SoftwareX 35 (2026) 102889

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

PyCRM: A Python library for reward machine-based reinforcement learning

Tristan Bester [a,*], Geraud Nangue Tasse [a,b], Benjamin Rosman [a,b], Steven James [a,b]

[a] University of the Witwatersrand, Johannesburg, Gauteng, South Africa
[b] Machine Intelligence and Neural Discovery (MIND) Institute, Johannesburg, Gauteng, South Africa

ARTICLE INFO

Keywords:
Reinforcement learning
Reward machines
Counterfactual learning
Non-markovian decision processes

ABSTRACT

Reinforcement Learning (RL) research often models environments as Markov decision processes. Yet many real-world tasks are non-Markovian, requiring memory of past events. A common approach encodes these history-dependent rewards as finite automata, but these are difficult to integrate into standard RL pipelines. We present PyCRM, a Python framework for non-Markovian RL built on two automata classes: Reward Machines and their Turing-complete generalisation, Counting Reward Machines. PyCRM supports automatic environment construction and integrates seamlessly with popular RL libraries. This lowers the barrier to entry, allowing researchers to focus on task design and theory while exploring more complex, temporally extended problems.

Metadata

Code metadata (mandatory).

Nr.	Code metadata description	Metadata
C1	Current code version	v1.1.0
C2	Permanent link to code/repository used for this code version	https://github.com/TristanBester/pycrm
C3	Permanent link to Reproducible Capsule	N/A
C4	Legal Code License	MIT License
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Python
C7	Compilation requirements, operating environments & dependencies	gymnasium, numpy, stable-baselines3
C8	If available Link to developer documentation/manual	Documentation: https://pycrm.xyz
C9	Support email for questions	tristanbester@gmail.com

1. Motivation and significance

Recent advancements in Reinforcement Learning (RL) have enabled agents to solve increasingly complex tasks, but most RL methods rely on the Markov assumption, where decisions depend only on the current state [1]. This assumption limits their applicability in many real-world scenarios, such as robotic manipulation or strategic planning, where optimal behaviour depends on the sequence of past events [2–4]. These non-Markovian problems require more expressive models capable of reasoning over history. As Markovian tasks become more tractable, the field is shifting towards temporally extended problems, with non-Markovian reward modelling emerging as a key approach [5–7]. Defining rewards over sequences of subgoals rather than individual states enables agents to tackle richer, more realistic tasks.

Reward Machines (RMs) offer a structured and expressive framework for specifying non-Markovian reward functions [8]. By representing tasks as finite-state machines, RMs track progress and assign rewards based on the satisfaction of high-level temporal conditions, naturally

* Corresponding author.
Email address: tristanbester@gmail.com (T. Bester).

https://doi.org/10.1016/j.softx.2026.102889
Received 15 December 2025; Received in revised form 22 March 2026; Accepted 12 July 2026
Available online 22 July 2026

supporting constructs like loops and conditionals. Counting Reward Machines (CRMs) extend this paradigm by integrating counters, achieving Turing-complete expressiveness and allowing the specification of any reward structure definable by an algorithm [9,10]. Both RMs and CRMs enhance sample efficiency by enabling agents to reason over sub-tasks in parallel, leveraging techniques such as counterfactual experience generation. A formal introduction to these concepts and the notation used throughout this paper is provided in Appendix A.

Despite their theoretical appeal, the practical adoption of RMs and CRMs has been hindered by significant implementation complexity. Integrating these automata into RL pipelines requires extensive modifications to the environment, state tracking, observation spaces, and learning algorithms, creating a substantial barrier to entry. As a result, non-Markovian tasks are often implemented using ad-hoc reward functions that are brittle, difficult to debug, and fail to leverage the benefits of structured representations [11]. Existing public implementations are typically designed for reproducibility rather than general use, limiting their flexibility and customizability [8–10,12–15]. To overcome these challenges, we introduce a unified software framework that makes RMs and CRMs accessible and practical, allowing researchers to utilise their advantages without prohibitive implementation costs.

To address these challenges, we present *PyCRM*, a Python software framework designed to facilitate efficient research and development in non-Markovian RL. Our framework overcomes the critical obstacles associated with non-Markovian reward specification by offering a modular and rigorously engineered system. It enables users to express temporally extended tasks declaratively and compiles these specifications automatically into Gymnasium-compatible environments [16], thereby removing significant engineering overhead. This design facilitates straightforward integration with existing RL pipelines such as Stable Baselines 3 (SB3) [17], allowing users to benefit from well-tested, sample-efficient algorithms without requiring modifications to their learning core. Specifically, PyCRM makes the following key contributions:

1. A high-level abstraction layer that cleanly separates reward specification from RL implementation, substantially lowering the entry barrier for non-Markovian RL research.
2. Integrated support for counterfactual experience generation in off-policy learning, enabling users to exploit the sample efficiency advantages of RMs/CRMs without manually defining alternative experience trajectories [8,9].
3. Seamless interoperability with SB3, including extended implementations of *all* off-policy algorithms with native support for counterfactual-enhanced learning.
4. Comprehensive documentation and worked examples spanning tabular domains to deep RL in both discrete and continuous action spaces, thereby bridging the gap between theoretical advances and practical deployment.[1]

2. Software description

PyCRM is a Python framework that simplifies solving complex, non-Markovian RL problems using RMs and CRMs. By offering a high-level interface and seamless integration with popular libraries like Gymnasium and SB3, it lowers the implementation barrier, enabling researchers to focus on task design rather than low-level coding.

2.1. Software architecture

PyCRM's modular architecture augments the standard agent–environment loop by compiling three user-defined components—a *ground environment*, a *labelling function*, and a *reward machine* (RM or CRM)—into a single, augmented Gymnasium environment (Fig. 2). The user first defines a labelling function, $\mathcal{L}$, which acts as a perception layer that maps low-level environment transitions $\langle s, a, s' \rangle$ to a set of high-level propositional events (Fig. 1). The RM or CRM then defines the task logic, prescribing how sequences of these events translate into rewards.

The PyCRM compiler automatically wraps these components, transparently handling the evaluation of the labelling function and the execution of the RM/CRM logic. To maintain the Markov property for the learning agent, the wrapper augments the observation space with the current machine state. The resulting environment is compatible with any standard RL algorithm and exposes methods for generating counterfactual experiences to accelerate learning. Crucially, PyCRM imposes

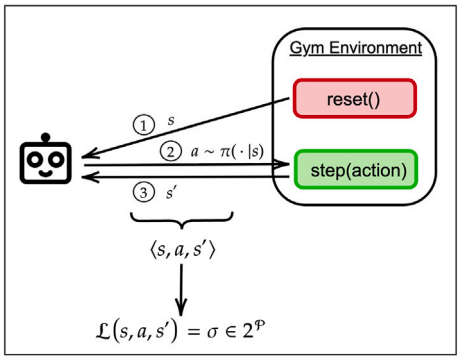

Fig. 1. The standard agent–environment interaction loop in Gymnasium. (1) the environment is reset, yielding an initial state s. (2) the agent selects an action a, which results in a new state s' (3). The labelling function $\mathcal{L}$ then maps each low-level transition $\langle s, a, s' \rangle$ to a set of high-level propositional symbols $\sigma \in 2^{\mathcal{P}}$, providing a compact description of the events that occurred.

