Reasoning Agents Project

Cardia Francesca
Coppa Federica
Del Vecchio Andrea

July 2021

1 Introduction

The aim of this project consists in developing a non-Markovian agent that solves *Sapientino case environment*. The interesting aspect of the problem at hand is that the goal is characterized by a sequence of actions that a standard RL agent could not solve. For this reason we combined a RL agent with an automaton. Specifically, the algorithm we use for the agent is *Proximal Policy Optimization* (PPO) and the automata are Deterministic Finite Automata (DFAs).

2 Deterministic Finite Automata

A Deterministic Finite Automaton (DFA) (as for instance (1)) is a mathematical model that maps an input sequence to an output. The result is that the computation is unique.A DFA can be exactly in one state at a given time and it makes the transition from one state to another state given some input and according to the transition function.
It is characterized by a tuple $\langle Q, \Sigma, \delta, q_o, F \rangle$, where Q is the set of states, Σ is the set of symbols (alphabet), δ is the transition function $Q \times \Sigma \longrightarrow Q$ that takes as input a state and a symbol and returns a state, $q_o \in Q$ is the initial state and $F \subseteq Q$ is the set of final states (or accepting states).

3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a family of policy gradient algorithms that became popular in recent years (3). A policy gradient algorithm defines its objective in terms of the **gradient of the logarithm of the policy**, which, in this case, is a parametric probability distribution on the available actions in the current state.

$$\hat{g} = \mathbb{E}[\nabla_\theta \log(\pi_\theta(a_t|s_t))A_t] \tag{1}$$

PPO most common implementations represent $\pi_\theta(a_t|s_{t)}$ with a (deep) neural network which takes as input a state representation s_t and produces a probability distribution over the action space π_θ. θ is the set of parameters of the neural network which get updated to maximize the policy gradient objective, usually by means of a stochastic gradient ascent algorithm. To optimize the policy gradient objective the algorithm should maximize not only the logarithmic term, which represents the contribution of the policy to the objective function, but also the **advantage term** A_t which can be interpreted as a measure of the benefits of taking that particular action a_t in the time step t.
PPO original version defines the advantage function as:

$$A_t = \sum_{k=1}^{T} \gamma^k r_{t+k} \; - V_{\theta_V}(s_t) \tag{2}$$

It is essentially a difference between the actual value of the action which is expressed in terms of the **expected discounted cumulative reward** (first term in equation [2])

and a baseline estimate of the value of the state s_t following that policy (the second term in the equation [2]). The baseline estimate is an estimation of the value of the cumulative discounted future reward given that action selection. It is a parametric function and, in deep reinforcement learning, is implemented as a neural network as well, called **baseline network**, which takes as input the current state s_t and produces an estimate of the future reward given that state and following the current policy.
To give a clearer explanation of the meaning of the advantage function, it is convenient to study both the two cases in which A_t is greater or smaller than 0:

- $A_t > 0$: it means that the action has produced a (discounted, cumulative) reward which is better than the baseline estimate. For this reason, we address the execution of the action a_t as advantageous and we would like that the algorithm increases the probability of selecting the same action in the future (given the same state).
- $A_t < 0$: it is the opposite case. The action is disadvantageous and we would like to decrease the probability of its selection in the future.

In order to avoid to produce excessively large updates of the policy, which is a common problem of policy gradient algorithms, PPO substitutes the policy gradient objective [1] with the **clipped surrogate** objective function:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}[\min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t] \tag{3}$$

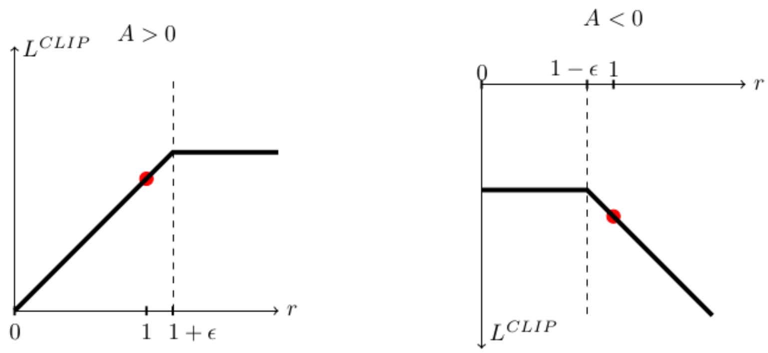


Figure 1: Figure representing the effect of the clipping operation on the surrogate objective function.

Where the expected value, in this case, is taken as the empirical average of a batch of samples.
The variable $r_t(\theta)$ is defined as

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{4}$$

and represents the probability ratio between the currently considered policy and the older policy, which is the policy of the agent before the last update.
If $r_t(\theta) > 1$, the action a_t is more likely to be selected in the current policy than in the older policy, and we have $r_t(\theta) \in [0, 1)$ in the opposite scenario.
The clipped surrogate objective forces the algorithm to perform conservative updates if the advantage estimates A_t become too large in magnitude, by clipping the ratio r_t to $1-\epsilon$ or $1+\epsilon$. An update is said to be conservative if the updated policy is not *"too far away"* (in terms of the Kullback-Leibler divergence
[1])
from the older policy.
ϵ is an hyperparameter of the algorithm and must be defined before istantiating the agent. The authors of the paper suggest using $\epsilon = 0.2$ but other choices are possible.

The most common implementation of the clipped surrogate objective function is composed by three terms.

$$L^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}[L^{CLIP}(\theta) + c1L^{VF}(\theta) + c2S[\pi_\theta](s_t)] \tag{5}$$

[1] The Kullback-Leibler divergence measures the distance between two probability distributions and is defined as $KB(P||Q = \sum_i P(i) \log_2 \left(\frac{P(i)}{Q(i)})\right)$ where P and Q are two discrete probability distributions.

