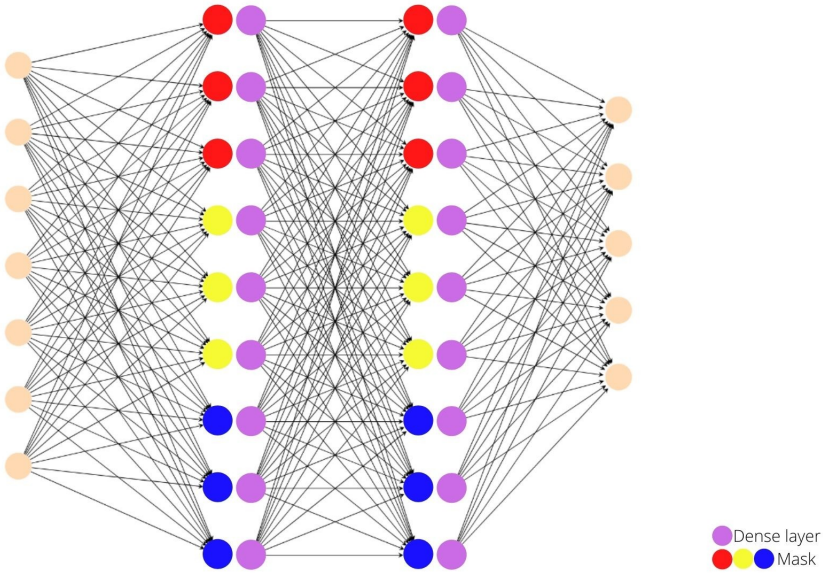


Figure 9: Graph of the model, depending on the automaton state just some neurons are used in the model.

4.4 Runner

We implement a class Runner along the lines of the Runner class of tensorforce. Our Runner has two main function: *train* and *evaluate*.

4.4.1 Train

This function is called when is required to train the agent. Here is the pseudocode of train function, s is the state, a the action and r the reward:

Algorithm 2 Train

```
for episode = 1, 2, ..., N do
    terminal = False
    s ← reset environment
    while not terminal do
        a ← get action given state s
        s, terminal, r ← execute action a
        Agent observe reward
        if terminal then
            reset environment
        end if
    end while
end for
```

4.4.2 Evaluate

This function is used to evaluate the agent, is similar to the algorithm 2 with the difference that now the agent does not observe the action and therefore it does not update its weights. Below you can see the pseudocode for the evaluation function, s is the state, a the action and r the reward:

Algorithm 3 Evaluate

```
for episode = 1, 2, ..., N do
    terminal = False
    s ← reset environment
    while not terminal do
        a ← get action given state s
        s, terminal, r ← execute action a
    end while
end for
```

4.4.3 Reward shaping

In order to shape the reward given to the agent we developed a function to customize it. Usually the agent receives the full reward only if the final state is

reached but we wanted to shape it differently.
The rewards can be given at different points of the training relying on how the agent moves accordingly to the task given. They can be negative in the case of unnecessary movements and actions, or positive in the case the agent moves closer to the desired goal.
In our case we have to follow a specific sequence of states therefore in this function we assign to the agent a non-negative reward r each time it visits a colored cell in the right order.
To improve this we added also the possibility to unbalance this reward, by using the *goal_reward_reduction_rate* parameter. In this way we decrease the rewards given at each state to increase the one given at the end. This is done in order to increase the relevance of the actions performed by the agents to successfully complete the task. In the end the total reward remains the same but the values given are not equal among the various states.
The function used in the Runner class in order to get the right reward is the following:

```
def get_reward_from_automaton_state(self, reward,
    current_automaton_state, previous_automaton_state, terminal,
    counter_array):
    for i in range(1, self.number_of_experts+1):
        if current_automaton_state == i and
    previous_automaton_state == i-1:
            reward += self.reward_step
            counter_array[i-1]+=1
            if current_automaton_state == self.number_of_experts:
                terminal = True
                reward += self.reward_step*(1-self.
    goal_reward_reduction_rate)*(self.number_of_experts)
            return reward, terminal, True
    return reward, terminal, False
```