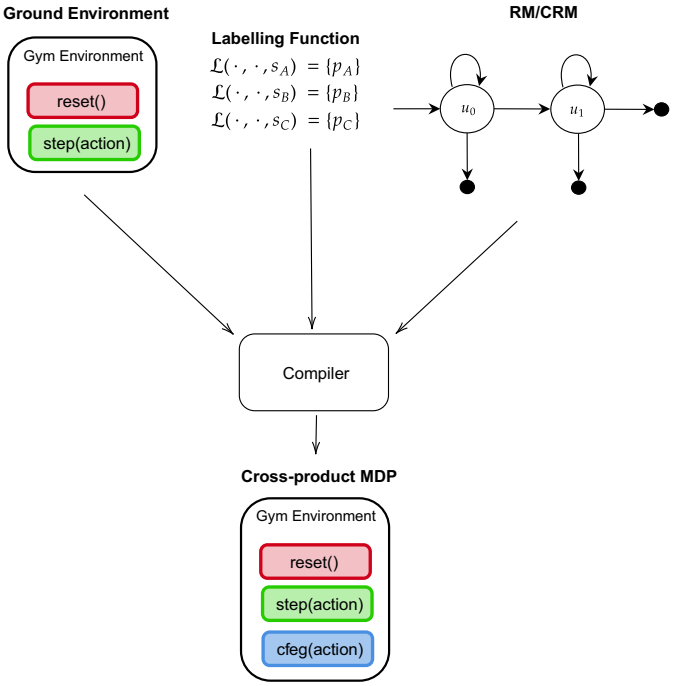

Fig. 2. The core architecture of PyCRM. The framework takes a standard Gymnasium environment, a user-defined labelling function, and a RM/CRM specification as input. The compiler automatically constructs a new, augmented Gymnasium environment that encapsulates the task logic and is compatible with standard RL agents. The new environment also exposes a method for counterfactual experience generation (`cfeg`—illustrated in blue). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

[1] PyCRM documentation is available here.

no constraints on the ground environment beyond compliance with the standard Gymnasium interface (`reset()`, `step(action)`), making it inherently agnostic to the environment's internal dynamics, including stochasticity, partial observability, and the structure of the state and action spaces.

2.2. Software functionalities

PyCRM provides a suite of functionalities to minimise implementation overhead while maximising flexibility and sample efficiency.

2.2.1. High-level specification and automatic wrapping

PyCRM offers a declarative, high-level API for defining RMs and CRMs, cleanly separating task specification from the learning algorithm. The framework automatically constructs a Gymnasium-compatible environment from these components, which transparently integrates the automaton's dynamics and augments the observation space to ensure that the Markov property is satisfied.

2.2.2. Counterfactual experience generation and algorithm integration

While the environment generated by PyCRM is compatible with any standard RL library, unlocking the full sample efficiency gains requires leveraging counterfactual experiences. A central feature of PyCRM is native support for counterfactual reasoning. For each transition, the framework generates counterfactual experiences by evaluating what would have happened if the automaton had been in a different state, dramatically improving sample efficiency. To leverage these experiences, PyCRM provides modified implementations of all off-policy algorithms in SB3 (DQN [18], DDPG [19], TD3 [20], and SAC [21]) as drop-in replacements equipped with specialised replay buffers. PyCRM additionally provides support for vectorized training to accelerate large-scale experiments (Appendix E).

3. Illustrative examples

We demonstrate the capabilities of PyCRM through two case studies in the *PuckWorld* environment [10], a continuous-state control problem featuring an agent, multiple moving targets, and an adversary (Fig. 3). Both discrete- and continuous-action variants of this environment exist. The first case study illustrates the core workflow for specifying and solving a sequential task with a standard RM in the discrete-action setting. The second case study highlights the framework's full expressive power by defining and solving a temporally extended task in the continuous-action setting using a CRM. While we focus on PuckWorld for narrative coherence—as it conveniently demonstrates both discrete and continuous action spaces as well as both RMs and CRMs within a single domain—additional environments, including stochastic settings, are provided in the repository and documentation.[2]

3.1. Case study 1: Reward machines

3.1.1. Problem description

We first consider a simple sequential task: the agent must track target 1, then target 2, and finally target 3, in that exact order. In this context, tracking is defined as staying within a specified distance of the target. This specification corresponds to the regular language $L = \{p_T^{(1)} p_T^{(2)} p_T^{(3)}\}$.[3]

3.1.2. Representing the task in PyCRM

To model this task, we first define the set of high-level events, $\mathcal{P} = \{p_T^{(1)}, p_T^{(2)}, p_T^{(3)}\}$, corresponding to the agent successfully tracking each target. This is implemented by subclassing `LabellingFunction` and defining event-detector methods decorated with `@event`, as shown in parts (1) and (2) of Listing 1. Each detector evaluates an environment transition $\langle s, a, s' \rangle$ and returns the corresponding event symbol if its condition is met—in this case, if the agent is within a threshold distance of the target.