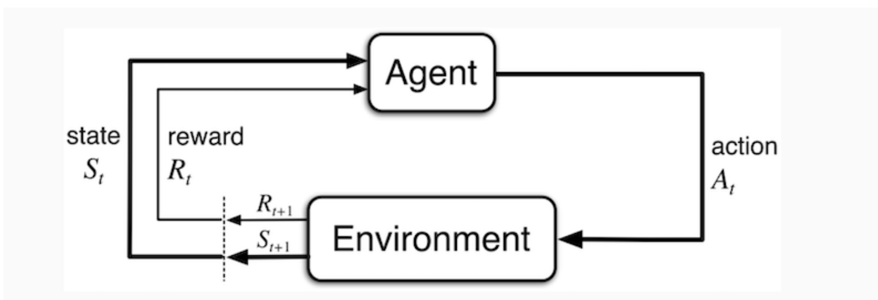
The second term is a *squared loss* term associated to the output of the basline network.

$$L^{VF} = (V_\theta(s_t) - V_t^{targ})^2 \tag{6}$$

The combination of the clipped surrogate objective term and of the value function term allows the algorithm to share parameters between the policy and the baseline networks. c_3 is the coefficient of the **entropy bonus** which ensures sufficient exploration.

4 Markov Decision Process

The general setting of Reinforcement Learning is the following:



The agent at the state S_t executes an action A_t and receives from the environment an associated reward. Most of RL algorithms assume that the environment can be modeled as MDP.
Markov Decision Processes (MDPs) are a central model for sequential making under uncertainty. A Markov Decision Process (MDP) is a tuple $\langle S, A, T, R, \gamma \rangle$, where S are the states, A are the actions called action space, T the transition function $T : S \times A \longrightarrow Prob(S)$ that returns for every state s and action a a distribution over the next state $(T(s, a, s') = P(s'|s, a))$. R represents the reward function $R : S \times A \longrightarrow \mathbb{R}$ that specifies the real valued reward received by the agent when applying action a in state s $(R(s, a, s') \in \mathbb{R})$ and γ represents the discount factor.

The next state s_{t+1} is not dependent from s_{t-1}, and consequently is not dependent from the history. In fact looking at 2 we can observe that if we remove s_t the graph becomes disconnected.
Then the following properties hold:

$$s_{t+1} \perp s_0, ..., s_{t-1} | s_t \quad \forall t \tag{7}$$

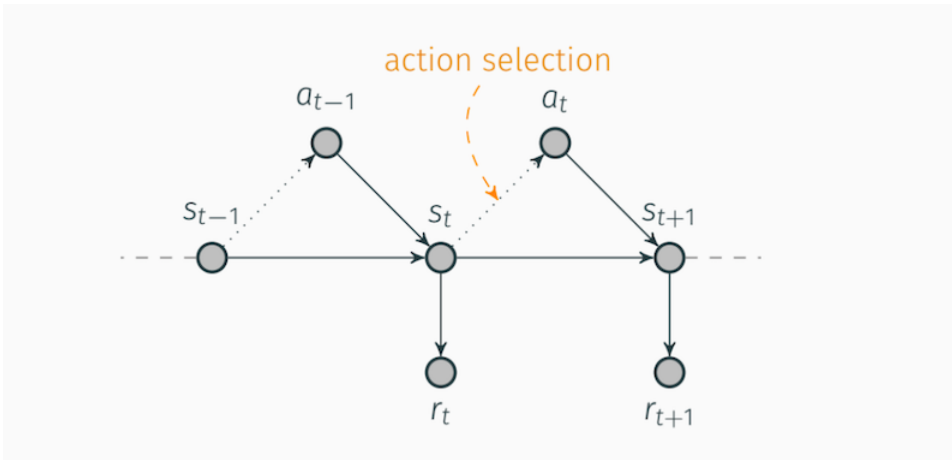


Figure 2: MDP example

$$r_{t+1} \perp s_0, ..., s_{t-1} | s_t \quad \forall t \tag{8}$$

So the next state is not dependent from the history if the current state s_t is given. The same holds for rewards. A solution to an MDP is called *policy*, and it assigns an action to each state, possibly conditioned on past states and actions. Every MDP has an *optimal policy*, which maximizes the expected sum of rewards for every starting state $s \in S$.

5 Non Markovian Rewards Decision Processes

A Non Markovian Decision Process is a stochastic process that does not exhibit the Markov properties [7], [8] (generalized as memoryless property). A Non Markovian Reward Decision Process (NMRDPs) is a tuple $M = \langle S, A, T, R, \gamma \rangle$, where S,A, T, γ are equivalent to MDPs case, while R is redefined as $R : (S \times A)^* \longrightarrow \mathbb{R}$.
Now the reward is a real valued function over finite state-action sequences (according to a specific history).

6 Gym-Sapientino

This is a new RL discrete environment that respects the gym interface (4).
It is characterized by a map which contains a sequence of colors (represented by colored cells). The agent can move in the map executing actions that have continuous effects on the state, moreover also the observations received from the

environment are continuous. The observations that are continuous on the state, are characterized by the *position*, the *linear velocity* and the *angular velocity*. In fact the agent executes an action, the environment returns an associated reward and an observation on the state and on automaton state.

