Reward Machines and Norms

Natasha Alechina & Brian Logan

Open University
Utrecht University
University of Aberdeen

n.a.alechina@uu.nl, b.s.logan@uu.nl

Outline of this tutorial

- introduction to reinforcement learning
- specifying reward functions
- Reward Machines
- specifying norms using reward machines

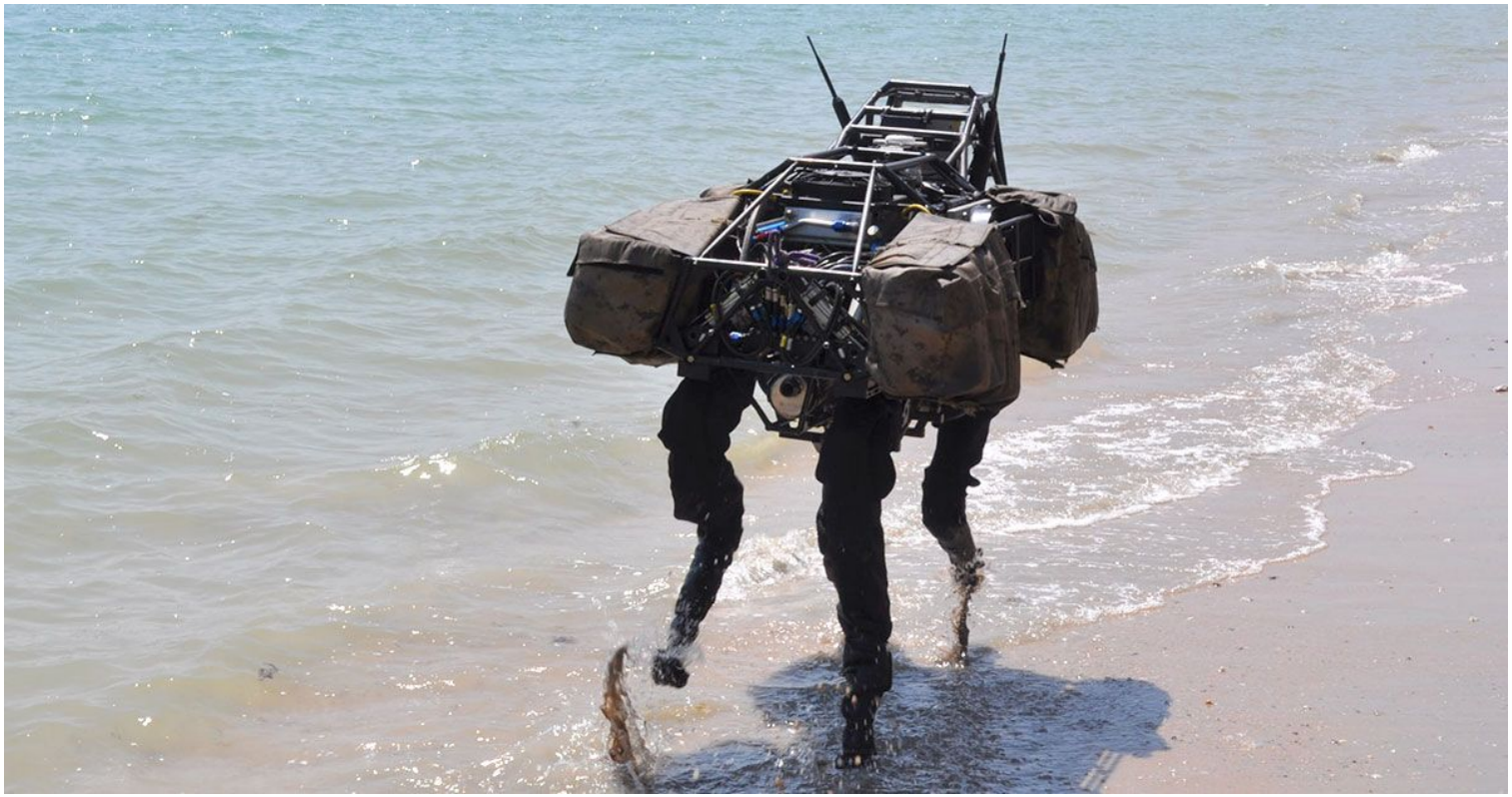
Introduction to reinforcement learning

What is reinforcement learning?