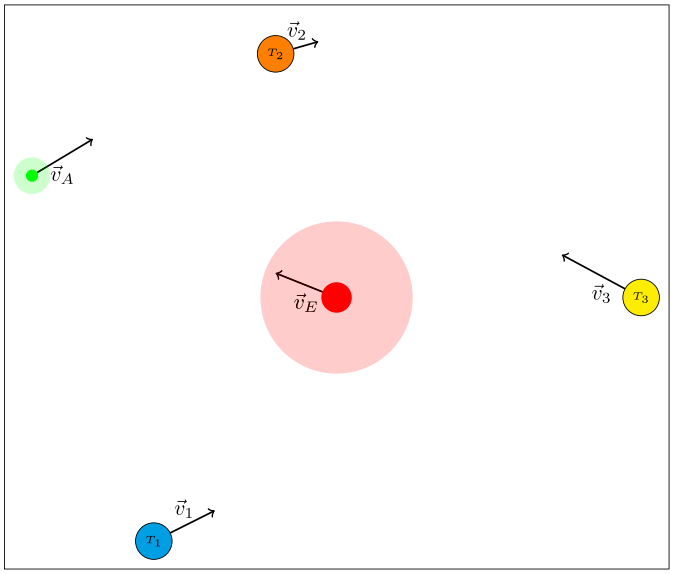

Fig. 3. Illustration of the extended *PuckWorld* environment [10]. The agent (green) must track a sequence of targets (T_1, T_2, T_3) while avoiding the adversary (red). An episode terminates if the agent enters the adversary's proximity zone (shaded red). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

```python
import numpy as np
from enum import Enum, auto
from pycrm.label import LabellingFunction
from pycrm.automaton import RewardMachine

# 1. Define high-level events
class PuckWorldEvent(Enum):
    T_1, T_2, T_3 = auto(), auto(), auto()

# 2. Implement the Labelling Function to detect events
class PuckWorldLF(LabellingFunction):
    @event
    def on_track_t1(self, s, a, s_prime) -> PuckWorldEvent | None:
        agent_pos, t1_pos, _, _ = self._unpack_obs(s_prime)
        if np.linalg.norm(agent_pos - t1_pos) < self.TARGET_THRESHOLD:
            return PuckWorldEvent.T_1
    # ... detectors for T_2 and T_3 are defined similarly ...

# 3. Define the Reward Machine for Case Study 1.
class PuckWorldRM(RewardMachine):
    def _get_state_transitions(self) -> dict:
        return {
            0: {"T_1": 1, "NOT T_1": 0},
            1: {"T_2": 2, "NOT T_2": 1},
            2: {"T_3": -1, "NOT T_3": 2},  # -1 is a terminal state
        }

    def _get_reward_transitions(self) -> dict:
        return {
            0: {"T_1": 10,   "NOT T_1": self._nav_to_t1_reward},
            1: {"T_2": 10,   "NOT T_2": self._nav_to_t2_reward},
            2: {"T_3": 1000, "NOT T_3": self._nav_to_t3_reward},
        }

# 4. Define the cross-product environment (default observation encoding)
class PuckWorldCP(CrossProduct):
    pass
```

Listing 1. Defining the labelling function and RM for the sequential tracking task. High-level events are defined as an `Enum` and detected by methods decorated with `@event`. The RM is specified declaratively via state-transition and state-reward dictionaries.

[2] See https://pycrm.xyz and the `examples/` directory in the repository.

[3] The formal language representation of a task describes the sequences of events that constitute successful task completion using symbols and syntactic rules. In this case, each target corresponds to a symbol, and the language defines the valid sequence of symbols.

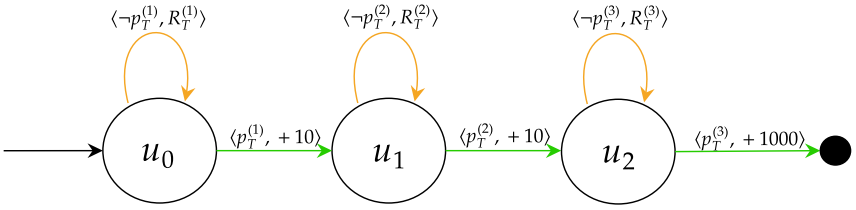

Fig. 4. Reward machine for the sequential tracking task $L = \{p_T^{(1)} p_T^{(2)} p_T^{(3)}\}$. Green arrows denote successful transitions with positive rewards. Orange arrows denote self-loops that provide dense navigation rewards ($R_T^{(i)}$) when the desired event is not observed. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

```python
from stable_baselines3 import DQN
from pycrm.crossproduct import CrossProduct
from pycrm.agents.sb3.dqn import CounterfactualDQN

# Instantiate components
ground_env = PuckWorld()
labelling_function = PuckWorldLF()
reward_machine = PuckWorldRM()

# Compile into a cross-product environment
cross_product_env = PuckWorldCP(
    ground_env=ground_env,
    machine=reward_machine,
    lf=labelling_function,
)

# Option 1: Train a standard SB3 agent (does not use counterfactuals)
# agent = DQN("MlpPolicy", env=cross_product_env)

# Option 2: Train a counterfactual-enabled agent (drop-in replacement)
agent = CounterfactualDQN("MlpPolicy", env=cross_product_env)

# The training call is identical for both agents
agent.learn(total_timesteps=1_000_000)
```

Listing 2. Solving the task by instantiating the user-defined cross-product MDP and training an RL agent.

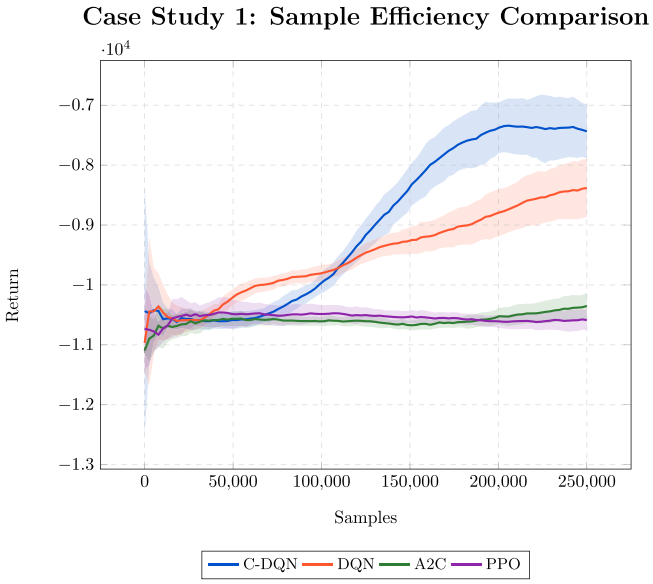

Fig. 5. Learning performance on the sequential *PuckWorld* task. Counterfactual-DQN (C-DQN) from PyCRM achieves significantly higher sample efficiency than the standard DQN baseline from SB3. On-policy baselines (A2C, PPO) are included for completeness. Results are averaged over 10 independent seeds; shaded regions indicate ±1 standard deviation.

Next, we specify the task logic and reward structure using an RM, whose structure is illustrated in Fig. 4. In PyCRM, this is achieved by subclassing the `RewardMachine` base class and implementing two methods that declaratively define the automaton's structure, as shown in part (3) of Listing 1.

The `_get_state_transitions` method defines the automaton's state-transition function. It returns a nested dictionary where outer keys represent non-terminal RM states ($u_0, u_1, \ldots$) and inner dictionaries map logical formulas over events to the next state. For example, the entry

$$0 : \{\texttt{"T\_1"} : 1, \texttt{"NOT T\_1"} : 0\}$$

specifies that from state u_0, observing the event $p_T^{(1)}$ transitions the automaton to state u_1, while observing any other event ($\neg p_T^{(1)}$) results in a self-loop. For convenience, the state index -1 is treated as a terminal state. Additional information related to the syntax of transition formulae is provided in Appendix B.

The `_get_reward_transitions` method defines the rewards associated with these transitions, following an identical structure. This method highlights the framework's flexibility: rewards can be simple scalar values (e.g., `10` for a correct step or `1000` for task completion) or entire callable functions. Here, `self._nav_to_t1_reward()` returns a function that provides a dense reward signal based on the agent's distance to the target, guiding the agent when it is not actively tracking the correct target. Additional details on the structure of reward functions are available in Appendix C.

