Policy Networks for Non-Markovian Deep RL

- Reasoning Agents Project -

Appetito Daniele 1916560
Cognetta Salvatore 1874383
Rossetti Simone 1900592

September 2021

All the students contributed equally to the project.

Contents

1 Introduction

This project aims at solving navigation tasks with non-Markovian rewards (i.e. the *Sapientino Case* environment), developing a non-Markovian agent. However most of Reinforcement Learning (RL) theoretical frameworks expects the task to be modeled as a Markov Decision Process (MDP), meaning that state transitions are conditionally independent from history of states, for this reason RL agents can not directly solve such problems. In temporal goals, next state and reward do depend on sequence of states and actions; on this purpose we need to produce an extended MDP, combining RL agents with automata. In particular we use *Deterministic Finite Automata* (DFAs) transitions (Section 2) to train a set of experts, independent agents specialized in reaching their own subgoal, using an off-policy algorithm, derived from Q-learning (Section 4), named Deep Q-Networks (DQN) (Section 4.1). Additionally we exploited the knowledge of the DFA to speed up the learning process of the experts by an adaptation of the *Counterfactual Experiences for Reward Machines* algorithm **??** (Section 6). The relative implementation details are treated in Section 9.

2 Deterministic Finite Automata

A Deterministic Finite Automata (DFA in short) as stated by Hopcroft Hopcroft et al. (2006) is described as *a finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string.*

Formally a `DFA` is defined as a tuple

$$A = (\Sigma, S, s^0, \rho, F) \tag{1}$$

where:

- Σ is a finite non empty alphabet, containing $a_0, \ldots, a_k$ with $k = |\Sigma|$;
- S is a finite non empty set of states;
- $s^0 \in S$ is the initial state;
- $F \subseteq S$ is the set of accepting states;
- $\rho : S \times \Sigma \to S$ is a transition function.

The transition function takes as input a state and a symbol of the alphabet and returns a state s' where A can move into.

DFA can be seen as an edge-labeled directed graphs, where the nodes are the states of the automaton and the edges are the transitions from a state to

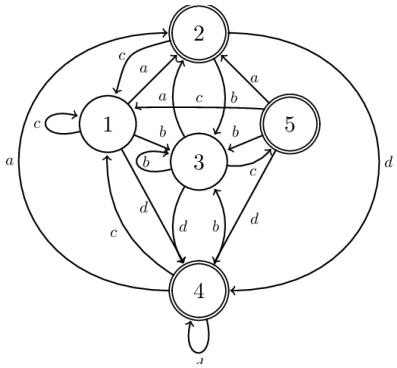


Figure 1: A representation of a DFA.

another. Each edge is labeled by a symbol inside the Σ alphabet. A graphical representation can be seen in figure (1).

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.

3 Markov Decision Process

Reinforcement learning is an area of machine learning which aims to learn an agent in order to maximize the total amount of reward, obtained form a complex, uncertain environment.

The agent must have a model of the actions that can be taken into the environment in which it acts, in order to be able to explore/interact with the environment itself. But the agent, during learning phase, is not told which actions to take, instead it must discover which actions yield the most reward by trying them. Moreover, actions taken might affect not only immediate reward, but also subsequent ones leading to delayed rewards. The agent in this way can learn its behaviour inside the specific environment incrementally, using its own experience of errors and successes. This is trial-and-error approach is widely discussed by Sutton and Barto in *Reinforcement Learning: An Introduction* Sutton and Barto (2018).
Typically the way in which the agent interact with the environment is modeled as shown in figure 2 also known as the Markovian stochastic control process, or Markov Decision Process (MDP).

Markov Decision Processes provide a mathematical model for decision making in a non-deterministic environment. A MDP can be defined as the tuple $\langle S, A, T, R, \gamma \rangle$, where S is the set of states A is the set of actions, T is a transition model, R is a reward function and γ as discount factor of the reward.

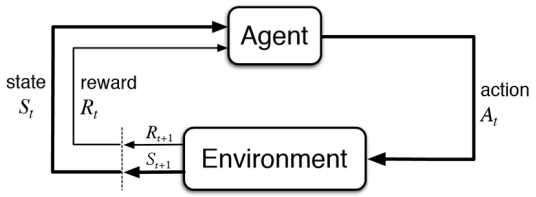


Figure 2: Reinforcement Learning: agent-environment interaction.

The transition model $T : S \times A \to P(S)$ is a function that given a tuple of state and action returns a distribution over the next state s':

$$T(s, a) = P(s', r, t|s, a) \tag{2}$$

where t is the terminal state, pointing that the agent reached a certain goal. The reward function $R : S \times A \to \mathbb{R}$, given an action a in state s returns a real valued reward of the action performed by the agent.

In MDPs the agent at time t interacts with the environment choosing an action $a_t \in A(s)$, obtaining a reward r_{t+1} and reaching a state s_{t+1}. These interactions are performed in discrete timesteps $t = 0, 1, 2, \ldots$ generating a *trajectory* $s_0, a_0, r_1, s_1, a_1, \ldots$ shown in figure (3).

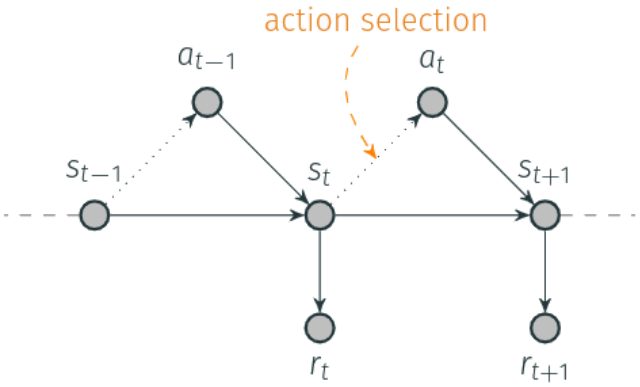


Figure 3: Markov trajectory.