6.1 Temporal Goal on Gym-Sapientino

The non markovian agent moves in the map executing a non markovian task (the ordered visit of a sequence of colors). In particular it should reach temporal goals, described properly by a specific sequence of colors. When the agent reach a temporal goal it receives a reward from the environment, and more frequently it earns rewards more it will be prodded to learn. This environment allows to work with four different colors (and consequently with a temporal goal characterized by four components):
yellow, **blue**, **green**, **red**.
It is possible to specify which colors are included in the temporal goal formulas ϕ, and consequently in the related automaton A_ϕ (which is provided by the environment). In the automaton the states related to the colors are specified through numbered codification. Moreover there is a particular failure state, denoted previously as *SINK*, that we have converted into numbered codification using 2 to indicate it. Then for instance, if we consider a map characterized by two colors, for instance blue and green, the related automata will be composed by four numbered states as showed in 3 :

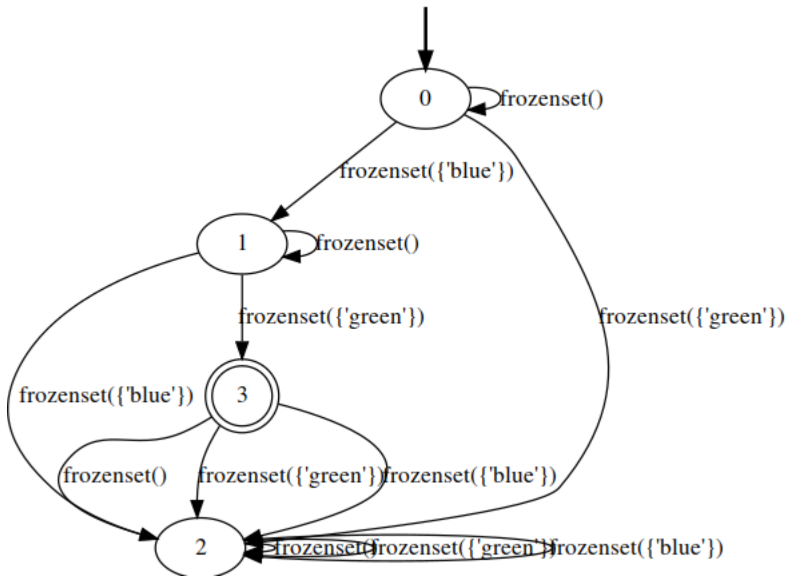


Figure 3: Representation of the problem at hand in term of automata

Notice that `0` is the initial state and `3` is the accepting state. In this case the agent can reach the temporal goal visiting the following sequence: `0,1,3`. In order to control the correct achievement of the temporal goal, we have also interfaced with the following problem:
in Gym-Sapientino environment the agent can execute some actions signaled by five particular strings: `LEFT`, `RIGHT`, `FORWARD`, `NULL`, `BEEP`. The first,the second and the third need to indicate in which direction the agent is moving, the fourth needs to signal that the agent is not moving and the last needs to signal that the agent is visiting a certain state. The problem rises when the agent enters in the same state twice consecutively, and this implies a double `BEEP` visualization. In our work we have controlled this problem, avoiding that the actual state reached by the agent was equal to the previous reached state. In this case we impose that the temporal goal is satisfied only when the sequence of states is does not include equal states. For instance in the previous example with two colors (blue and green):
the temporal goal is reached when is visited this sequence `0,1,3` and not `0,1,1,3` or `0,1,3,3`.

7 The non markovian agent

Since we have to solve a non markovian problem, the agent must take into account past experience in order to select the optimal action. In this case, the agent is said to be **non markovian**. To build a non markovian agent, we might start considering the problem of visiting the color sequence as a composition of a number of smaller sub-problems: for example, given the goal {*blue*, *red*, *green*}, we extract three sub-problems:

1. Visit the blue color.
2. **Then** visit the red color.
3. **Finally** visit the green color.

In this way we end up reducing the original, non markovian problem, to a set of markovian problems which we can solve **sequentially** using SOA methods. Moreover, this approach completely cut out the issue of keeping track of past state, action and reward traces.
With this in mind, we can build 3
[2]

separate markovian agents and train them to reach the corresponding colored tile in the map (figure [7], [10]).

However, at any given time, only one markovian agent can select an action to execute for the non markovian policy (figure [4]).
The action selection is constrained by the goal formulation. In fact, we consider the gym Sapientino task to be solved if and only if the non markovian agent visits the colors **in the order they appear in the temporal goal formula**. In fact, if the agent visits the colors in the wrong order (for example {*blue*,*green*, *red*} instead of {*blue*, *red*, *green*})

the automaton sinks and episode terminates.

To reach the goal,then, the agent must be aware of which color in the goal sequence it has already visited and which color is the next one in the sequence. Such an information may be inferred by inspecting the states of the goal DFA. (see the example in figure [3]. Luckily the environment allows the agent to keep

[2] In general, the more colors are involved in the goal sequence, the higher is the number of markovian agents we have to train. In the case of a sequence of the type {blue,green,red,blue} there is no need to build an additional agent for the last blue color in the sequence, as we already have at our disposal an agent which can reach the blue color (the first one). **This happens because agents are associated to colors and not to sequence elements**. Be aware that, theoretically, the proposed implementation supports sequences of arbitrary size.