3.1.3. Implementation and results

With the ground environment, labelling function, and RM defined, the final step is to combine them into a single MDP, known as the *cross-product MDP* [2,22,23], which can be solved by an RL agent. In PyCRM, this is implemented by subclassing `CrossProduct`. The base class provides a default implementation of the `_get_obs` method that concatenates the ground observation with a one-hot encoding of the machine state and the raw counter values—the standard encoding used in the literature [8,9]. A corresponding `to_ground_obs` method is also provided, which inverts this encoding to extract the original ground observation. Users may override these methods to implement custom observation encodings.

The resulting `PuckWorldCP` class can then be instantiated to create a standard Gymnasium environment, making it compatible with off-the-shelf RL libraries.[4] Listing 2 demonstrates this instantiation and subsequent training with a Deep Q-learning agent from SB3 [17,18]. PyCRM also provides implementations of common off-policy algorithms that leverage the counterfactual experiences generated by the cross-product MDP to improve sample efficiency. As PyCRM fully implements the SB3 interface, all existing code—including training routines, evaluation scripts, and utilities—works immediately with PyCRM agents; switching to a counterfactual-enabled agent requires only changing the class import. We additionally include on-policy baselines (A2C [24],

[4] As this class defines an MDP, it can be used with any RL algorithm, and all associated theoretical guarantees are preserved.

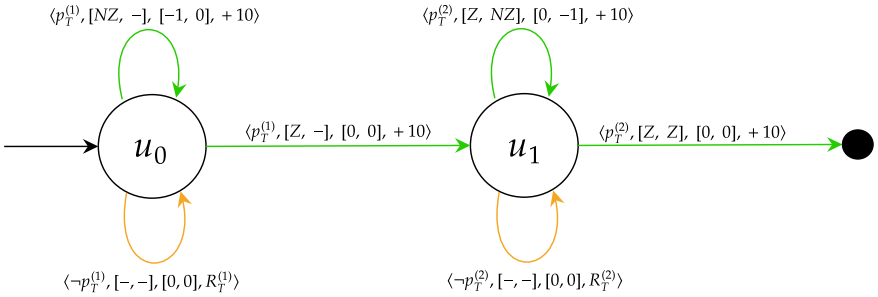

Fig. 6. Counting reward machine for the temporally extended task. The agent must decrement the first counter to zero (Z) by observing $p_T^{(1)}$ while it is non-zero (NZ) before it can transition to u_1. The same logic applies to $p_T^{(2)}$ and the second counter.

```python
from pycrm.automaton import CountingRewardMachine

class PuckWorldCRM(CountingRewardMachine):
    def _get_state_transitions(self) -> dict:
        return {
            0: {
                "T_1 / (Z,-)": 1,
                "T_1 / (NZ,-)": 0,
                "NOT T_1 / (-,-)": 0,
            },
            1: {
                "T_2 / (Z,Z)": -1,
                "T_2 / (Z,NZ)": 1,
                "NOT T_2 / (-,-)": 1,
            },
        }

    def _get_counter_updates(self) -> dict:
        return {
            0: {
                "T_1 / (Z,-)": (0, 0),
                "T_1 / (NZ,-)": (-1, 0),
                "NOT T_1 / (-,-)": (0, 0),
            },
            1: {
                "T_2 / (Z,Z)": (0, 0),
                "T_2 / (Z,NZ)": (0, -1),
                "NOT T_2 / (-,-)": (0, 0),
            },
        }

    def _get_reward_transitions(self) -> dict:
        return {
            0: {
                "T_1 / (Z,-)": 10,
                "T_1 / (NZ,-)": 10,
                "NOT T_1 / (-,-)": self._nav_to_t1_reward,
            },
            1: {
                "T_2 / (Z,Z)": 10,
                "T_2 / (Z,NZ)": 10,
                "NOT T_2 / (-,-)": self._nav_to_t2_reward,
            },
        }
```

Listing 3. Declarative specification of the CRM for the temporally extended task. The CRM is defined using dictionaries for state-transition, state-reward, and counter-update functions. The counter-update function encodes the counting logic, specifying how each counter is incremented or modified after the corresponding transition.

PPO [25]) for comparison. On-policy algorithms are inherently incompatible with counterfactual experience generation [8,9]; consequently, no counterfactual variants exist for these methods. Fig. 5 compares the performance of standard DQN with its counterfactual-enhanced counterpart, as well as the on-policy baselines, showing a clear improvement in learning speed and final performance.

3.2. Case study 2: Counting reward machines

3.2.1. Problem description

We now increase task complexity to demonstrate the full expressive power of PyCRM. The agent must track target T_1 for a pre-specified number of timesteps, $n \in \mathbb{N}$, and then track target T_2 for another n timesteps. Formally, this task is described by the language $L = \{(p_T^{(1)})^n (p_T^{(2)})^n \mid n \in \mathbb{N}\}$, which requires counting and is therefore non-regular, placing it beyond the expressive capacity of standard RMs, and thus necessitating the use of a CRM.[5]

3.2.2. Representing the task in PyCRM

Such tasks are naturally modelled using a CRM, such as the one employed for this case study, illustrated in Fig. 6. A CRM extends the RM formalism with a set of internal counters, allowing it to represent non-regular languages. Transitions in a CRM are governed by a more expressive rule, defined by a tuple: $\langle$event, [counter conditions], [counter updates], reward$\rangle$. A transition is taken only if both the event formula is satisfied and all counter conditions are met. The conditions for each counter can be:

- `Z`: The counter's value must be **Z**ero.
- `NZ`: The counter's value must be **N**on-**Z**ero.
- `-`: The counter's value is irrelevant.

When a transition is taken, the counters are modified according to the additive updates specified in the tuple. The CRM for our task (Fig. 6) uses two counters, c_0 and c_1, initialized to $n \in \mathbb{N}$ at the start of each episode. To progress from state u_0 to u_1, the agent must first satisfy the condition of the green self-loop on u_0: $\langle p_T^{(1)}, [\text{NZ}, -], [-1, 0], +10\rangle$. This rule specifies that if event $p_T^{(1)}$ is observed while counter c_0 is **N**on-**Z**ero, the automaton remains in u_0, c_0 is decremented by 1, and a reward of +10 is provided. For all other cases in which c_0 is not yet zero, the self-loop transition returns a function that provides a dense reward signal based on the agent's distance to the target, guiding the agent towards the correct target. Once c_0 reaches zero, this transition is no longer possible, and the transition from u_0 to u_1 becomes possible: $\langle p_T^{(1)}, [\text{Z}, -], [0, 0], +10\rangle$. This transition requires c_0 to be **Z**ero, and upon observing $p_T^{(1)}$, the automaton moves to u_1. A similar process with c_1 and $p_T^{(2)}$ is required to complete the task. This logic is specified declaratively in PyCRM as shown in Listing 3.

3.2.3. Implementation and results

Despite the increased task complexity, the solution procedure remains similar to the RM case. A cross-product class compiles the ground environment, labelling function, and CRM into a unified MDP. To ensure the state representation is Markovian, the agent's observation is constructed by concatenating the ground observation with the CRM's state and internal counter values [8–10].