In this model, given the present state s_t all the following states are independent of all past states, meaning that there are no dependencies between a state $s_t + 1$ and s_{t-i} with $i = 1, \ldots, k$ as shown in figure (3). This property is known as the **Markov assumption**, which states:

$$s_{t+1} \perp s_0, \ldots, s_{t-1}|s_t \quad \forall t \tag{3}$$

$$r_{t+1} \perp s_0, \ldots, s_{t-1} | s_t \quad \forall t \tag{4}$$

After describing the mathematical model, the next step is to choose a method for action selection. The way in which the agent selects actions, given a state, is due to a function called policy $\pi : S \to A$. Learning a policy that allows the agent to maximize the total amount of reward received is the goal of Reinforcement Learning.

To find an optimal policy π^*, we have to maximize an expected return, which consider all the reward, not only obtained with the current action, but also the ones that we expect to obtain in the future. This is the value function $V : S \to \mathbb{R}$:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s, \pi\right] \tag{5}$$

Therefore, the optimal expected return can be defined as:

$$V^*(s) = \max V^{\pi}(s) \tag{6}$$

Although state-values suffice to define optimality, it is useful to define action-values Q function $Q : S \times A \to \mathbb{R}$, which computes the expected return starting in s, taking an action a and then following the policy π:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s, a, \pi\right] \tag{7}$$

As for the value function, the optimal $Q - value$ function is defined as:

$$Q^*(s, a) = \max Q^{\pi}(s, a) \tag{8}$$

In summary, the knowledge of the optimal action-value function alone suffices to know how to act optimally.

4 Q-learning

Q-learning, proposed by Watkins and Dayan Watkins and Dayan (1992) in 1992 is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards without requiring adaptations.

For any finite Markov Decision Process (FMDP), Q-learning finds an optimal policy π in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state, $Q^{\pi}(s, a)$. Q-learning can identify an optimal action-selection policy for any given FMDP,

given infinite exploration time and a partly-random policy. The optimal Q-function $Q^*(s, a)$ is the maximum return that can be obtained starting from observation s, taking action a and following the optimal policy thereafter.

The algorithm, therefore, has a function that calculates the quality of a state–action combination:

$$Q : S \times A \to \mathbb{R} \tag{9}$$

Q is in general randomly initialized. During the execution of the algorithm the agent at time t, in a state s_t, selects an action a_t[1] and observes a reward r_t entering in a new state s_{t+1}.

The optimal Q-function obeys the Bellman optimality equation:

$$Q^*(s_t, a_t) = \mathbb{E}\big[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})\big] \tag{10}$$

The basic idea behind Q-Learning is to use the Bellman optimality equation as an iterative update to maximize the expected value:

$$Q(s_t, a_t)' \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})) \tag{11}$$

where α is the learning rate and γ is the discount factor. The new Q-table values $Q(s_t, a_t)'$ are given by the terms:

$(1 - \alpha)Q(s_t, a_t)$: the current value weighted by the learning rate. Values of the learning rate near to 1 make the changes in Q more rapid.

$\alpha \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$: the maximum reward that can be obtained from state s_{t+1}, weighted by learning rate and discount factor.

αr_t : the reward weighted by the learning rate.

The discount factor has the effect of valuing rewards received earlier higher than those received later.

4.1 Deep Q-Network

The DQN (Deep Q-Network) algorithm was developed by DeepMind Mnih et al. (2015) in 2015. It was able to solve a wide range of Atari games (some to super-human level) by combining reinforcement learning and deep neural networks at scale. The algorithm was developed by enhancing a classic RL algorithm called Q-learning with deep neural networks and a technique called *experience replay.*

[1] Q-learning is an off-policy algorithm that learns about the *greedy policy* $a_t = \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$ while using a different *behaviour policy* for acting in the environment/collecting data. This *behaviour policy* is usually an ε-greedy policy that selects the greedy action with probability $(1-\varepsilon)$ and a random action with probability ε to ensure good coverage of the state-action space.

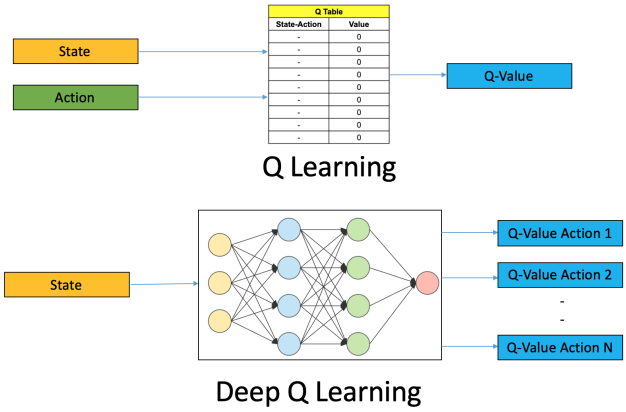


Figure 4: The conceptual difference of Q-learning and Deep Q-learning. Form the Q-table we get the Q-value from a (state,action) pair. In a DQN the neural network produces a Q-value distribution over the actions, its maximum value correspond to the best action to take.

"Q" refers to the function that the algorithm computes, the expected rewards for an action taken in a given state.

The core difference between Deep Q-learning and Vanilla Q-learning is the *agent brain*, Figure 4. Critically, Deep Q-learning replaces the regular Q-table with a neural network. Rather than mapping a state-action pair to a *q-value*, a neural network maps input *states* to (*action*, *q-value*) pairs in order to estimate the optimal Q-table. Such approach is particularly suited when dealing with problems in which it is impractical to represent the Q-function as a table containing values for each combination of states and actions, i.e. continuous space domains.

The DQN learns a function approximator (a neural network with parameters θ), to estimate the Q-values, i.e. $Q(s, a; \theta) \approx Q^*(s, a)$, by minimizing the temporal distance error using the loss at each step i:

$$L_i(\theta_i) = \mathbb{E}\big[(r_t + \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_{i-1}) - Q(s_t, a_t; \theta_i))^2\big] \tag{12}$$

which is minimized for $Q(s_t, a_t; \theta_i) \approx r_t + \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_{i-1})$ meaning that the estimated Q-values are as close as possible to the expected reward discount (this and the next reward), i.e. if this condition holds for every state (the loss is minimized) then we have the optimal Q-table. Minimization is performed by using stochastic gradient descent over mini-batches of *experience replay*.