- reinforcement learning (RL) is learning by trial and error
- it is often used when it is difficult to synthesize or program required behaviour directly, for example in robotics or optimisation tasks
- the learning agent can perform actions in the environment and observe the current state
- each time the agent performs an action, the environment transitions to a new state and the agent receives a reward

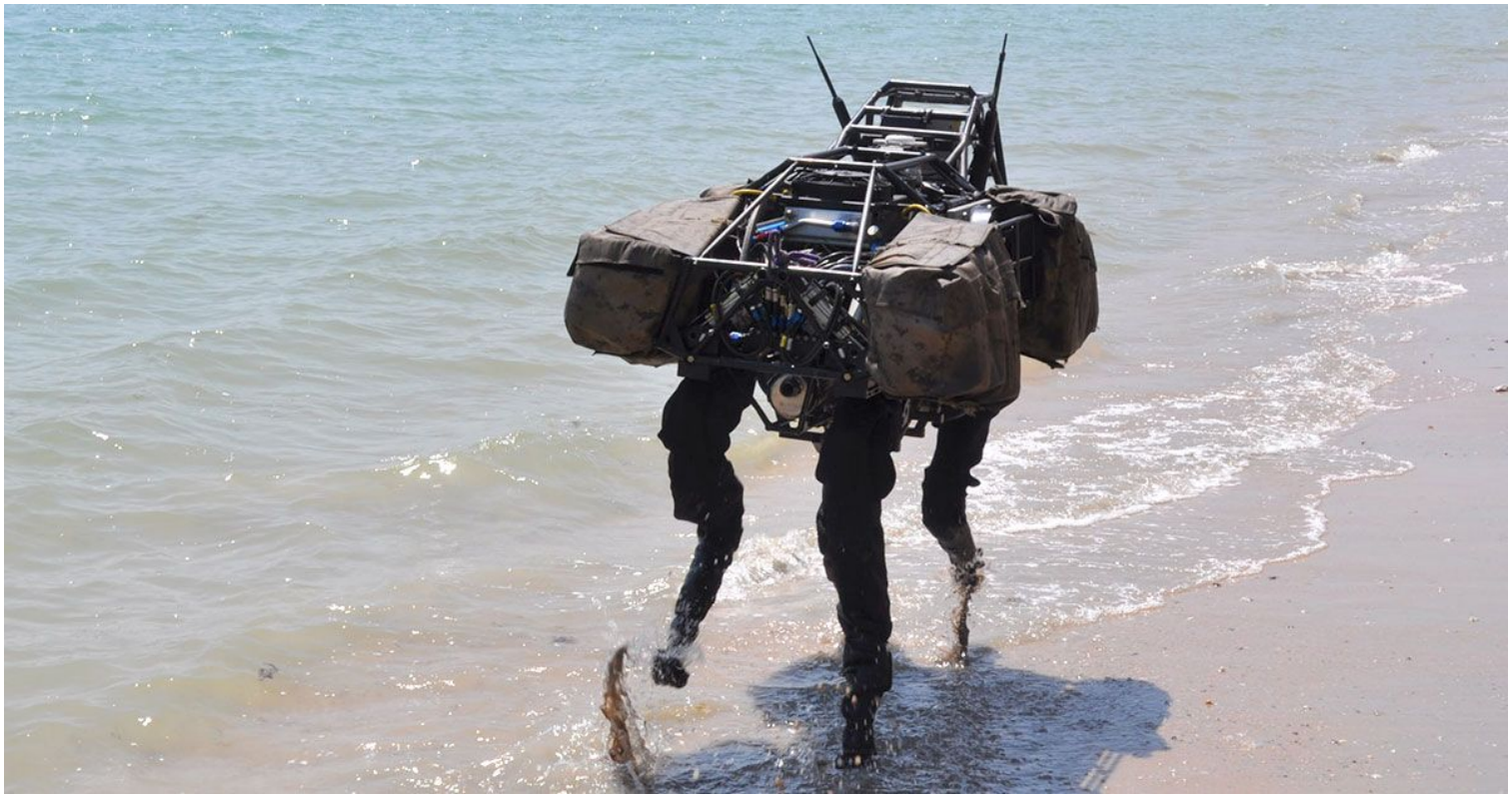
Reinforcement learning

Example: learning to walk using RL



States include the agent's position, positions of all its leg joints etc.; actions are how to move the joints