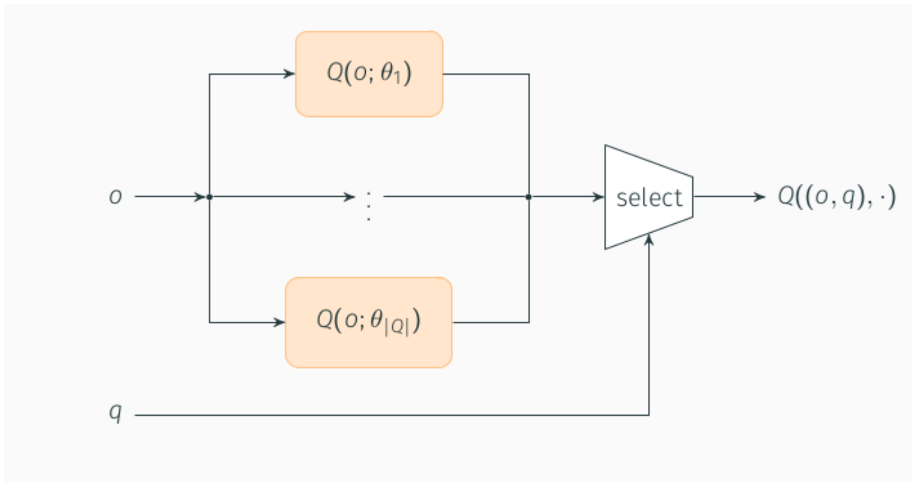


Figure 4: Baseline implementation for the non markovian policy network. It takes as input both the observation o and the state of the DFA q. Then, according to the state of the automaton, the agents selects one of the $|Q|$ *separate experts* (the markovian agents we have discussed so far) for the action selection. This image is taken from the slides of Roberto Cipollone for the Reasoning Agent course held in Sapienza in 2020/2021.

track of the automaton states. Therefore, in order to visit the goal, it is sufficient to select, from time to time, the ”correct” markovian agent depending on the automaton state. In the section below, we present our solution for the expert selection in the context of a popular deep reinforcement learning open source framework. The approach focuses on efficiency and network parameter sharing and is based on parallel computation and automatic differentiation.

8 Implementation Details

For the agent implementation we have used Tensorforce [3], an open source library for deep reinforcement learning based on Tensorflow. Tensorforce supports a variety of deep RL agents like DQN, PPO, DDPG and features the most common neural network architectures (dense, recurrent, convolutional, attention based) for policy network implementation.
Due to its ease of use and the great effectiveness, Tensorforce is one of the most frequently used frameworks for reinforcement learning experiments. However,

[3] The latest version of Tensorforce documentation can be found at https://tensorforce.readthedocs.io/en/latest/

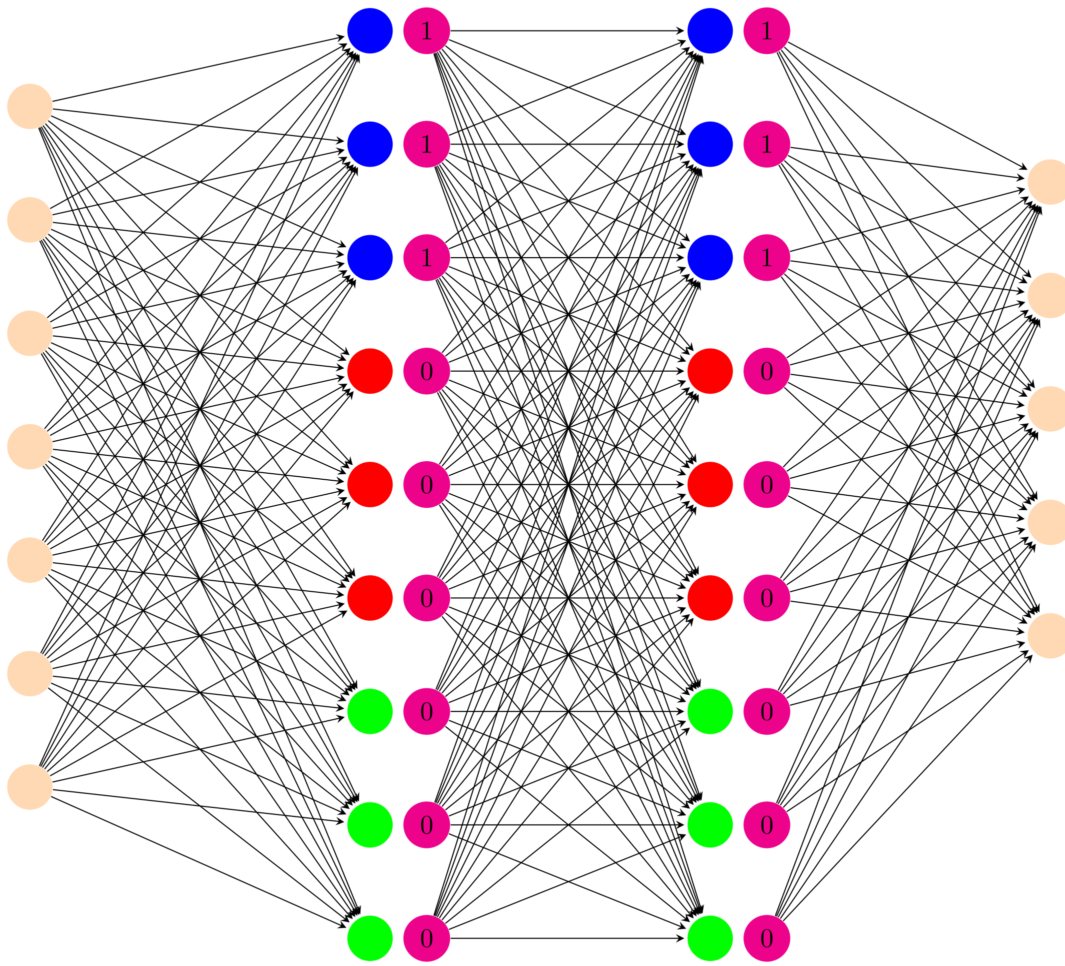


Figure 5: This sketch represents the non markovian policy network scheme implemented as a two hidden layer fully connected neural neural network. The architecture is employed in solving the problem of visiting a sequence of three colors (blue , red and green). In magenta, we represent the automaton state binary vector as described in the section [8]. The blue neurons represent the policy network of the markovian agent that is trained to reach the blue color; same for the red and the green neurons.

as far as we know, the latest version of the library (Tensorforce 0.6.4) does not support non markovian agents by default. Therefore, for the purpose of our project, we must manipulate the tensorforce agent policy networks and adapt them to a non markovian framework.