4.1.1 Experience Replay

The DQN work by DeepMind introduced a technique called Experience Replay to make the network updates more stable. At each time step of data collection, the transitions are added to a circular buffer called the *replay buffer*. Then during training, instead of using just the latest transition to compute the loss and its gradient, they are computer by using a mini-batch of transitions sampled from the *replay buffer*. This has two advantages: better data efficiency by reusing each transition in many updates, and better stability using uncorrelated transitions in a batch.

5 Non Markovian Rewards Decision Processes

A Non Markovian Decision Process (NMDP) is, as the name suggests, a stochastic process that does not exhibit the Markov properties (Thiebaux et al. (2006)). The reward in this case is different in that it depends on the history of visited states, and its domain is a set of finite state sequences drawn from S, denoted as S^*.

A general function for a Non Markovian Reward can be stated as going from $(S \times A)^* \rightarrow \mathbb{R}$ [Camacho et al. (2017)]. Similar to the reward function, the policy for an NMRDP depends on history, and can be identified as a mapping from S^* to A. An NMRDP can be represented as a Tuple $\langle S, A, Tr, R \rangle$ where S, A, Tr as the same as in an MDP, and R is reinterpreted as described in the previous paragraph.

A main problem that occurs with NMRDP is that the reward may not be stationary and thus it (along with the optimal policy ρ^*) cannot be computed normally. To solve this we can define an MDP that is equivalent to the Non Markovian one (this can be seen in "The non markovian agent" section 8 later on).

5.1 LTLf/LDLf

To better express temporal properties over sequences of states we can use Linear Temporal Logic (LTL) [Brafman et al. (2018)] which is a language that allows us to encode automaton information into formulae φ that balance expressiveness with ease of use. We can use LTL_f (a variant of LTL expressed over finite states) to describe Markovian rewards, as well as construct a corresponding DFA (A_φ) that accepts a sequence of states π if and only if it satisfies φ (Camacho et al. (2017)).

In the case of non-Markovian rewards, we need to introduce a new formalism: LDL_f, *Linear Dynamic Logic over finite traces*. LDL_f is an extension of LTL_f,

including Regular Expressions in the temporal formulas, as we can see in the equations 13.

$$\begin{aligned} \varphi &::= tt|ff|\neg\varphi|\varphi_1 \wedge \varphi_2|\langle\varrho\rangle\varphi|[\varrho]\varphi \\ \varrho &::= \phi|\varphi?|\varrho_1 + \varrho_2|\varrho_1;\varrho_2|\varrho^* \end{aligned} \tag{13}$$

where tt and ff stand for `true` and `false`; ϕ is a propositional formula on current state; $\neg\varphi, \varphi_1 \wedge \varphi_2$ are boolean connectives; ϱ is a regular expression on propositional formulas ϕ with the addition of the test construct $\varphi?$; $\langle\varrho\rangle\varphi$ says that exists an execution of the Regular Expression (RE) ϱ that ends with φ holding; $[\varrho]\varphi$ states that all executions of RE ϱ along the trace end with φ holding.

It is more expressive than the Linear Time Logic, as it caputres Monadic Second-Order Logic (MSO) instead of First-Order Logic (FOL).

6 Counterfactual Experiences for Reward Machines (CRM)

Algorithm 2 Tabular q-learning with counterfactual experiences for RMs (CRM).

1: **Input:** S, A, $\gamma \in (0,1]$, $\alpha \in (0,1]$, $\epsilon \in (0,1]$, $\mathcal{P}$, L, U, u_0, F, δ_u, δ_r.
2: Initialize $\tilde{q}(s,u,a)$, for all $s \in S$, $u \in U$, and $a \in A$ arbitrarily
3: **for** $l \leftarrow 0$ **to** num_episodes **do**
4: Initialize $u \leftarrow u_0$ and $s \leftarrow$ EnvInitialState()
5: **while** s is not terminal **and** $u \notin F$ **do**
6: Choose action a from (s,u) using policy derived from $\tilde{q}$ (e.g., ϵ-greedy)
7: Take action a and observe the next state s'
8: Compute the reward $r \leftarrow \delta_r(u)(s,a,s')$ and next RM state $u' \leftarrow \delta_u(u, L(s,a,s'))$
9: Set experience $\leftarrow \{\langle s, \bar{u}, a, \delta_r(\bar{u})(s,a,s'), s', \delta_u(\bar{u}, L(s,a,s'))\rangle \mid \forall \bar{u} \in U\}$
10: **for** $\langle s, \bar{u}, a, \bar{r}, s', \bar{u}'\rangle \in$ experience **do**
11: **if** s' is terminal **or** $\bar{u}' \in F$ **then**
12: $\tilde{q}(s,\bar{u},a) \xleftarrow{\alpha} \bar{r}$
13: **else**
14: $\tilde{q}(s,\bar{u},a) \xleftarrow{\alpha} \bar{r} + \gamma \max_{a' \in A} \tilde{q}(s', \bar{u}', a')$
15: Update $s \leftarrow s'$ and $u \leftarrow u'$

Figure 5: Adaptation example of CRM method to tabular q-learning (off-policy learning) with the addition of lines 9 and 10.

Counterfactual Experiences for Reward Machines (CRM) is an algorithm from Icarte et al. (2020) which exploit the information in the reward machine (RM) to facilitate learning. It aims at learning policies $\pi(a|s,u)$ (with π the agent's policy, a the distribution over the actions, s the observation and u the

reward machine state) but uses counterfactual reasoning to generate *synthetic experiences*.

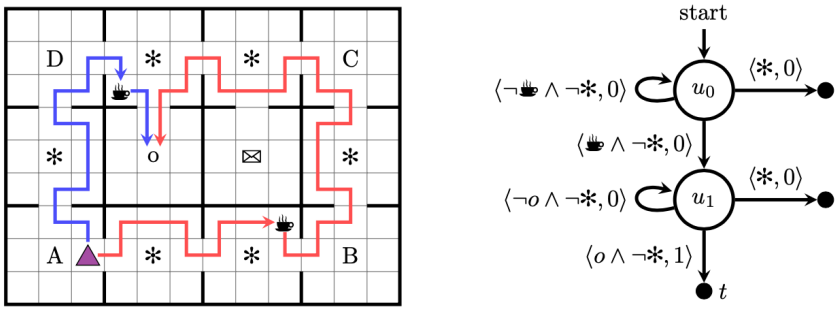
This algorithm is easily extendable to work with deep RL agents. In fact these experiences can then be used by an off-policy learning method, such as tabular Q-learning, DQN, or DDPG, to learn the policy $\pi(a|s,u)$ faster.