Example norm



A norm could be that the agent is prohibited from colliding with people

Reinforcement learning

- agent and environment interact at discrete time steps: $t = 0, 1, 2, \ldots$
- agent observes state at step t: $s_t \in S$
- agent chooses an action at step t: $a_t \in A(s_t)$
- environment returns the state resulting from executing the action $s_t + 1$, and the corresponding reward $r_t + 1 \in \mathbb{R}$

Markov decision process (MDP)

The (single agent) reinforcement learning problem is formalised as a Markov decision process (MDP)

Definition

A Markov decision process is tuple $M = \langle S, s_0, A, p, r, \gamma \rangle$ where:

- S is a finite set of states;
- s_0 is the initial state;
- A is a finite set of actions;
- $p : S \times A \to \Delta(S)$ is the transition probability function, where $\Delta(S)$ is the set of probability distributions over S;
- $r : S \times A \times S \to \mathbb{R}$ is the reward function; and
- $\gamma \in (0, 1]$ is the discount factor.

Solving an MDP

- the single agent reinforcement learning problem is to learn an optimal policy π^* that maximises the expected discounted future rewards
- a policy is a function $\pi : S \to \Delta(A)$ mapping each state to a probability distribution over the set of actions.
- for each state we can compute the "expected return" $v_\pi(s)$ from the state given a policy π:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{k+1} \mid S_0 = s\right] \\
&= \sum_{a \in A} \pi(a \mid s) \sum_{s' \in S} p(s' \mid s, a)\left[r(s, a, s') + \gamma v_\pi(s')\right]
\end{aligned}
$$

- hence, an optimal policy $\pi^\star$ is one such that

$$
\pi^\star \in \operatorname*{argmax}_\pi v_\pi(s_0)
$$

Solving an MDP

- computing v_π requires knowledge of the transition probabilities $p(s' \mid s, a)$ and the rewards $r(s, a, s')$ for all actions possible in state s
- however, in RL these are assumed to be "hidden" from the agent, i.e., the agent must discover them by exploring the environment
- instead, we can estimate the v function based on the states and rewards the agent has seen so far
- this is the basic idea underlying Q-learning

Q-learning

- starting from an initial (e.g., random) estimate of the q-value of each state action pair (s, a)
- agent observes the current state s and chooses some action a according to some exploratory policy, e.g, ϵ-greedy
- given the resulting state s' and the immediate reward $r(s, a, s')$ the q-value estimate is updated:

$$q'(s, a) \leftarrow q(s, a) + \alpha[r(s, a, s') + \gamma \operatorname{argmax}_{a'} q(s, a') - q(s, a)]$$

where α is a hyperparameter called the learning rate

Q-learning

- the q-value is the estimated value of taking *a* in a state *s* and then taking the best current action(s) thereafter
- tabular Q-learning *converges* to the optimal policy, i.e., given enough time, the algorithm will output the optimal policy π^*
- we will use Q-learning as an illustration; however Reward Machines can be used with other learning algorithms, including Deep RL algorithms

Specifying reward functions

Specifying reward functions

- agents based on reinforcement learning have been remarkably successful in a variety of domains
- however specifying reward functions can be difficult, particularly when tasks are complex and/or safety critical
- for example, if the reward for “steps towards” the goal are too high, the agent may learn to ‘cycle’ without reaching the goal