This case study uses the continuous-action variant of *PuckWorld*. We evaluate off-policy algorithms (DDPG, TD3, and SAC) alongside on-policy baselines (A2C, PPO). Fig. 7 compares the performance of

[5] As with Case Study 1, the formal language encodes sequences of events that define successful task completion. Here, the language additionally requires maintaining a count of repeated events, illustrating the need for CRMs.

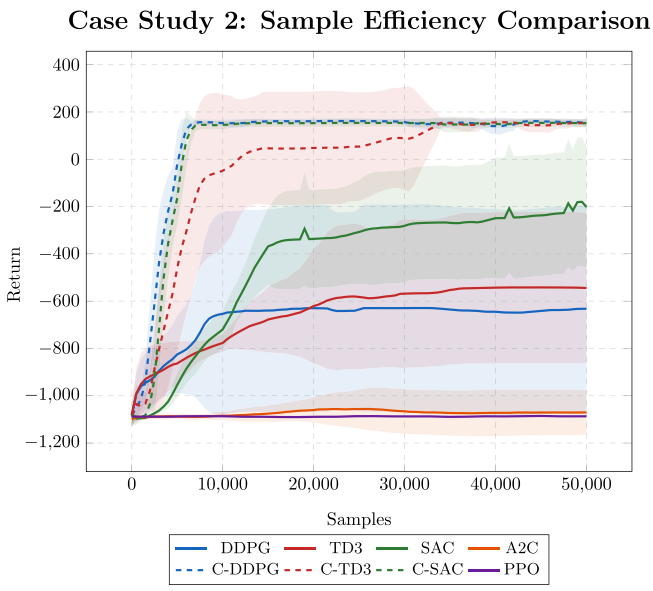

Fig. 7. Learning performance on the temporally extended, continuous-action *PuckWorld* task. All counterfactual-enabled agents successfully solve the task, demonstrating the significant sample efficiency gains of counterfactual methods in complex control problems when compared to standard off-policy and on-policy baselines. Results are averaged over 10 independent seeds; shaded regions indicate ±1 standard deviation.

standard SB3 agents with PyCRM's counterfactual-enabled counterparts, which are fully API-compatible and act as drop-in replacements. The results show that while standard agents and on-policy baselines fail to solve the complex, temporally extended task within the training budget, the counterfactual agents succeed with significantly higher sample efficiency. As noted in Section 3, counterfactual experience generation is incompatible with on-policy methods [8,9], so no counterfactual variants are provided for A2C and PPO. This highlights the advantages of PyCRM's counterfactual learning across multiple state-of-the-art algorithms in challenging, high-dimensional continuous control settings.

Importantly, the sample-efficiency gains of counterfactual learning come at a modest computational cost. Across both case studies, the wall-clock overhead of PyCRM's counterfactual agents relative to their standard SB3 counterparts ranges from approximately 2% to 15%, with throughput remaining comparable (Appendix D).

4. Impact

PyCRM significantly lowers the barrier to entry for non-Markovian RL by offering a user-friendly framework for specifying and using RMs and CRMs, reducing development time for both newcomers and experts. The framework accelerates experimentation by simplifying environment creation and integrating sample-efficient learning algorithms from SB3, enabling rapid prototyping and rigorous testing. PyCRM streamlines existing pipelines by eliminating the need for custom wrappers and has already proven essential for scaling experiments with complex reward structures in published work [9,10,26]. It enables previously impractical research directions, including zero-shot generalisation and hierarchical RL. Future extensions aim to bridge theoretical models and practical tools, transforming non-Markovian RL from a niche challenge into an accessible and scalable research direction.

5. Conclusion

We have introduced PyCRM, a Python software framework designed to address the significant implementation challenges associated with Reward Machine-based reinforcement learning. By providing a high-level declarative interface for specifying RMs and CRMs, seamless integration with standard RL libraries like Gymnasium and Stable Baselines 3, and built-in support for sample-efficient counterfactual learning, PyCRM automates the most complex aspects of the non-Markovian RL workflow.

By abstracting away the implementation complexity of automata-based reward specifications, our framework enables researchers to focus on high-level task design and algorithmic innovation rather than low-level engineering. Future work will extend PyCRM to support richer automata classes, tighter integration with emerging RL paradigms, and broader environment backends, further bridging the gap between theoretical models and practical deployment.

CRediT authorship contribution statement

Tristan Bester: Visualization, Validation, Software, Project administration, Methodology, Investigation, Formal analysis, Conceptualization. **Geraud Nangue Tasse:** Writing – review & editing, Supervision, Formal analysis, Conceptualization. **Benjamin Rosman:** Writing – review & editing, Supervision, Formal analysis, Conceptualization. **Steven James:** Writing – review & editing, Supervision, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the authors used Gemini 2.5 Pro to enhance the readability and language of the manuscript. After utilising this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

Appendix A. Background

This appendix provides a concise overview of the key concepts and notation used throughout the paper, including Markov decision processes, labelling functions, Reward Machines, and Counting Reward Machines.

A.1. Markov decision processes

Reinforcement learning formalises sequential decision-making as an agent interacting with an environment modelled as a *Markov decision process (MDP)* [1,27]. An MDP is defined as a tuple $\mathcal{M} = \langle S, A, P, \gamma, R \rangle$, where S is the state space, A is the action space, $P(s' \mid s, a) = \mathbb{P}(s_{t+1} = s' \mid s_t = s, a_t = a)$ defines the transition dynamics, $\gamma \in [0, 1)$ is the discount factor, and $R : S \times A \times S \to [r_{\min}, r_{\max}]$ is the bounded reward function. At each timestep, an agent in state $s \in S$ selects an action $a \in A$, transitions to a successor state s' according to P, and receives a scalar reward $R(s, a, s')$.

When the environment dynamics are specified independently of any particular task, we use the notation $\mathcal{M} \setminus R = \langle S, A, P, \gamma \rangle$ to denote an *MDP without reward* [8]. Different tasks can then be induced over the same environment by introducing different reward functions.

A.2. Labelling functions and propositional events

To bridge the gap between low-level environment transitions and high-level task specifications, we define a finite set of propositions $\mathcal{P}$ representing abstract events of interest (e.g., "target reached" or "obstacle hit"). A *labelling function* $\mathcal{L} : S \times A \times S \to 2^{\mathcal{P}}$ maps each environment transition $\langle s, a, s' \rangle$ to the subset of propositions that hold after

that transition. This abstraction provides a compact, symbolic description of the events that occurred, which can then be consumed by an automaton-based reward specification.

A.3. Reward machines

A *Reward Machine (RM)* [8] is a finite-state machine that specifies a non-Markovian reward function over sequences of high-level events. Formally, given an MDP without reward $\mathcal{M} \setminus R$ with propositions $\mathcal{P}$, an RM is defined as a tuple $\langle U, F, u_0, \delta_u, \delta_r \rangle$, where:

- U is a finite set of non-terminal machine states,
- F is a finite set of terminal states ($U \cap F = \emptyset$),
- $u_0 \in U$ is the initial state,
- $\delta_u : U \times 2^{\mathcal{P}} \to U \cup F$ is the state-transition function, and
- $\delta_r : U \times 2^{\mathcal{P}} \to (S \times A \times S \to [r_{\min}, r_{\max}])$ is the state-reward function, which outputs a reward function after each transition.