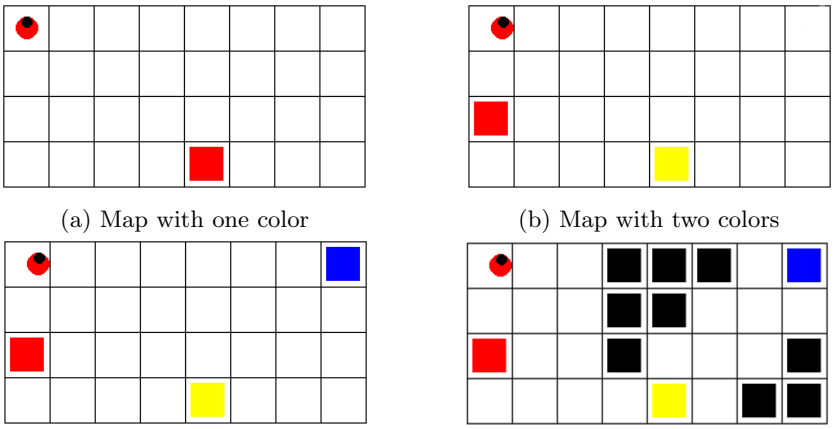
Listing 2: Reward shaping function

5 Results

In this section we will show the results obtained for both the algorithms. We will show cases which are identical for PPO and DDQN in terms of complexity of the task(same map) and cases more or less complex based on the interestingness of them.

5.1 PPO

The first test was executed on a simple map with a simple goal composed of the task of reaching just one single color, since we wanted to evaluate the algorithm in scenarios of increasing difficulty (see figure 10). The map is reported in the figure 10a, as we can see, since the map is extremely simple, the agent achieve to master the task in 500 epochs, gaining an increasing reward over the epochs



(a) Map with one color

(b) Map with two colors

(c) Map with three colors

(d) Map with three colors and obstacles

Figure 10: All the maps used during the training

and succeeding at the task around 80% of the times. It is also possible to notice

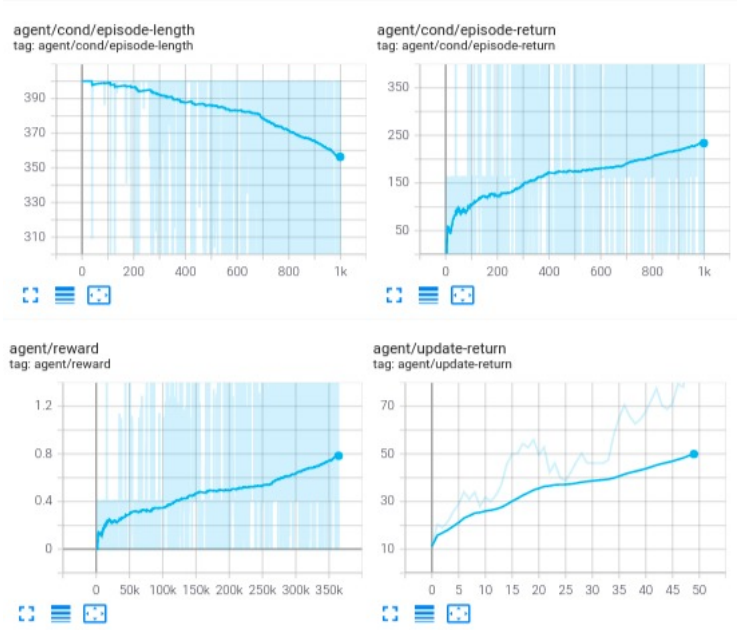


Figure 11: Relevant graphs of the performance of PPO on map in figure 10b

that the length of each timestep decrease overtime, which means that the agent learn how to solve the task more efficiently after each iteration.
After this test run, we moved to a non markovian task composed of two colors.