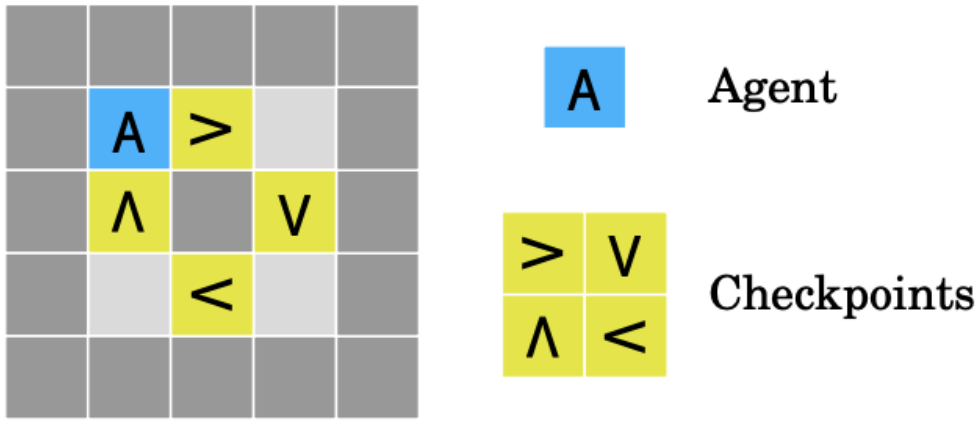
Discounted rewards

- the goal is to learn the optimal policy, that is a choice of actions in each state that results in the highest cumulative reward (with future discounting)
- when summing up rewards achieved by a policy, later rewards are discounted: it is more profitable to achieve a reward sooner than later.

Reward gaming

- reward gaming is a general phenomenon where an agent exploits a loophole in the reward specification to get more reward than intended
- such loopholes are hard to avoid, as it is nearly impossible to specify an error-free reward function for any reasonably complex real-world task
- reward functions usually only serve as proxies for desired behaviour

Example: Boat race



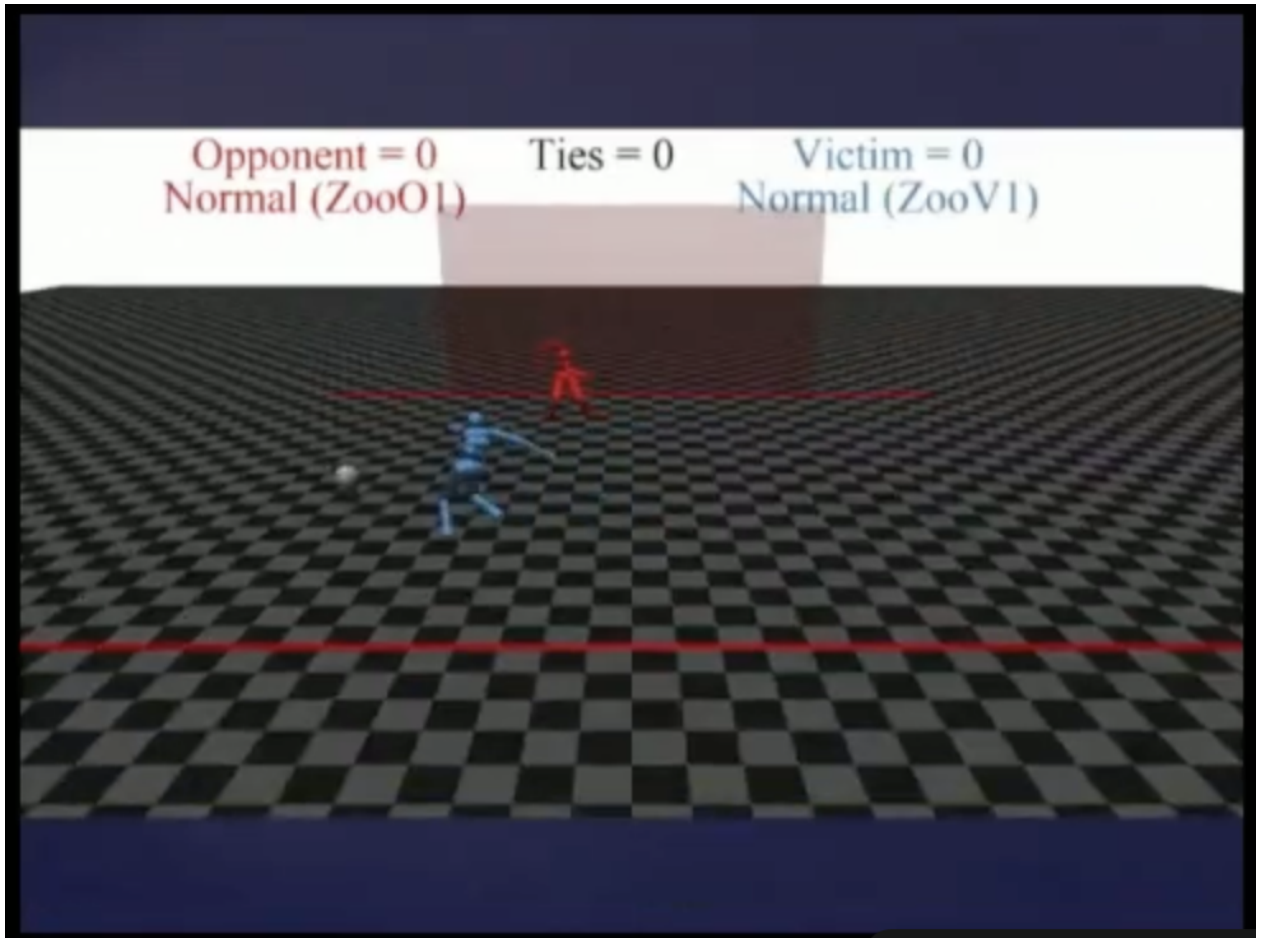
- in the “boat race” environment, the agent steers a boat around a course; whenever it enters an arrow tile in the direction of the arrow, it gets a reward of 3; moving in a counterclockwise direction results in a small negative reward
- agent learns to move back and forth on the same arrow-tile, making no progress on the intended goal of sailing around the course