At each timestep, the labelling function produces an event set $\sigma = \mathcal{L}(s, a, s') \in 2^{\mathcal{P}}$, which drives the RM to transition from its current state u to $u' = \delta_u(u, \sigma)$ and emit a reward function $\delta_r(u, \sigma)$. This reward function is then evaluated on the ground transition $\langle s, a, s' \rangle$ to produce the scalar reward. In the common *simple* formulation, δ_r outputs a constant scalar rather than a function (see Appendix C). Since RMs are based on finite-state machines, they can represent exactly the class of regular languages, enabling the specification of tasks involving sequences, loops, and conditionals over high-level events.

A.4. Counting reward machines

Counting Reward Machines (CRMs) [9,10] generalise RMs by augmenting the finite-state machine with a set of k integer-valued counters, yielding a formalism based on k-counter machines. A 2-counter machine operating without temporal constraints is Turing-complete [28], and CRMs inherit this property, enabling them to express any reward function definable by a well-defined algorithm.

Formally, given an MDP without reward $\mathcal{M} \setminus R$ with propositions $\mathcal{P}$, a CRM is defined as a tuple $\langle U, F, u_0, \delta_u, \delta_c, \delta_r \rangle$, where U, F, and u_0 are as in an RM, and:

- $\delta_u : U \times 2^{\mathcal{P}} \times \{0,1\}^k \to U \cup F$ is the state-transition function,
- $\delta_c : U \times 2^{\mathcal{P}} \times \{0,1\}^k \to \{+m : m \in \mathbb{Z}\}^k$ is the counter-update function, and
- $\delta_r : U \times 2^{\mathcal{P}} \times \{0,1\}^k \to (S \times A \times S \to [r_{\min}, r_{\max}])$ is the state-reward function.

Each function additionally depends on the zero-tested counter values $Z(\boldsymbol{c}) \in \{0,1\}^k$, where $Z(\boldsymbol{c})_i = 0$ if $c_i = 0$ and 1 otherwise. This zero-test mechanism enables the machine to branch on whether a counter has reached zero, providing the counting capability that elevates CRMs beyond regular languages. Given a machine configuration $\langle u, \boldsymbol{c} \rangle$ and an event set $\sigma = \mathcal{L}(s, a, s')$, the CRM updates as:

$$u' = \delta_u(u, \sigma, Z(\boldsymbol{c})),$$
$$\boldsymbol{c}' = \boldsymbol{c} + \delta_c(u, \sigma, Z(\boldsymbol{c})),$$
$$r = \delta_r(u, \sigma, Z(\boldsymbol{c}))(s, a, s').$$

A.5. Cross-product MDP

Since the reward generated by an RM or CRM depends on the machine's internal state (and counter values), the ground environment observation alone is insufficient for Markovian decision-making. To restore the Markov property, the ground environment is combined with the automaton into a *cross-product MDP* [2,22,23]. The state space of the cross-product MDP is the Cartesian product $S \times C$, where C denotes the set of automaton configurations (machine state for RMs; machine state and counter values for CRMs). The transition dynamics jointly evolve both the ground state and the automaton configuration, and the reward is determined by the automaton's output function. Since the resulting MDP is fully Markovian, standard RL algorithms can be applied directly, and all associated convergence guarantees are preserved.

Appendix B. Transition formulae

PyCRM supports a flexible syntax for defining the transition conditions of RMs/CRMs, allowing for the creation of complex task structures based on logical combinations of high-level events. These formulas are evaluated at each step to determine which state transition should occur in the automaton. The syntax is designed to be intuitive and expressive, accommodating a wide range of logical constructs.

B.1. Propositional events

The fundamental building blocks of transition formulae are propositional events. These are user-defined symbols, represented as strings (e.g., "`T_1`", "`door_open`"), that correspond to specific events detected in the environment by the `LabellingFunction`. A formula is constructed by combining these propositions with logical operators.

B.2. Logical operators

The framework supports standard Boolean operators to create expressive formulae:

- **NOT:** The negation operator is denoted by the prefix `NOT`. It inverts the truth value of a proposition. For example, `NOT T_1` is true only if the event `T_1` did not occur.
- **AND:** The conjunction operator, represented by `AND`, is used to combine two or more propositions. The resulting formula is true only if all constituent propositions are true. For example, `T_1 AND T_2` is true if both `T_1` and `T_2` events occur simultaneously.
- **OR:** The disjunction operator, represented by `OR`, is also used to combine propositions. The formula is true if at least one of the constituent propositions is true. For example, `T_1 OR T_2` is true if `T_1` occurred, `T_2` occurred, or both occurred.

Parentheses `()` can be used to group expressions and enforce a specific order of operations.

B.3. Example formulae

Here are some examples of transition formulae supported by PyCRM, ranging from simple to complex:

- **Simple Propositions:**
  - "`T_1`": The machine transitions if the event `T_1` is detected.
  - "`NOT T_1`": The machine transitions if the event `T_1` is *not* detected.
- **Compound Formulae:**
  - "`T_1 AND T_2`": Transition occurs if both `T_1` and `T_2` occur.
  - "`T_1 OR T_2`": Transition occurs if either `T_1` or `T_2` (or both) occur.
  - "`(T_1 OR T_2) AND NOT T_3`": Transition occurs if the agent has reached either target 1 or target 2, but has not reached target 3.
- **De Morgan's Laws:** PyCRM's syntax is expressive enough to support logical equivalences such as De Morgan's laws, which are useful for simplifying complex conditions:
  - "`NOT (T_1 OR T_2)`" is equivalent to "`NOT T_1 AND NOT T_2`". This formula is true only when neither `T_1` nor `T_2` occurs.
  - "`NOT (T_1 AND T_2)`" is equivalent to "`NOT T_1 OR NOT T_2`". This formula is true if at least one of `T_1` or `T_2` does not occur.

B.4. Counter conditions for CRMs

When using Counting Reward Machines (CRMs), the transition logic is extended to depend not only on propositional events but also on the

current values of the internal counters. This allows for the specification of non-regular tasks that require memory or counting. The syntax for a transition formula is augmented to include these counter conditions. The general format for a CRM transition is: "`<event_formula>` / (`<counter_conditions>`)"

Here, the `<event_formula>` is the same logical expression over propositions as used in standard RMs. The `<counter_conditions>` part is a tuple of characters, where each character specifies a condition that the corresponding counter must satisfy for the transition to be valid. The conditions are specified as a comma-separated string, with the position corresponding to the counter's index (e.g., the first condition applies to c_0, the second to c_1, and so on). The supported conditions for each counter are:

- `Z` (Zero): The counter's current value must be exactly 0.
- `NZ` (Non-Zero): The counter's current value must be anything other than 0.
- `-` (Irrelevant): The counter's value does not affect the transition; it acts as a wildcard.