In this case, the results are shown in figure 11. As we can see, the behavior is steady and the convergence is reached smoothly.
After this intermediate run, we tackled an even more complex task, where three colors were needed to be crossed in sequence.
In figure 10c is reported the map, where we can see the three colors positioned. In such environment, in figure 12 are shown the results obtained from the training. As we can see, the learning from the graphs is an obvious behavior. Such

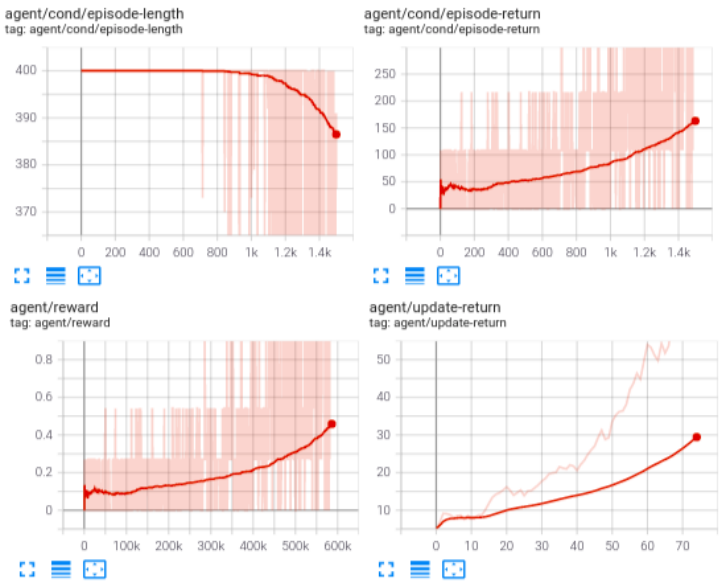


Figure 12: Relevant graphs of the performance of PPO on map in figure 10c

solution was obtained with quite an amount of epochs, which in this case was 1400.
To conclude the tests, we tried on pushing to the extent the algorithm, testing it in a more complex environment, with obstacles obstructing the path.
In figure 11 is reported the map, in which can be clearly see the presence of obstacles. The results in this case are shown in figure 11.
Regardless of the complexity, the algorithm managed to solve the task. Let's point out two interesting aspects: the first one is that the algorithm is still converging, but not with the same steepness as in the previous case; this is because of the more complex task, and it was an expected behavior. The second one is that such result was obtained in the same amount of epochs of the previous case, 1400; this means that the complexity handling of the algorithm seems to scale well with the complexity of the task.

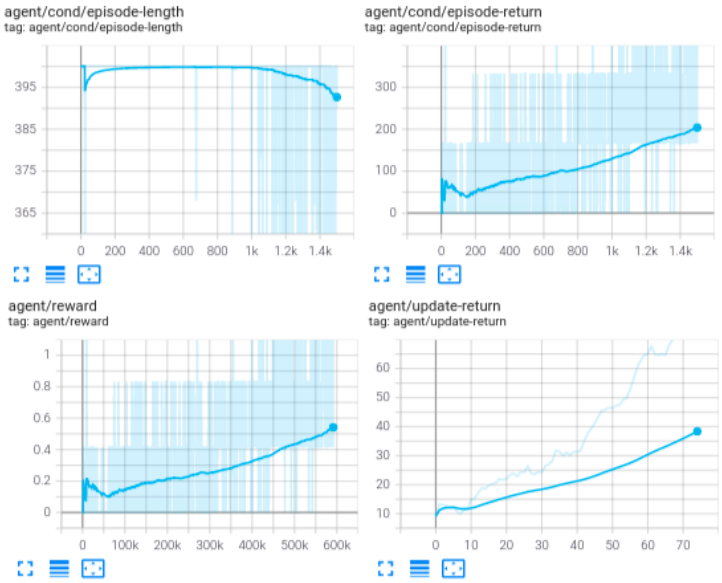
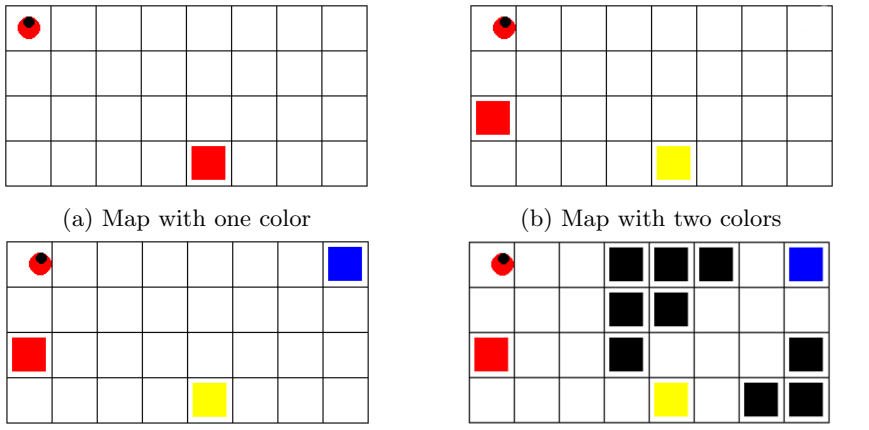