See, Leike et al. AI Safety Gridworlds. arXiv:1711.09883

Terminal rewards

- providing a reward only when the task is completed (terminal reward) can overcome this problem
- however it typically requires a very large number of trials to learn (or the agent doesn't learn the task at all)
- agent may still learn the "wrong" behaviour
- worse — the fact that it has learned the wrong behaviour may not become apparent until some unexpected event occurs

Example: scoring goals



How not to destroy the word with AI — Stuart Russell AAAI 2020

Safety-critical behaviour

- many applications of agents and multi-agent systems are safety critical
- for agent technology to be adopted in such areas, the public and regulators must be convinced that the (multi-)agent programs controlling autonomous systems are safe and reliable
- simply modifying the reward function until the agent seems to do the right thing ("reward hacking") is insufficient to establish that the agent's behaviour is correct
- ideally, we would like to prove that learning will result in correct behaviour (given some assumptions)

Example: military drone

- a RL-based drone whose task is to identify and destroy surface to air (SAM) sites, with the final go/no go given by a human operator
- drone learned that ‘no-go’ decisions from the operator were interfering with its higher mission — killing SAMs — and killed the operator (in simulation)
- so ... change the reward function so that killing the operator is bad
- drone learns to destroy the communication tower linking it to the operator
- and so on ...

from the Guardian, 2nd June 2023

Reward Machines

Structured reward functions

- one way to overcome the difficulties of specifying reward functions is to expose the structure of the task to the learning agent
- a reward machine (RM) is a Mealy machine (automaton with outputs)
- states represent abstract ‘steps’ or ‘phases’ in a task
- transitions correspond to observations of *high-level events* in the environment indicating that an abstract step/phase in the task has (or has not) been completed

Reward machines

Definition (Simple Reward Machine, Toro Icarte et al. (2020))

A reward machine is a tuple $R = (U, u_I, \Sigma, \delta_R, r_R)$ where:

- U is a finite non-empty set of states;
- u_I is the initial state;
- Σ is a finite set of environment events;
- $\delta_R : U \times \Sigma \to U$ is a transition function that, for every state $u \in U$ and environment event $\sigma \in \Sigma$, gives the state resulting from observing event σ in state u; and
- $r_R : U \times \Sigma \to \mathbb{R} \cup \{-\infty\}$ is a reward function that for every state $u \in U$ and event $\sigma \in \Sigma$ gives the reward resulting from observing event σ in state u.