In this work, we propose a non markovian agent implementation which employs a single policy network for action selection. The architecture is divided in *chunks* (or portions), each one assigned to a different markovian agent (or color)(figure 5. Each chunk has the same size and acts as **policy network** for the markovian sub-problem.

It is worth including brief textual description of the training loop algorithm of the non markovian agent for a clearer explanation: For each step in a given episode, we sample the environment state, a seven elements vector, and the state of the automaton, an integer number. Afterwards, we create a binary *version* of the automaton state which is built as follows:

1. Instantiate a vector filled with zeros of size $H * N$ where H is the size of the hidden layers and N is the number of automaton states.

2. Divide the encoded state vector into equally sized sub-portions. Each chuck will have a size of H.

3. Assign the sub-portions to a different automaton state.

4. Fill the part corresponding to the **current automaton state** with *ones* and leave the rest with *zeros*.

5. Multiply the binary vector to the output of each dense layer to **select**, from time to time, the chunk of the network of interest.

In this way the neurons of the network multiplied by the zeros in the binary vector do not contribute to the action selection (they are *"zeroed"*) and **not get updated during backpropagation**.

Tensorforce default PPO implementation allows parameter sharing between the baseline network and the policy network.

For this reason, we have used the same network for both action selection and action value estimation. This means that also the state baseline estimates strictly depend on the automaton state and thus, on the color of the sequence which should be reached at any given time.

In figure [5] we show a graphical example of a forward pass in the non markovian policy network. The input layer receives as input the 7 dimensional state vector returned by gym Sapientino environment at each iteration, while the (two) hidden layers are subdivided into three parts each one corresponding to a different color. The correct network sub-portion is selected by multiplying the binary vector, which is built according to the algorithm sketched above, to the output of each hidden layer. In this figure we sketch an example of execution in which we select the first three neurons only from the two hidden layers (the binary vector contains 1 in the first three elements and 0 in all the other elements). In the example, they correspond to the network sub-portion associated to the markovian agent which is learning (has learnt) to visit the blue color and so, this means that the agent action selection leans towards reaching the blue color in the environment.

8.1 Network Description

We have implemented a custom network in the following way (figure [5]):

```
network=dict(type = 'custom', layers= [dict(type = 'retrieve',
    tensors= ['gymtpl0']),dict(type = 'linear_normalization'),dict(
    type='dense', bias = True,activation ='tanh',size=
    AUTOMATON_STATE_ENCODING_SIZE), dict(type= 'register',tensor =
    'gymtpl0-dense1'),

#Perform the product between the one hot encoding of the automaton
    and the output of the dense layer.

dict(type = 'retrieve',tensors=['gymtpl0-dense1','gymtpl1'],
    aggregation = 'product'),dict(type='dense', bias = True,
    activation ='tanh',size=AUTOMATON_STATE_ENCODING_SIZE),dict(
    type= 'register',tensor = 'gymtpl0-dense2'), dict(type = '
    retrieve',tensors=['gymtpl0-dense2','gymtpl1'], aggregation = '
    product'),dict(type='register',tensor = 'gymtpl0-embeddings')
    ,],)
```

This network is characterized by:

- **Retrieve layers**: they are useful when defining more complex network architectures which do not follow the sequential layer-stack pattern, for instance when there are multiple inputs. It allows to aggregate multiple inputs through concatenation, product,sum operations (in our case we have used the product).
- **Linear normalization layer**: which scales and shifts the input to [-2.0, 2.0], for bounded states with min/max_value.
- **Register layers**: they are an other type of retrieval layers useful for complex networks and for multiple inputs.
- **Dense layers**: they are Dense fully-connected layers.

8.2 Reward shaping

Most of the techniques in Reinforcement Learning (RL) assumes that the learning starts from a blank slate and improves only by means of trial and error. This learning approach takes a huge amount of trials and as a direct consequence it implies that the time required is a lot. This is why we talk about **reward shaping** (2). This method is useful to incorporate domain knowledge in the RL agents. In reward shaping, the domain knowledge is represented as a supplementary reward that allows the RL agent to learn more efficiently. In our code the **Reward shaping** has been implemented and applied as showed in 6, where according to 3 the agent received a great reward of 500 when reaching the right state, otherwise in case of *SINK_ID* the reward is -500.
In particular in our case the reward shaping technique is useful because the reward is positive and scattered. Moreover, we can monitor when the agent is approaching to the goal, in fact the path in the automata is "obligated", so this is a good reason to reward the agent inciting to proceed.

```
#Reward Shaping
if automaton_state == SINK_ID:
    reward = -500.0
    terminal = True


elif automaton_state == 1 and prevAutState != 1:
    reward = 500.0

elif automaton_state == 3:
    reward = 500.0
    print("Visited the goal in episode: ", episode)
```

Figure 6: Reward shaping

9 Experimental environment

In the following sections we discuss some experiment results on different gym Sapientino maps for a number temporal goals.
All the experiments we discuss are run on local machines with no GPUs equipped. This is possible as the network size is relatively small and . All the agents have been trained for a small number of episodes. The two color agents trained for 1000 episodes, whereas the three color agents trained for 2000 episodes. We used the hyperparameter configuration which is reported in table [5] for all the

experiments.

10 Trials

We have done progressive trials in order to arrive to the situation in which the agent can reach a temporal goal characterized by three colors. We have training using CPU and we have collected the training plots using **tensorboard**.
In particular we focus the attention on the following plots:

- **agent/cond/episode-length**: the mean length of each episode in the environment for all agents.
- **agent/cond/episode-return**:the discounted rewards during the episodes.