The key idea behind CRM is about enhancing the exploration of the reward machine state space, it allows the agent to reuse experience to learn the right behaviour at different RM states. Given the cross-state $\langle s,u\rangle$, the agent perform an action a and observe the new cross-state $\langle s',u'\rangle$ and get a reward r. We can exploit the reward machine to experience more outcomes of the cross-state and speed up the training. In fact after have performed the actual experience $\langle s,u,a,r,s',u'\rangle$, we can accumulate more experience collecting, for each RM state $\bar{u}\in U$, the resulting rewards $\bar{r}$ and outcomes $\bar{u}$ and generate the set of experiences:

$$\big\{\langle s,\bar{u},a,\bar{r},s',\bar{u}'\rangle \mid \forall \bar{u}\in U\big\}. \tag{14}$$

An example of possible implementation for tabular Q-learning is shown in Figure 5. For both DQN and DDPG, the counterfactual experiences would simply be added to the experience replay buffer and then used for learning as is typically done with these algorithms.

To better understand the content we can look at the example in Figure 6



(a) The office gridworld

(b) A simple reward machine

Figure 6: An example environment (a) and one reward machine for it (b) i.e. get coffee, then reach office.

Suppose that the agent gets to the office before getting the coffee. The Q-learning baseline would use that experience to learn that going to the office is not an effective way to get coffee. In contrast, Q-learning with CRM would also use that experience to learn how to get to the office. As such, CRM will already have made progress towards learning a policy that will finish the task as soon as it finds the coffee, since it will already have experience about how to get to the office.

CRM also converges to optimal policies when combined with q-learning and this is a theorem in Icarte et al. (2020):

Theorem. Given an MDPRM (Markov Decision Process with Reward Machine), CRM with tabular q-learning converges to an optimal policy for the problem in the limit (as long as every state-action pair is visited infinitely often).

While the convergence proof for CRM is directly given as a consequence of the convergence proof provided by Watkins and Dayan (1992) for tabular q-learning when we consider that the experience is sampled according to the its transition probability $p(\langle s', u' \rangle | \langle s, u \rangle, a) = p(s'|s, a)$.

7 Gym-Sapientino

This is a new RL discrete environment that respects the gym interface[2]. In this environment, we have a planar unicycle robot that can move inside a 2-D grid, which is basically a rectangular map with some coloured cells, as depicted in Figure 7. The goal of the agent-robot is to visit the coloured cells of a continuous state space in a specific order. SapientinoCase has a low–dimensional observation space, which allows to use simple Feed–Forward Neural Networks.

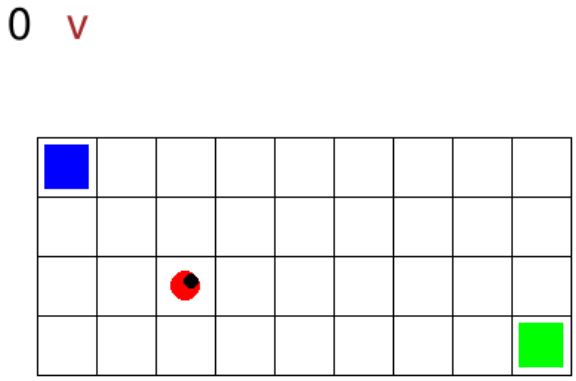


Figure 7: Gym-Sapientino environment.

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*.
After an action executed by the agent executes an action, the environment re-

[2]https://github.com/cipollone/gym-sapientino-case

turns an associated reward and an observation on the state and on automaton state.

7.1 Temporal Goal on Gym-Sapientino

The non markovian agent moves in the map executing a non markovian task: visit a sequence of cells identified by colors, in a specified order. Essentially the agent should be prone to reach some temporal goals, identified by the sequence of colors like.
When the agent reach a temporal goal it receives a specific reward from the environment, moreover following the temporal goals it will accumulate an amount of rewards. Therefore more frequently the agent earns rewards more it will be led to learn.
This environment allows to work with four different colors (and consequently with a temporal goal characterized by four components): **yellow**, **blue**, **green** and **red**.
An example of temporal reward inside the *Gym-Sapientino* environment is LDL_f formula: `<!red*; red; !green*; green; !blue*; blue>end`.
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. In the previous version of *Gym-Sapientino* there was a particular failure state, denoted as `SINK`. In the new version of this environment this variable is removed and therefore the dfa is sequential, therefore if we consider a map characterized by two colors, for instance *blue* and *green*, the related automaton will no longer have a frozen set as shown by figures (8) and (9).

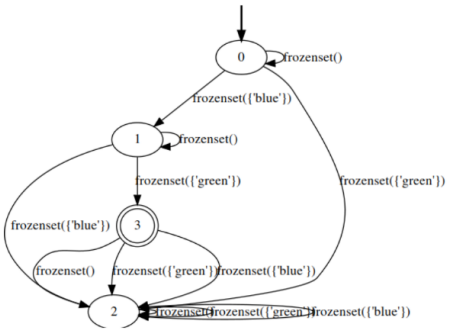


Figure 8: Old DFA with sink state.

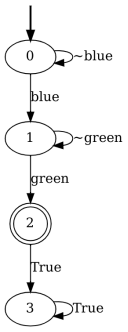


Figure 9: New DFA without sink state.

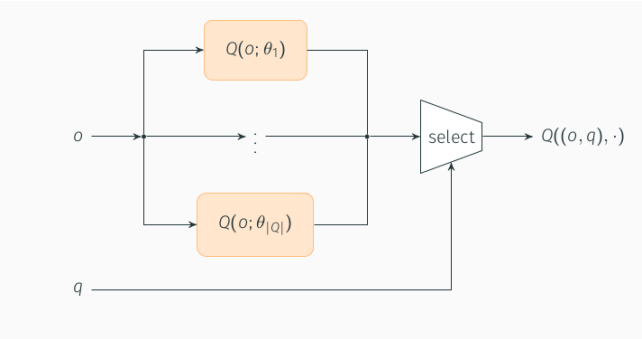


Figure 10: Baseline implementation of a non markovian policy network. It takes as input the observation o and the state of the goal DFA q. Then, according to the state of the automaton, the agents selects one of the $|Q|$ *separate experts* for the action selection. This image is taken from the slides of PhD student Roberto Cipollone for the Reasoning Agent course held in Sapienza in 2020/2021.