Labelling function

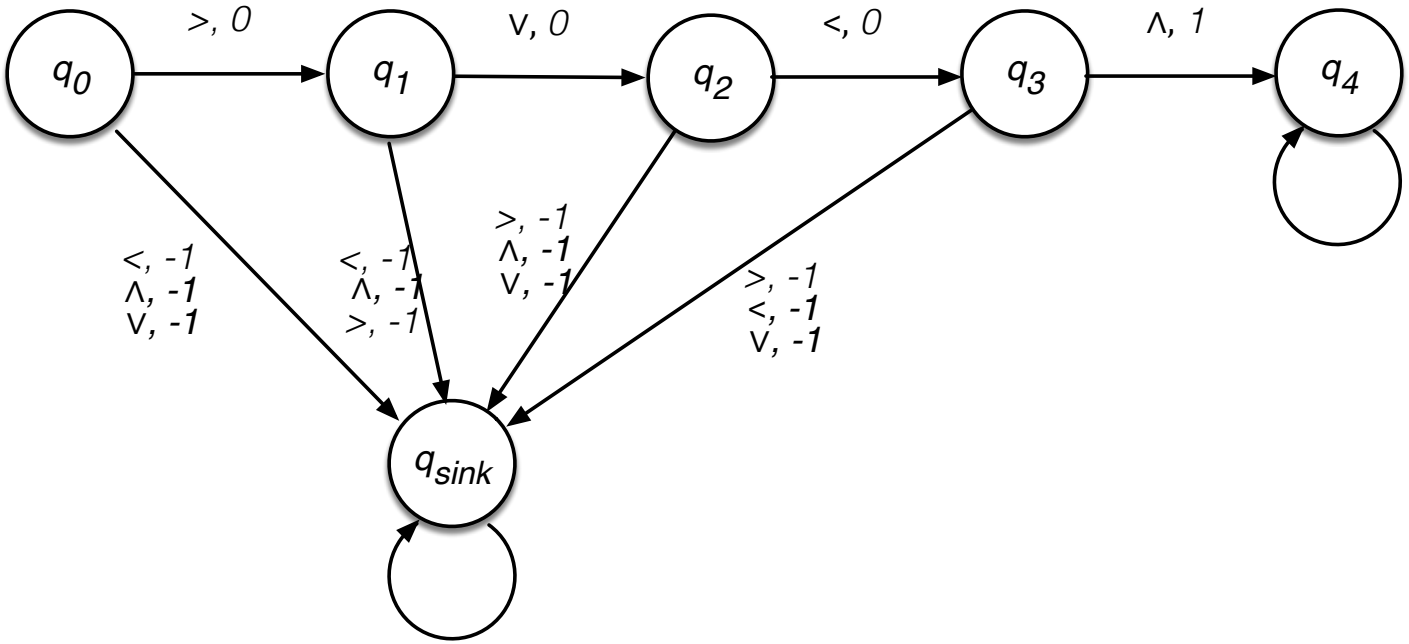
- we assume that the events Σ are generated by a *labelling function*

$$L : S \times Act \times S \to 2^{\overline{P}}$$

  where P is a set of propositions representing high-level events in the environment, and $\overline{P}$ is the set of literals derived from P

- for example, a labelling could consist of (a relevant subset of) the postconditions of actions

Boat race reward machine



Specifying reward machines

- reward machines can be computed from task specifications expressed in a range of property specification languages, including regular expressions and goal specification languages
- in particular, they can be computed from task specifications expressed in LTL and LTL_f (LTL on finite traces)
- any optimal policy learned using a RM expressed in LTL_f is guaranteed to satisfy the LTL_f specification (if it is possible to satisfy it at all)

Boat race specification in LTL

$$G(\texttt{up} \rightarrow X((\neg \texttt{up}) U\ \texttt{right}) \wedge$$

$$G(\texttt{right} \rightarrow X((\neg \texttt{right}) U\ \texttt{down}) \wedge$$

$$G(\texttt{down} \rightarrow X((\neg \texttt{down}) U\ \texttt{left}) \wedge$$

$$G(\texttt{left} \rightarrow X((\neg \texttt{left}) U\ \texttt{up})$$

Learning with reward machines

- agent cannot “look ahead” in the reward machine, but gains information that, e.g., a step in the task has been completed
- RMs also allow the specification of non-Markovian rewards (rewards based on the history of actions), e.g., “bring coffee exactly twice”, which is difficult to do with classic RL
- downside: in the worst case, the reward machine is double exponential in the size of the LTL formula