11 Experiment with two colors with a small map

In the next step we have tested maps characterized by two colors, and in particular during the training we have monitored how many times the agent reach the temporal goal (determined by the right sequence of visited colors) for achieving the convergence.
The sequence of colors the agent should visit are, in this case $\{blue, \; green\}$ Below we show the gym Sapientino map for the experiment.

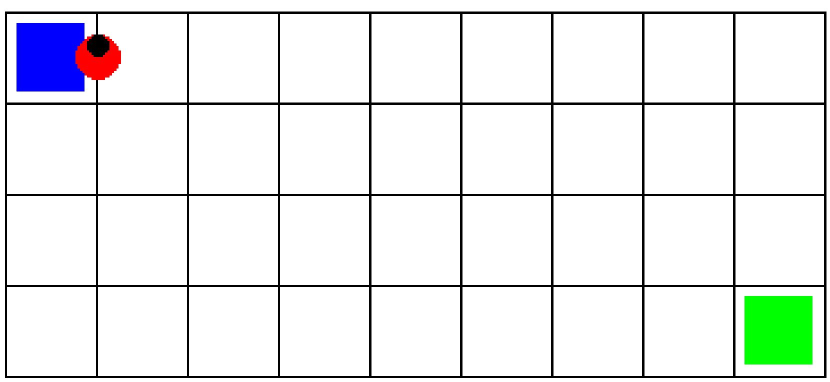


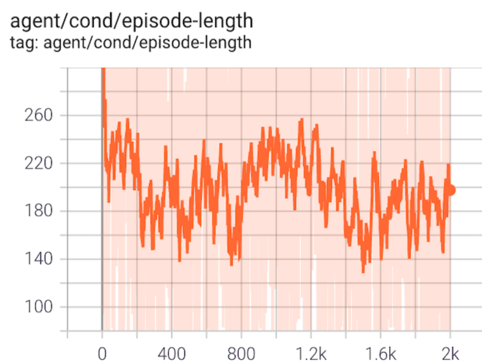
Figure 7: 4x9 Gym Sapientino map for the experiment with two colors.

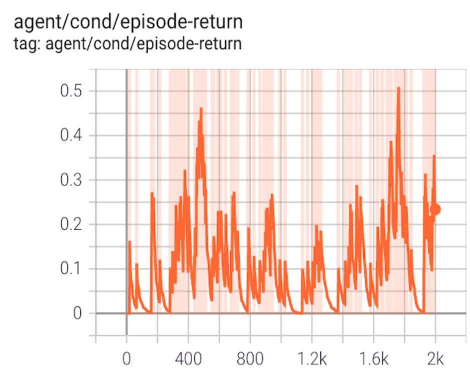
We present two progressive trials for this specific case, in particular in the second trial reaches properly the convergence.

Agent parameters								
Trial	Algorithm name	batch size	memory	exploration	lr	update fre-quency	reward shap-ing	episodes
trial1	PPO	32	20000	0.4	0.001	32	no	2000
trial2	PPO	64	64	0.0	0.001	20	yes	1000

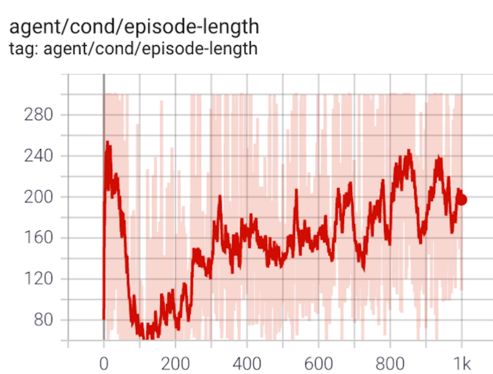
Table 1: Table containing the relevant hyperparameter configuration of the tested agents related to the two experiments with two colors

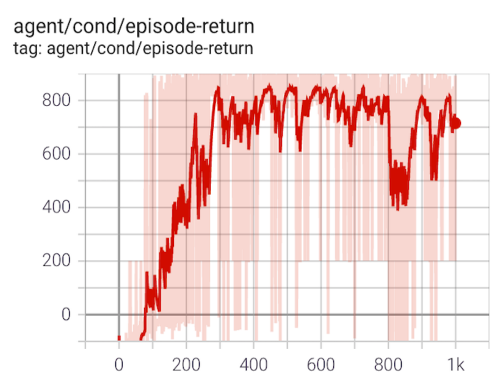
Plots related to the first trial:





Plots related to the second trial:





12 Experiments with two colors with bigger maps

In these experiments we have tried to train the agent with big complex maps in order to the reach the temporal goal *blue,green*.