Notice that `0` is the initial state and `2` is the accepting state. In this case the agent can reach the temporal goal visiting the following sequence: `0,1,2`.

8 The non markovian agent

Our main focus is to have an agent go through a specific state sequence such that it passes through the blue square first, then the red square, and finally end on the green square. Thus meaning that to get to the goal state in sequence we have to have passed through all the previous ones in order. This sequence of events therefore requires the agent to know where it has been in the past, making it a non markovian problem, and, in turn, a non markovian agent. However, as stated in the NMDRP section before, it is much easier to compute a system that behaves in a Markovian way.

To allow us to create a Markovian problem from our task we split it into smaller parts. Hence the new problem becomes a combination of smaller sub-tasks, instead of having the agent follow the full sequence, we will have it reach the individual states. The first sub-task will require the agent to reach the blue square, the second the red, and the third the green. By doing this we remove the dependency on history, as each problem is treated individually, and we create three separate Markovian problems out of a non Markovian one.

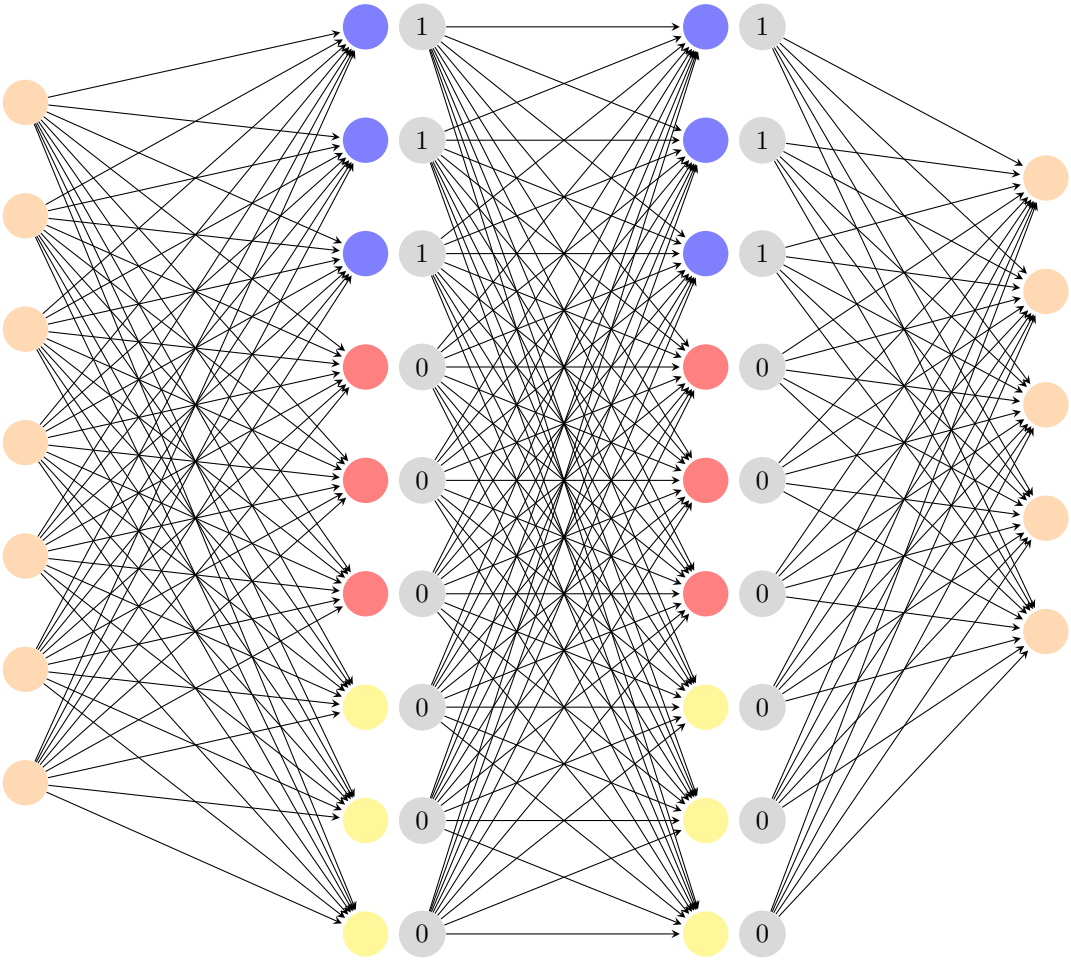


Figure 11: Conceptual representation of the non markovian policy network used, a 2 hidden layer Multi Layer Perceptron (MLP). Notice that the hidden layers activates depending on the one hot encoded automata state in input, this makes each set of perceptrons specialized in solving one automata state. The architecture in the figure may employed in solving the problem of visiting a sequence of three colors amid blue , red and yellow. Color repetition is also applicable, the more the number of automata states the larger the hidden layers.

9 Implementation Details

The agent and its policy are implemented in Tensorforce[3], an open source library for deep reinforcement learning based on Tensorflow and Keras. Tensorforce supports a variety of deep RL agents on-policy and off-policy like DQN, PPO, DDPG, etc.
Tensorforce is one of the most frequently used frameworks for reinforcement learning experiments. However, as far as we know, the latest version of the library (Tensorforce 0.6.5) does not support non markovian agents implementation 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 fact it permits to internally develop custom networks; it offers the implementation of most common neural network layers (dense, recurrent, convolutional, attention based) for policy network implementation.

9.1 Network

In this work, we propose a non markovian agent implementation inspired from the work in Figure 10 about the *separate experts* and from[4] which employs a single policy network for action selection divided in sets of neurons each defining a single markovian agent, see Figure 11. We make each chunk of the same size and associate it to a different color in the goal sequence.

The network itself is very simple, a 2 hidden layers Multi Layer Perceptron (MLP). We have two inputs, the first is a 7-dimensional real input space (state space), while the second is a $num\_automata \times hidden\_size$-dimensional binary input, and a 5-dimensional real output space (action space). The binary input is created by one hot encoding of the automata state (using *ones* for the current automata state correspondent neurons and *zeros* otherwise), this is needed to gather the hidden neurons. In this way we are able to select the neurons, thus the current expert, which will contribute to the action selection.