Figure 13: Relevant graphs of the performance of PPO on map in figure 10d

5.2 DDQN

To create a baseline of confrontation, we decided to run the other algorithm, DDQN, on the same maps used before for PPO, which are reported again in the figure below, 14.



(a) Map with one color

(b) Map with two colors

(c) Map with three colors

(d) Map with three colors and obstacles

Figure 14: All the maps used during the training

In the beginning, in order to implement and fine tune the DDQN case we started as before with a simple markovian scenario counting a single goal to be reached by the agent(in red, fig.14a). The default configuration showed some suboptimal results with no convergence. To achieve it, we decided to focus first on the fixed value of the learning rate and exploration rate hyperparameters, discovering an improvement of the result if set at 0.65 for the exploration rate and 1e-3 for the learning rate. In this way some results were obtained but with an unstable trend. To overcome this problem, we decided to choose a time decaying implementation for these two values, thus decreasing gradually the exploration(thus increasing the exploitation of the policy) and decreasing the learning steps while reaching the minimum. This has been done linearly with the following code:

```
'learning_rate': dict(
type='linear', unit='episodes', num_steps=EPISODES,
                initial_value= LR_INIT,
                final_value=LR_FINAL),
'exploration': dict(
type='linear', unit='episodes', num_steps=EPISODES,
                initial_value=EXP_INIT,
                final_value=EXP_FINAL),
```

Listing 3: hyperparameters linear decay

Together with a bigger replay memory(simply called *memory*, in the code) and a shorter episode length, we obtained a much better results and after only 300 episodes we can see a complete convergence to the desired result. The learning

rate variables above had an initial and final values of 1e-3 and 1e-4, while the exploration ones were from 0.75 to 0.01. In the fig.15 below the results are shown.