12.1 First Trial

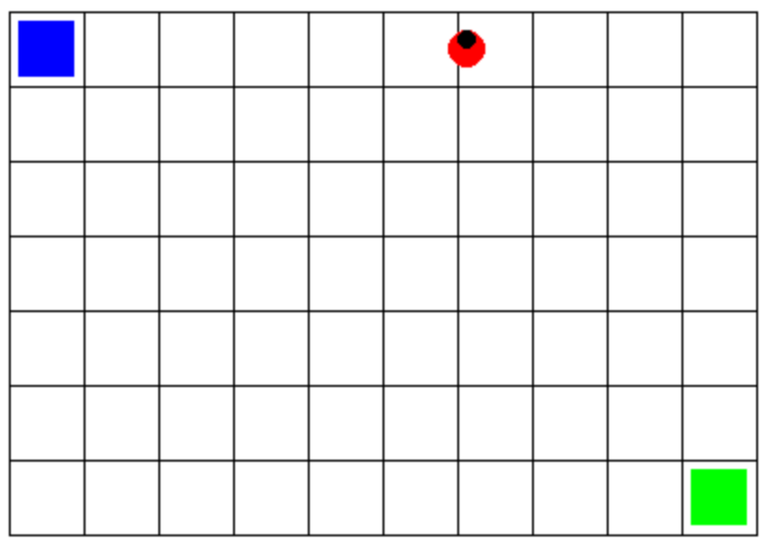
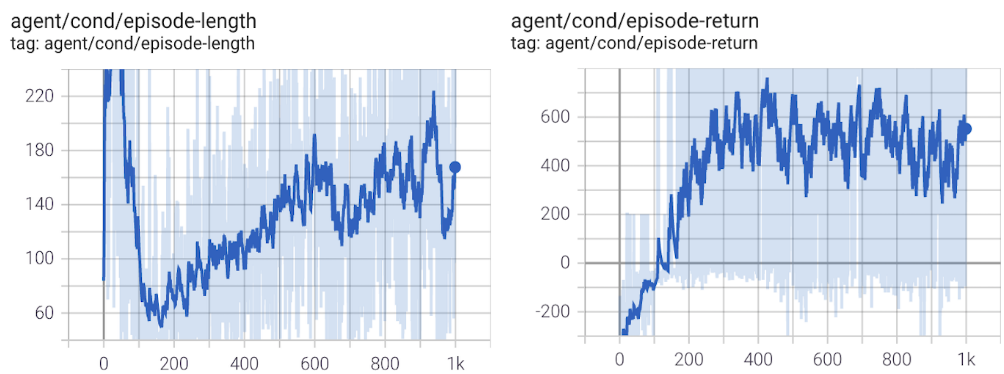


Figure 8: 7x10 Gym-Sapientino map



In this case the agent has a good behaviour, slightly worse than in the previous case with a smaller map. The agent reach the temporal goal quite frequently and thanks to the exploration parameter (set to 0.25) it can probe better the action space.

Agent parameters								
Trial	Algorithm name	batch size	memory	exploration	lr	update fre-quency	reward shap-ing	episodes
trial1	PPO	64	64	0.25	0.001	20	yes	1000

Table 2: Table containing the relevant hyperparameter configuration of the tested agents related to the experiment with two colors in the first bigger map

12.2 Second Trial

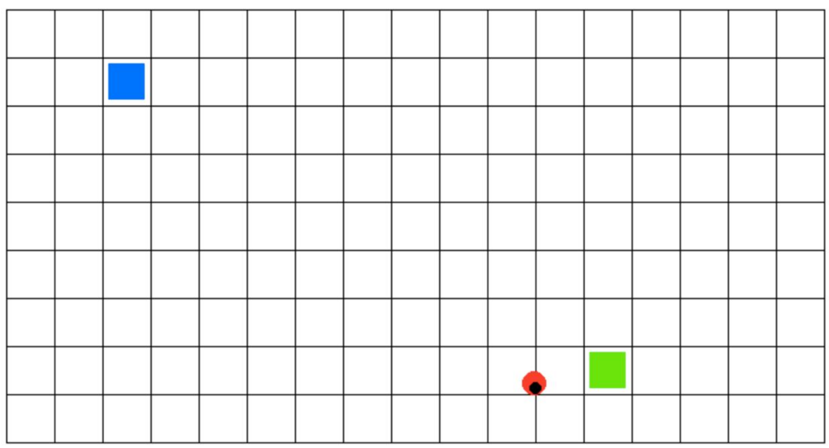
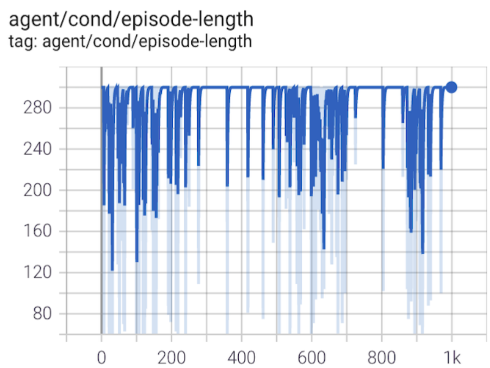
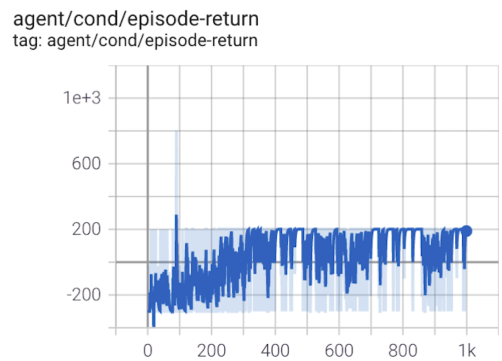


Figure 9: 9x17 Gym-Sapientino map

Plots related to this trial:





The rewards accumulated during the episodes are decidedly lower with respect to the previous experiments. In fact in this case the agent doesn't reach the goal frequently, the map is very large and despite the exploration addition, the agent's behaviour is decidedly worse.

Agent parameters								
Trial	Algorithm name	batch size	memory	exploration	lr	update frequency	reward shaping	episodes
trial1	PPO	64	64	0.3	0.001	20	yes	1000

Table 3: Table containing the relevant hyperparameter configuration of the tested agents related to the experiment with two colors in the second bigger map

13 Experiment with three colors

As last work step, we have tested maps with three colors, registering in the trials when agent reach the goal (characterized by the sequence of the three colors) and arriving at convergence situation.
The temporal goal for this experiment is: $\{$*blue*,*red*, *green*$\}$. Below we show the Gym-Sapientino map for the experiment (figure [10]).

In the optimal configuration, we expect the agent to execute a triangular trajectory reaching the blue tile first and then all the other colours.
For this specific temporal goal we have performed two experiments, the first one using our custom network explained in 8.1, the second one using the default tensorforce implementation.