We have implemented the custom network in the following way (figure [11]):

```
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),

```

[3] https://tensorforce.readthedocs.io/en/latest/
[4] https://github.com/francycar/RA_project/tree/main

```
    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**: permits to retrieve tensors by name, such as layers outputs, and to aggregate them 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**: used jointly with Retrieve layers permits naming the tensors.
- **Dense layers**: MLPs.

9.2 Reward Shaping

In order to speed up training we decided to input multiple rewards, one for each automata state. *Reward Shaping* is a common technique in RL, it aims at simplify the task to learn. This is particularly suited for the *separate experts* formulation since it permits to fasten the learning process of each agent. In fact in this way each agent independently learns to reach its subgoal. For instance if we reach the second automata state on a sequence of three, in case of failure (of goal reaching) the entire episode is not to be considered pointless, since two experts over three have received their reward and learnt something more. 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 which implies that the this method is a also very time consuming.

In our code the *Reward Shaping* has been implemented and applied by an *if-else* cascade, where according to automata state transition the agent receive

a great reward of 500 when reaching the right state, 1000 for goal reaching and -1 for each step.
In our particular case, reward shaping is very effective since it permits to supervision each agent learning independently. In practice each agent incrementally learn and improve its knowledge about subgoal reaching, while attending to goal reaching in a *collaborative* way.

9.3 Counterfactual Experiences for Reward Machines

The *Tensorforce* framework does not permit to reproduce natively the CRM algorithm. In fact it makes available to the user only two possibles interfaces to perform training:

act-observe this interface permits to observes reward at each timestep using the method `observe()`, it must be preceded by the method `act()`, which allows to store in memory the inputs/outputs/internals and use them in updates of the training process once the terminal state is reached.

act-experience-update this second interface instead permits to collect inputs/outputs/internals in independent mode, without updating the policy in the current episode. Once the terminal state is reached the method `experience()` permits to feed the experience traces. While the method `update()` permits to update the policy.

These two interfaces can not be customized and neither combined since it is mandatory that the `update()` method is called on the complete sequence, i.e. the last trace must be terminal.
Furthermore, since in the *Gym Sapientino Case* environment manages the DFA states for us, i.e. observations and automata states are packed together, this does not allow the user to *observe* the next automata state without actually performing an action, and thus modifying the environment state and timestamp. This is particularly critical for the *Counterfactual Experiences for Reward Machines* algorithm which instead requires (see Equation 14) the computation of the automata reward in the current environment configuration, i.e. $\forall \bar{u} \in U$ $\langle s, a, s' \rangle$ must not change, while $\langle \bar{u}, \bar{r}, \bar{u}' \rangle$ should be observed.
With the described setting, the problem is intractable, moreover, since for the Deep Q-Networks case, the CRM algorithm leads to adding the *counterfactual experiences* to the *replay buffer* (Section 4.1.1) as explained in Section 6, we decided to extend the *Gym Sapientino Case* environment in order to produce synthetic experiences.
Gym Sapientino Case is built upon a stack of wrappers, namely the `gym.Wrapper` class; in turn it is included into the `TemporalGoalWrapper` which manages the automata formulas and states. Thus we decided to extends the environment by

including an independent `TemporalGoalWrapper`, namely the `TemporalGoalWrapperSynthetic`, defined in the `temprl` package, specially modified creating a fork[5] of the original repository[6].
The synthetic wrapper simulate the behaviour of the automata but this time without afflicting the environment state. In order to do so, in the `step` function of the synthetic wrapper the next state returned is not chosen accordingly to the temporal goals, instead is chosen randomly from the set of possible states in which the automata can go. The list of possible next state of the automata is taken from the trajectories of the DFA itself, thanks to the built-in function `get_transitions()`.
At training time we use the *act-experience-update* interface. We collect the tuples $\langle s, a, s' \rangle$ until the terminal state is reached without performing any update. In a second loop, for each state collected, we produce synthetic automata trasitions using the method described above generating the set described in Equation 14. The resulting synthetic experience is then concatenated to the real traces. At the end the experience is fed into the buffer replay by calling `experience()` and the policy is updated. We do not produce synthetic copies for the terminal state which otherwise would produce a RunTimeError when calling the method `update()` (the reason is explained above).

10 Experiments