B.5. Example CRM formulae

Building on the previous examples, here is how counter conditions can be integrated:

- **Simple Counter Condition:**
  - `"T_1 / (NZ, -)"`: The machine transitions if event `T_1` is detected AND the first counter (c_0) is non-zero. The value of the second counter (c_1) is ignored.
- **Zero-Check Condition:**
  - `"T_1 / (Z, -)"`: The transition occurs if the event `T_1` is detected AND the first counter (c_0) is zero. This is useful for triggering an event only after a countdown has been completed.
- **Complex Combined Conditions:**
  - `"T_2 / (Z, NZ)"`: The transition is taken if event `T_2` occurs, the first counter (c_0) is zero, AND the second counter (c_1) is non-zero. This allows for creating sequential dependencies based on multiple counters.
  - `"(T_1 OR T_2) / (NZ, NZ)"`: The transition occurs if either event `T_1` or `T_2` is detected, provided that *both* counter c_0 and counter c_1 are non-zero.
  - `"NOT T_1 / (-, Z)"`: The transition is taken if event `T_1` does *not* occur AND the second counter (c_1) is zero.

This extended syntax provides the expressive power needed to formally specify complex, history-dependent tasks that are beyond the scope of standard Reward Machines.

Appendix C. Reward specification: simple and full formulations

PyCRM provides a flexible reward specification mechanism that supports both the "simple" and "full" formulations of RMs and CRMs. This design allows users to define straightforward, constant rewards for most transitions while offering the expressive power of dynamic reward functions for more complex reward shaping scenarios.

C.1. The "Simple" vs. "Full" formulations

In the literature, Reward Machines can be defined in two primary ways:

- **Simple Formulation:** In this version, each transition in the automaton is associated with a fixed, scalar reward value. For example, successfully tracking target 1 might always yield a reward of +10. This is the most common and intuitive formulation, where the reward is a property of the automaton's state change itself, independent of the underlying environment dynamics.
- **Full Formulation:** The more general or "full" formulation associates each transition with a *reward function*, not just a scalar. This function takes the low-level environment transition (s, a, s') as input and returns a reward. The reward can therefore depend on the specific states and actions that caused the high-level event to occur.

The simple formulation is a special case of the full formulation. A simple RM with a scalar reward r for a given transition can be expressed in the full formulation by defining a constant reward function $R(s, a, s') = r$ that ignores its inputs and always returns r.

C.2. Unified implementation in PyCRM

PyCRM leverages this relationship to provide a powerful yet user-friendly API. The framework's internal architecture is built around the more general "full" formulation, allowing it to handle dynamic, state-dependent rewards natively. However, for user convenience, it seamlessly accepts rewards specified in the simple style. When a user provides a scalar value (e.g., `1000`) for a transition reward, PyCRM internally **compiles** this simple specification into the full format. It automatically constructs a constant "reward function callable" that wraps the scalar. This internal function conforms to the required signature but simply returns the fixed reward value for any input. This design offers two key advantages:

1. **User Convenience:** Researchers can use simple, intuitive scalar rewards for most cases without needing to define trivial functions.
2. **Internal Consistency:** The core learning algorithms only need to handle one type of reward signal—the output of a callable function—which simplifies the internal logic and reduces redundancy.

C.3. Reward function callable signature

For transitions that require dynamic rewards, users can provide a Python callable (e.g., a function or a class method). This function is executed at every transition, and its return value is used as the reward signal. The signature for a reward function callable is as follows:

```
reward_function(s, a, s_prime) -> float
```

The function receives a comprehensive set of arguments that describe the full transition:

- `s`: The state of the ground environment *before* the action was taken.
- `a`: The action executed by the agent.
- `s_prime`: The state of the ground environment *after* the action was taken.

These arguments can then be used to compute a feedback signal for the agent in the form of a reward scalar.

Appendix D. Computational overhead analysis

To quantify the computational cost of counterfactual experience generation, we measure wall-clock training time and throughput (frames per second) for one representative algorithm per action-space type: DQN for discrete actions (Case Study 1) and SAC for continuous actions (Case Study 2). SAC is chosen as representative because the counterfactual overhead is identical across continuous-action algorithms—it stems from evaluating the labelling function and reward transitions for every non-current machine state, which is independent of the policy update rule. TD3 and DDPG therefore exhibit comparable overhead. All runs use identical hyperparameters and hardware, with 10 independent seeds per algorithm. We report medians and interquartile ranges (IQR) to provide robust estimates.

Table D.1 summarises the results. For the discrete-action setting (Case Study 1), C-DQN incurs a 14.7% wall-clock overhead relative to standard DQN (median 322.9 s vs. 281.5 s), corresponding to a throughput reduction from 889 to 806 frames per second. In the continuous-action setting (Case Study 2), the overhead is substantially smaller:

Table D.1
Computational overhead of counterfactual experience generation. Wall-clock time and throughput (FPS) are reported as median (IQR) over 10 runs.

Metric	Algorithm	Standard	Counterfactual	Δ (%)
Time (s)	DQN	282 (280–283)	323 (320–327)	+14.7%
	SAC	1037 (1025–1038)	1057 (1039–1078)	+1.9%
FPS	DQN	889 (855–928)	806 (767–828)	−9.3%
	SAC	48 (48–49)	47 (46–48)	−3.0%

```python
from pycrm.agents.sb3.vec import DispatchSubprocVecEnv
from pycrm.agents.sb3.sac import CounterfactualSAC

# Create a vectorised environment for parallel execution
envs = DispatchSubprocVecEnv([
    lambda: create_cross_product_env() for _ in range(8)
])

# The agent automatically leverages the vectorised environment
agent = CounterfactualSAC(
    policy="MlpPolicy",
    env=envs,
    verbose=1
)

agent.learn(total_timesteps=1_000_000)
```

Listing 4. Example usage of `DispatchSubprocVecEnv` to create a vectorized environment for parallel training with a counterfactual-enabled agent.

C-SAC adds only 1.9% to SAC's training time (median 1056.5 s vs. 1036.8 s), with throughput decreasing marginally from 48.4 to 47.0 FPS. The higher relative overhead for DQN is expected: DQN's gradient update is considerably cheaper than SAC's (which optimises an actor, a critic, and an entropy coefficient), so the fixed cost of counterfactual experience generation constitutes a larger fraction of the total per-step computation.

Crucially, this modest overhead must be weighed against the substantial sample-efficiency gains demonstrated in Figs. 5 and Fig. 7. The overhead arises from evaluating the labelling function and reward transitions for every non-current machine state at each environment step, and therefore scales linearly with the number of RM/CRM states—a quantity that remains small for typical task specifications. In practice, the net effect is faster convergence in wall-clock time despite the per-step cost increase, as far fewer environment interactions are needed to reach a given performance level.