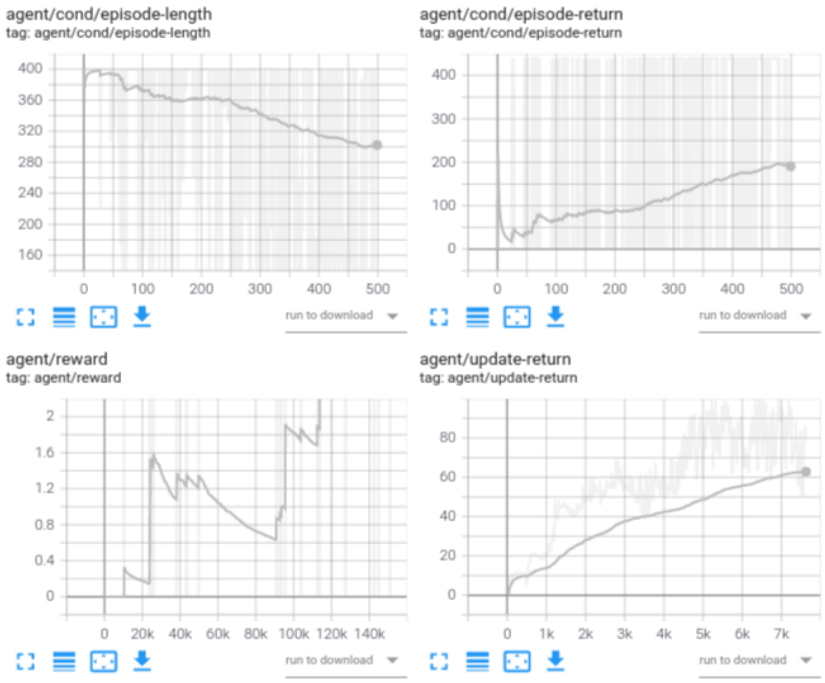


Figure 15: Graphs of the performance of the DDQN algorithm tested in the map with one color in figure 14a

Then we moved to the harder, non markovian tasks, with two and three colors. The starting configuration was similar to the one developed for the first case in both of these two new scenarios (shown in the fig.14b,14c). In particular, we found that a longer training with a network hidden size of 16 times the number of colors and some other minor changes were enough to reach the convergence. Graph results are shown at the end of this section, together with the next test. The last case increase even more the difficulty by introducing some obstacles into the map and reducing the possible ways the agent can exploit in order to reach each goal. As we can see in fig.14d, only a 1 block space is available to reach the yellow and blue goal. Moreover, here the agent encountered new difficulties as the possibility to avoid being blocked inside a corner between two walls, encountered much more frequently than before when this phenomena was possible only in the corners of the map. So we conducted different experiments trying to achieve convergence, which we obtained after some minor fine tuning of the parameters in respect to the case in fig.14c.

The trend of those last three cases, so the tests on the map with two, three colors and three colors with obstacles are the following:

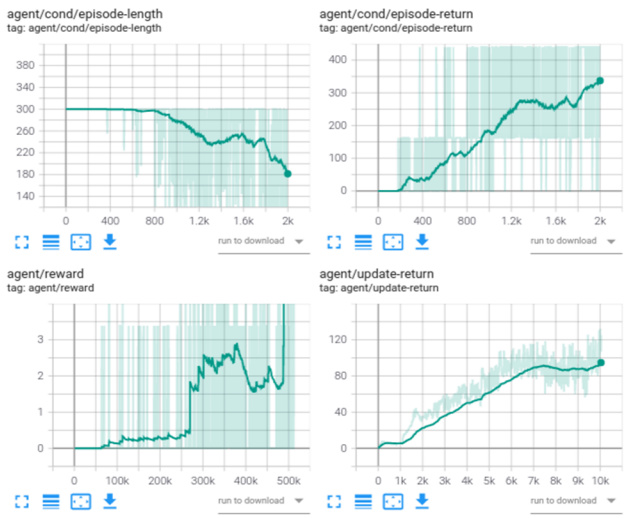


Figure 16: Graphs of the performance of the DDQN algorithm tested in the map with two color in figure 14b

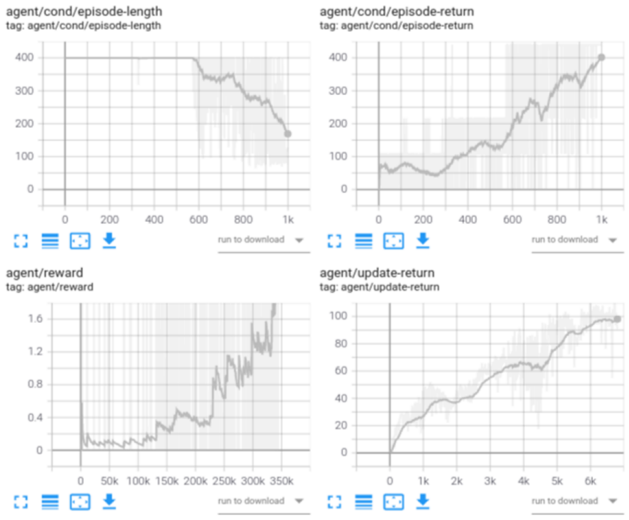


Figure 17: Graphs of the performance of the DDQN algorithm tested in the map with three color in figure 14c