In this section we are going to discuss some of the results obtained, namely we focused on comparing the learning efficiency with respect to the increasing of the number of temporal goals. In fact, according to the different kind of *Gym Sapientino* maps we are going to consider, the performances of the algorithm largely varies.
Furthermore we will evaluate the feasibility of correctly implementing the *Counterfactual Experiences for Reward Machines* algorithm, described in Section 6, in the joint *Tensorforce* framework and *Gym* environment, which is natively a closed system and do not permits to the user to directly implement the learning-by-experience in a temporal goals domain. All the experiments we are going to discuss are run on our local machines with no GPU acceleration enabled. Different from (un)supervised deep learning, RL does not always benefit from running on a GPU, depending on environment and agent configuration. In particular for RL-typical environments with low-dimensional state spaces (i.e., no images), one usually gets better performance by running on CPU only. In fact *Tensorforce* is configured to run on CPU by default.

The default setting for our *Sapientino Case* experiments are as follows:

[5] https://github.com/SalvatoreCognetta/temprl/tree/develop
[6] https://github.com/whitemech/temprl/tree/develop

Gym Sapientino Case settings	
initial_position	(2, 2)
angular_speed	30.0
acceleration	0.10
max_velocity	0.40
min_velocity	0.0
reward_outside_grid	0.0
reward_duplicate_beep	0.0
reward_per_step	-0.1

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

Of course to understand the ideal parameters we had to perform progressive trials and constantly examine the training process on *Tensorboard*, in particular the two measurements:

episode-length this measure the quantity of timestamps required to reach the goal. This is a measurement of the optimization of the time. In fact for shorter episode length the agent is learning to optimize the moves to immediately reach the goal. Furthermore since in this environment we do not have the failure state, this score is particularly indicative of the progression of the learning, i.e. small episode length means a successful terminal state.

episode-return indicate the discounted reward of the episode, higher values of course means better learning. This measurement is instead a limited performance index since it do not inform about strategy optimization.

10.1 One Color Easy

In this first trial we simply experimented our implementation of the Deep Q-Network using *Counterfactual Experiences for Reward Machines* algorithm (Section 6). The map is relatively simple (Figure12). In fact the algorithm seems to converge pretty fast (Table 13). The parameters used for the training are shown in Table 2.

Agent hyperparameters				
Map	Algorithm	batch size	memory	exploration
easy	DQN	500	10000	0.5
easy	DQN_{CRM}	500	10000	0.5

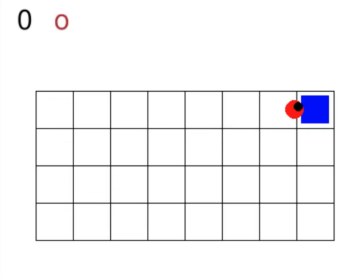
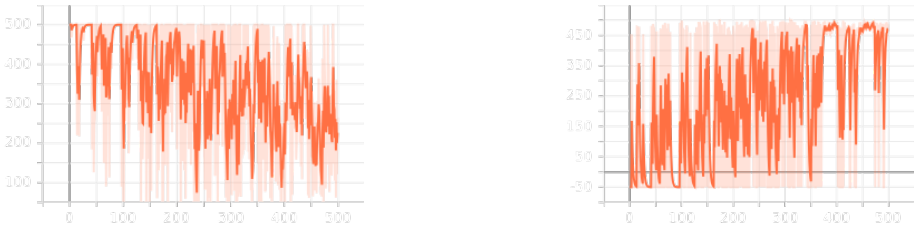


Figure 12: 4x8 Gym Sapientino map for the experiment with one color. The initial position is in the cell [2,2].

Agent hyperparameters (contd.)			
lr	update frequency	episodes	timestamps
0.001	500	1000	500
0.001	500	1000	500

Table 2: Table containing the relevant hyperparameter configuration of the tested agents related to the experiment with 1 color



(a) epoch-length

(b) epoch-reward

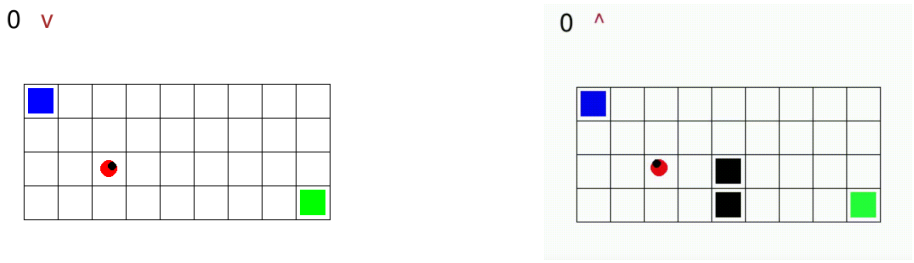
Figure 13: The training plots for DQN with parameters in Table 2, in Sapientino Map with one colour in Figure 12. The convergence rate increases as the epoch length decreases.

As shown in Table 13 the DQN_{CRM} algorithm converges for the simple case of a single automata state. In the following experiments we performed the same experiment for a larger number of states.

10.2 Two Colors Easy and Hard

In this experiment we reproduced the same training but using two different maps to see how the presence of obstacles affect the training process. Figure 14b shows the *hard* case used in which there is an obstacle between the two temporal goals.

We can notice from plots in Figure 15 that the agent struggle at finding the first temporal goal, this is because the goal is particularly far from the initial position, the agent need to explore for several episodes before start getting some rewards.



(a) Easy map with two colors

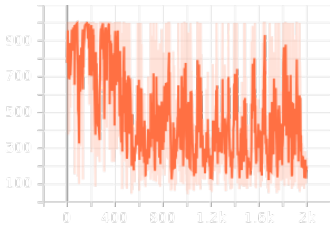
(b) Hard map with two colors

Figure 14: 4x9 Gym Sapientino maps for the experiment with two color. The initial position is in the cell [2,2]. (b) shows the *hard* case used in which there is an obstacle between the two temporal goals.

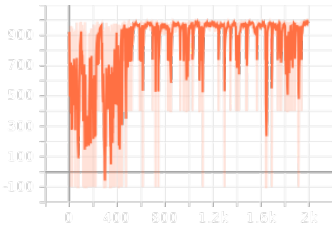
(a) epoch-length

(b) epoch-reward

Figure 15: The training plots for DQN with parameters in Table 3, in an easy Sapientino Map with two colours in Figure 14a. The convergence rate increases as the epoch length decreases.



(a) epoch-length



(b) epoch-reward

Figure 16: The training plots for DQN with parameters in Table 3, in an *hard* Sapientino Map with two colours in Figure 14b. Unexpectedly, using the parameters in Table 3, the training converges faster than the *easy* case.

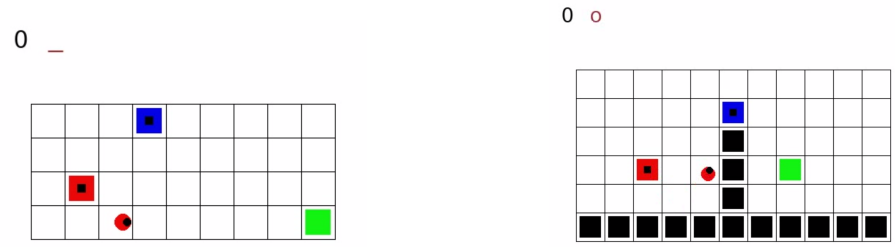
Agent hyperparameters				
Map	Algorithm	batch size	memory	exploration
easy	DQN	256	10000	0.4
hard	DQN	500	32500	0.5

Agent hyperparameters (contd.)			
lr	update frequency	episodes	timestamps
0.001	256	1000	500
0.001	500	2000	1000

Table 3: Table containing the relevant hyperparameter configuration of the tested agents related to the experiment with 2 color

10.3 Three Colors Easy and Hard

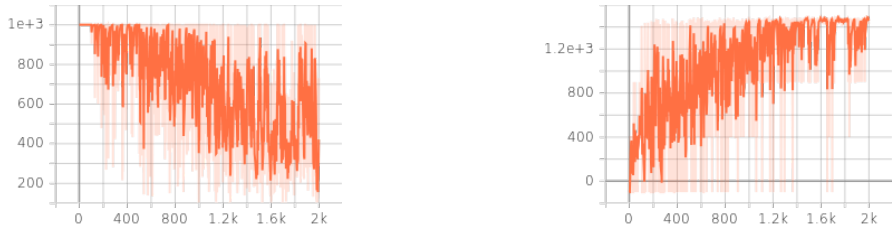