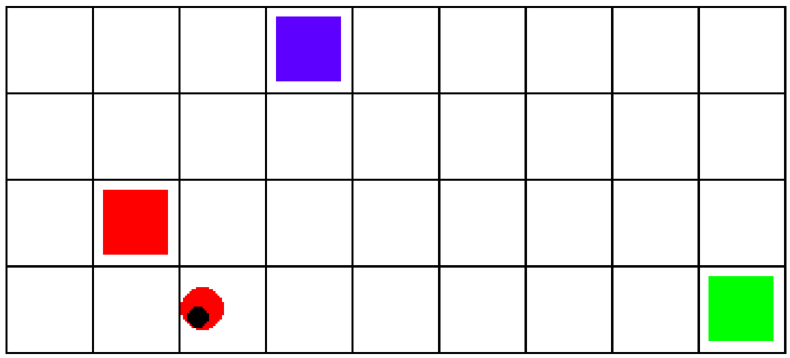


Figure 10: 4x9 Gym Sapientino map for the experiment with three colors.

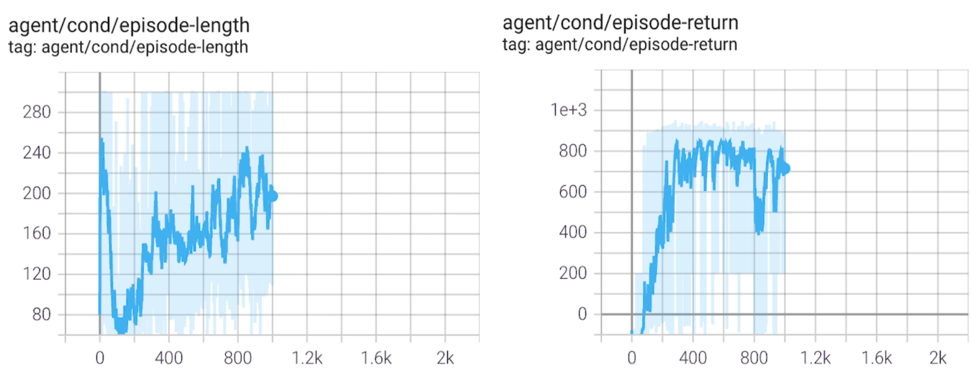


Figure 11: Plots related to the first trial with three colors and with our custom network.

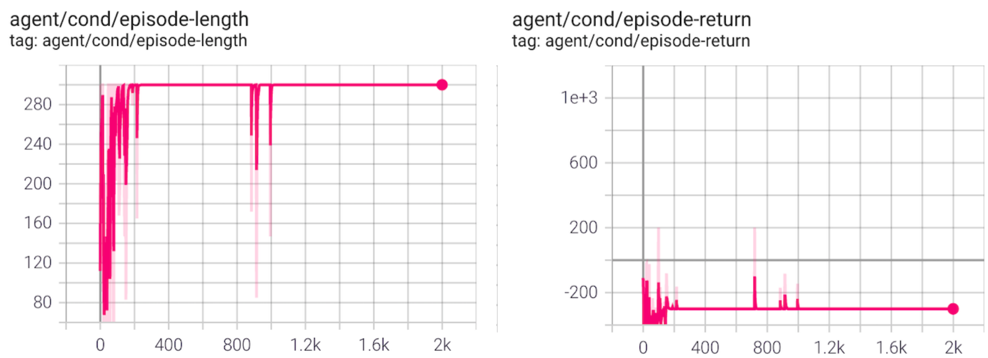


Figure 12: Plots related to the first trial with three colors and with auto network.

The auto network configuration used for the second trial (figure [12]) is automatically configured based on input types and shapes, and it is characterized by: one Retrieve layer that takes as input the state, two dense layers of size 64, one Register layer, an other Retrieve layer that takes as input the encoded state, two new dense layers of size 64, an other Register layer. Finally a Retrieve layer that concatenate the state and the encoded state tensors and a last dense layer of size 64.

It is possible to observe that in this case the agent doesn't reach the goal during the episodes, and then the rewards are always negative. So the behaviour of the non markovian agent in this case is decidedly worse w.r.t. the trial with our custom network.
In fact thanks to this custom network the agent can reach the temporal goal frequently arriving to convergence condition.

Agent parameters									
Trial	Alg. name	batch size	memory	exploration	lr	update frequency	reward shaping	episodes	network
trial1	PPO	64	64	0.0	0.001	20	yes	2000	custom
trial2	PPO	64	64	0.0	0.001	20	yes	2000	auto

Table 4: Table containing the relevant hyperparameter configuration of the tested agents related to the two experiments with three colors

14 Conclusion and result discussion

In our experiments we have find out that our agent suffers a major problem when it solving a problem in a large map. Due to the nature of the task, the markovian agents are likely to recieve rewards which are sparse and not seen very often. If it happens, it may happen that the agent converge sub-optimally which results in an agent policy that leads to negative reward action selection. To overcome this problem, we think the agent should be provided with a sufficient amount of exploration to better probe the action space

[4] .

In addition, we may shrink the map and create tasks which are easier the solve (the color tiles are nearer) so that the agents sample goal more frequently.

15 Tables and additional plots

Agent parameters				
Agent name or algorithm name	batch size	memory	exploration	lr
PPO	64	64	0.0	0.001

Table 5: Table containing the relevant hyperparameter configuration of the tested agents.

References

[1] G. De Giacomo and M. Favorito. Compositional approach to translate ltlf/ldlf into deterministic finite automata. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 122–130, 2021.

[2] A. D. Laud. *Theory and application of reward shaping in reinforcement learning.* University of Illinois at Urbana-Champaign, 2004.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[4] https://github.com/cipollone/gym-sapientino-case/tree/master/gym_sapientino_case

[4]Exploring the action space often leads to the execution of sub-optimal actions. However, if the agent is already behaving sub-optimally, an exploration phase may lead the agent to discover traces arrive closer to the goal.