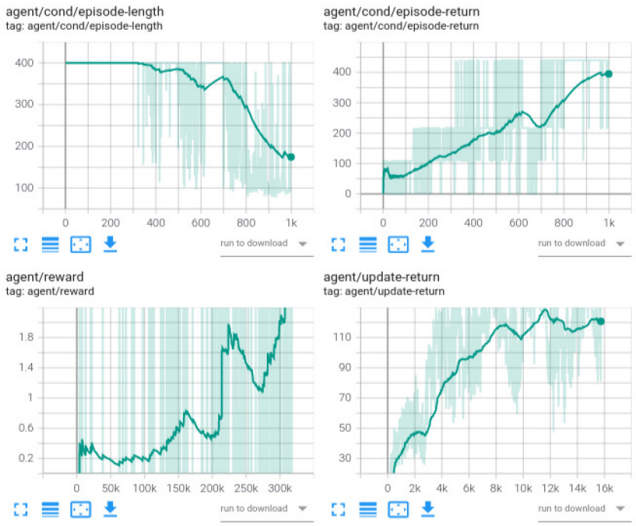


Figure 18: Graphs of the performance of the DDQN algorithm tested in the map with three color with obstacles in figure 14d

6 Conclusion

Conducting these analysis on the same environment with two different algorithms allowed us to compare the two solutions. One of the main things that we understood through the tests and work is that both the algorithms can, in general, solve the task in the given environment. Solutions are also obtained when the environment is made more complex, showing their behavior when the complexity of the task is scaled up.
Running tests with both the algorithms though, it was clear that, generally speaking, the convergence of DDQN was slower if not coupled with a more delicate fine tuning with respect to PPO.
Also, PPO appear to be a generally faster algorithm, since it is capable of reaching a convergence behavior in less epochs than DDQN; this was also shown by the work of one of the previous groups that worked on such environment using PPO as main algorithm for their analysis.
Another fact to point out is that PPO seems to be better at handling complexity scalability, since the differences between the test with three colors tests with three colors and obstacles, are almost the exact same, while for DDQN is it possible to notice an oscillating behavior making its appearance between the simple three color map and the one with the obstacles.
To conclude, the results were still satisfactory for both of the algorithms, and this allowed us to demonstrate how the environment does not reject a solution respect to the other, but in general has just a tendency on being solved faster by one algorithm compared to the other.

References

[Pnu77] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE. 1977, pp. 46–57.

[TV05] Deian Tabakov and Moshe Y. Vardi. "Experimental Evaluation of Classical Automata Constructions". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Geoff Sutcliffe and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 396–411.

[DN08] Peter Dayan and Yael Niv. "Reinforcement learning: the good, the bad and the ugly". In: *Current opinion in neurobiology* 18.2 (2008), pp. 185–196.

[DV13] Giuseppe De Giacomo and Moshe Y Vardi. "Linear temporal logic and linear dynamic logic on finite traces". In: *Twenty-Third International Joint Conference on Artificial Intelligence*. 2013.

[HGS15] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: `1509.06461 [cs.LG]`.

[Sch+17] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[BGP18] Ronen Brafman, Giuseppe De Giacomo, and Fabio Patrizi. *LTLF/LDLF Non-Markovian rewards*. 2018. URL: `https://ojs.aaai.org/index.php/AAAI/article/view/11572`.

[Gia+19] Giuseppe De Giacomo et al. "Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications". In: *ICAPS*. 2019.

[Ant21] Luca Lobefaro Antonella Angrisani Andrea Fanti. *Policy Networks for Non-Markovian Reinforcement Learning Rewards*. July 2021. URL: `https://github.com/lucalobefaro/reasoning_agent_project/blob/main/Policy_Networks_for_Non_Markovian_Reinforcement_Learning_Rewards.pdf`.