In this experiment we tested again the convergence of the algorithm on harder maps with bigger obstacles. The Figure 17b shows a very hard map 6x11 in which there is a very large obstacle which separate two near temporal goals, in particular here the goal sequence is defined as $\{blue, red, green\}$, which makes the problem really interesting since the agent has to perform very difficult maneuvers to reach each subgoal.
Looking at the plots in Figures 18 and 19, it is easy to recognize how the presence of obstacles delays the learning and slow down the convergence.



(a) Easy map with three colors

(b) Hard map with three colors

Figure 17: (a) 4x9 Gym Sapientino map for the easy experiment with three color. (b) 6x11 Gym Sapientino map shows the *harder* map used in which there is an obstacle between the two temporal goals. The initial position is in the cell [2,2] for both experiments.



(a) epoch-length

(b) epoch-reward

Figure 18: The training plots for DQN with parameters in Table 4, in an *easy* Sapientino Map with three colours in Figure 17a.

(a) epoch-length

(b) epoch-reward

Figure 19: The training plots for DQN with parameters in Table 4, in an *hard* Sapientino Map with three colours in Figure 17b. The presence of obstacles delays the learning process with respect to the *easy* case.

Agent hyperparameters				
Map	Algorithm	batch size	memory	exploration
easy	DQN	32	10000	0.4
hard	DQN	500	32500	0.6

Agent hyperparameters (contd.)			
lr	update frequency	episodes	timestamps
0.001	32	2000	1000
0.001	500	2000	1000

Table 4: Table containing the relevant hyperparameter configuration of the tested agents related to the experiment with 3 color

10.4 Four Colors Easy

In this experiment we used the color sequence {*blue*,*red*,*yellow*,*green*}, Figure 20. This sequence required a longer training with respect to the other experiments. In fact we can say the training time required is directly proportional to the number of colors, thus experts.
We performed this experiment using the DQN algorithm without CRM, the parameters used are shown in Table 5. From the plots in Figure 21 we can

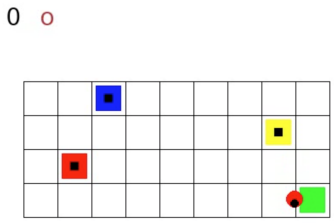
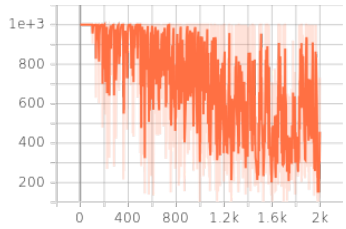
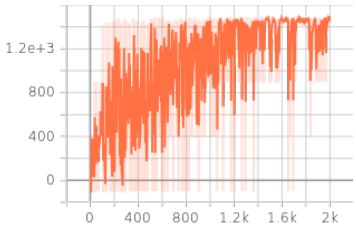


Figure 20: The Sapientino Map we used for the experiment with 4 colours. The temporal goal sequence is {*blue*,*red*,*yellow*,*green*}.

understand how the agent learns to converge faster as the episodes increase. Anyway convergence is reached at half epochs, but the epochs left are needed to optimize the time for convergence.



(a) epoch-length



(b) epoch-reward

Figure 21: The training plots for DQN with parameters in Table 5, in Sapientino Map with four colours in Figure 20. The convergence rate increases as the epoch length decreases.

Agent hyperparameters				
Map	Algorithm	batch size	memory	exploration
4 easy	DQN	500	32500	0.5

Agent hyperparameters (contd.)			
lr	update frequency	episodes	timestamps
0.001	500	2000	1000

Table 5: Table containing the relevant hyperparameter configuration of the tested agents related to the experiment with 4 colours.

11 Conclusion and result discussion

In this paper we have proposed a Tensorforce based non markovian agent implementation which is able to solve a non markovian navigation task in *Gym Sapientino*. We proved the effectiveness of the method, showing that it is possible to solve temporal goals in this domain by using as many specialized experts in the hidden layers as the automata states.
We used *Reward Shaping* which has been shown from[7] to have many benefits, they also shown that the sparse reward problem, which was one of the major issue our agent experienced in the training process, can be easily overcome by introducing intermediate positive rewards which boost the learning process and encourages the agent to complete the various steps that are needed to reach the goal in a supervised manner.

[7] `https://github.com/francycar/RA_project`

We realized an implementation of the *Counterfactual Experiences for Reward Machines* algorithm (Section 6) in *Tensorforce* framework by customizing the *Sapientino Case* environment to separate the *Deterministic Finite Automata* (Section 2) from the `gym.Environment` in order to cycle over the automata states as described in Equation 14. Moreover we conduct several experiments to demonstrate the success of the implementation and of the CRM algorithm itself. In particular we noticed, but it is also intuitive a priori, that the CRM algorithm is particularly useful when dealing with larger automata state spaces. On the counter part it is particularly time consuming when implemented to run locally on the CPU, especially for larger state spaces.

References

R. Brafman, G. De Giacomo, and F. Patrizi. Ltlf/ldlf non-markovian rewards. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

A. Camacho, O. Chen, S. Sanner, and S. A. McIlraith. Decision-making with non-markovian rewards: From ltl to automata-based reward shaping. 2017. URL `http://www.cs.toronto.edu/~sheila/publications/cam-etal-rldm17.pdf`.

J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 0321455363.

R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning, 2020.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

S. Thiebaux, C. Gretton, J. Slaney, D. Price, and F. Kabanza. Decision-theoretic planning with non-markovian rewards. *J. Artif. Intell. Res. (JAIR)*, 25:17–74, 01 2006. doi: 10.1613/jair.1676.

C. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8: 279–292, 05 1992. doi: 10.1007/BF00992698.