Appendix E. Vectorized environment support

To support large-scale experiments and accelerate training with modern off-policy algorithms, PyCRM provides specialised support for vectorized environments. This is crucial for leveraging parallel hardware to improve sample throughput. The core of this support is the `DispatchSubprocVecEnv`, located in the `crm.agents.sb3.vec` module. This class is an extension of Stable Baselines 3's standard `SubprocVecEnv` and is specifically designed to enable the efficient, parallel generation of counterfactual experiences across multiple environment instances. By dispatching the counterfactual generation logic to the subprocesses, it minimises the communication overhead that would otherwise create a bottleneck.

This allows counterfactual-enabled agents, such as `CounterfactualSAC`, to scale effectively, combining the sample efficiency gains from counterfactual learning with the throughput benefits of vectorization. The implementation is designed for seamless integration, requiring minimal changes to the standard training workflow, as shown in Listing 4. By using `DispatchSubprocVecEnv`, researchers can significantly speed up convergence. The agent benefits from both a higher volume of environment interactions per second and the enhanced sample efficiency provided by learning from counterfactual experiences generated in parallel.

References

[1] Sutton RS, Barto AG, et al. Reinforcement learning. In: An introduction, vol. 1. MIT Press Cambridge; 1998.
[2] Gaon M, Brafman R. Reinforcement learning with non-Markovian rewards. In: Proceedings of the AAAI conference on artificial intelligence, vol. 34; 2020. p. 3980–7.
[3] Gupta G, Yin C, Deshmukh JV, Bogdan P. Non-Markovian reinforcement learning using fractional dynamics. In: 2021 60th IEEE conference on decision and control (CDC). IEEE; 2021. p. 1542–7.
[4] Huang R, Liang Y, Yang J. Robust offline reinforcement learning for non-Markovian decision processes. IEEE Trans Inf Theory 2025;71(9):7208–28. https://ieeexplore.ieee.org/document/11075721.
[5] Tang Y, Cai X-Q, Pang J-C, Wu Q, Ding Y-X, Sugiyama M. Beyond simple sum of delayed rewards: arXiv preprint arXiv:2410.20176. 2024.
[6] Early J, Bewley T, Evers C, Ramchurn S. Non-Markovian reward modelling from trajectory labels via interpretable multiple instance learning. Adv Neural Inf Process Syst 2022;35:27652–63.
[7] Dohmen T, Topper N, Atia G, Beckus A, Trivedi A, Velasquez A. Inferring probabilistic reward machines from non-Markovian reward signals for reinforcement learning. In: Proceedings of the international conference on automated planning and scheduling, vol. 32; 2022. p. 574–82.
[8] Icarte RT, Klassen TQ, Valenzano R, McIlraith SA. Reward machines: exploiting reward function structure in reinforcement learning. J Artif Intell Res 2022;73:173–208.
[9] Bester T, Rosman B, James S, Tasse GN. Counting reward automata: sample efficient reinforcement learning through the exploitation of reward function structure. In: NuCLeaR workshop at the 38th AAAI conference on artificial intelligence; 2024.
[10] Bester T. Counting reward automata: exploiting structure in reward functions expressible in decidable formal languages [Master's thesis]. Johannesburg: University of the Witwatersrand; 2024.
[11] Ibrahim S, Mostafa M, Jnadi A, Salloum H, Osinenko P. Comprehensive overview of reward engineering and shaping in advancing reinforcement learning applications. IEEE Access; 2024.
[12] Icarte RT, Klassen T, Valenzano R, McIlraith S. Using reward machines for high-level task specification and decomposition in reinforcement learning. In: International conference on machine learning. PMLR; 2018. p. 2107–16.
[13] Furelos-Blanco D, Law M, Jonsson A, Broda K, Russo A. Hierarchies of reward machines. In: International conference on machine learning. PMLR; 2023. p. 10494–541.
[14] Umili E, Argenziano F, Capobianco R. Neural reward machines. In: ECAI 2024 – 27th European conference on artificial intelligence, Vol. 392 of frontiers in artificial intelligence and applications, IOS Press; 2024. p. 3055–62. https://doi.org/10.3233/FAIA240847
[15] Corazza J, Gavran I, Neider D. Reinforcement learning with stochastic reward machines. In: Proceedings of the AAAI conference on artificial intelligence, vol. 36; 2022. p. 6429–36.
[16] Towers M, Kwiatkowski A, Terry J, Balis JU, De Cola G, Deleu T, Goulão M, Kallinteris A, Krimmel M, Kg A, et al. Gymnasium: a standard interface for reinforcement learning environments. In: Advances in neural information processing systems 38 (NeurIPS), datasets and benchmarks track; 2024.
[17] Raffin A, Hill A, Gleave A, Kanervisto A, Ernestus M, Dormann N. Stable-baselines3: reliable reinforcement learning implementations. J Mach Learn Res 2021;22(268):1–8.
[18] Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, Graves A, Riedmiller M, Fidjeland AK, Ostrovski G, et al. Human-level control through deep reinforcement learning. Nature 2015;518(7540):529–33.
[19] Lillicrap TP, Hunt JJ, Pritzel A, Heess N, Erez T, Tassa Y, Silver D, Wierstra D. Continuous control with deep reinforcement learning. In: 4th international conference on learning representations (ICLR); 2016.
[20] Fujimoto S, Hoof H, Meger D. Addressing function approximation error in actor-critic methods. In: International conference on machine learning. PMLR; 2018. p. 1587–96.
[21] Haarnoja T, Zhou A, Abbeel P, Levine S. Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: International conference on machine learning. Pmlr; 2018. p. 1861–70.
[22] Abate A, Almulla Y, Fox J, Hyland D, Wooldridge M. Learning task automata for reinforcement learning using hidden Markov models. In: ECAI 2023 – 26th European conference on artificial intelligence, Vol. 372 of frontiers in artificial intelligence and applications, IOS Press; 2023. p. 3–10. https://doi.org/10.3233/FAIA230247
[23] Brafman R, De Giacomo G, Patrizi F. LTLf/LDLf non-Markovian rewards. In: Proceedings of the AAAI conference on artificial intelligence, vol. 32; 2018.
[24] Mnih V, Badia AP, Mirza M, Graves A, Lillicrap T, Harley T, Silver D, Kavukcuoglu K. Asynchronous methods for deep reinforcement learning. In: International conference on machine learning. PMLR; 2016. p. 1928–37.
[25] Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O, arXiv preprint arXiv:1707.06347. 2017.
[26] Varricchione G, Klassen TQ, Alechina N, Dastani M, Logan B, McIlraith SA. Pushdown reward machines for reinforcement learning. In: Proceedings of the 22nd international conference on principles of knowledge representation and reasoning (KR); 2025. p. 566–76. https://doi.org/10.24963/kr.2025/55
[27] Puterman ML. Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons; 2014.
[28] Fischer PC, Meyer AR, Rosenberg AL. Counter machines and counter languages. Math. Syst. Theory 1968;2(3):265–83.