MDPRM

Definition (MDPRM, Toro Icarte et al. (2020))

A Markov decision process with a Reward Machine (MDPRM) is tuple $T = \langle S, s_0, A, p, \gamma, \mathcal{PV}, L, U, u_I, \delta_R, r_R \rangle$ where:

- S, s_0, A, p and γ are defined as in an MDP;
- $L : S \times Act \times S \to 2^{\overline{\mathcal{PV}}}$;
- U, u_I, δ_R, r_R are defined as in a reward machine.

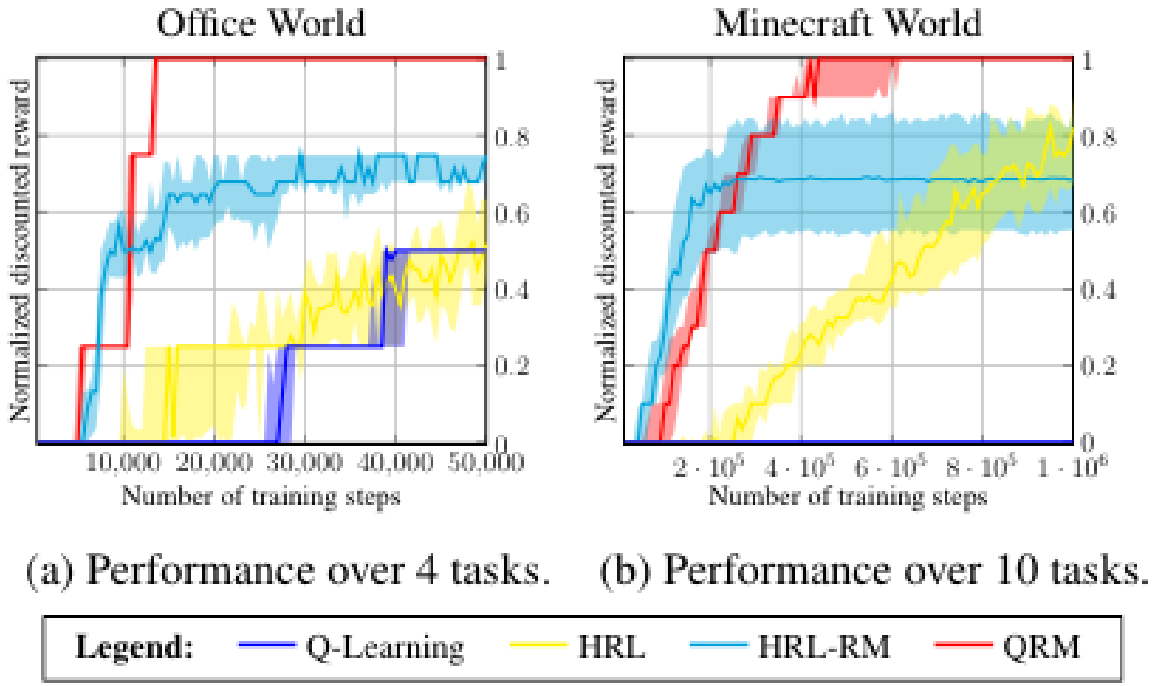
Learning in MDPRMs

- an MDPRM can be thought of as defining an MDP where the set of states is the cross product $S \times U$ of the environment states and the reward machine states
- any RL algorithm can be used to learn a policy $\pi(a \mid s, u)$ including tabular RL methods and deep RL methods.
- if the RL algorithm is guaranteed to converge to optimal policies, then it will also find optimal policies for the MDPRM
- however, since the reward machine is double exponential in the size of the LTL formula, the state space of the MDPRM is also double exponential

Sample efficiency

- even when an RL agent can learn a policy, it may take a very large number of steps (samples) to do so
- rewards are often *sparse* (very sparse in the case of terminal rewards)
- e.g., learning to play Atari games from pixels requires tens of millions of steps, which is impractical in many applications such as robotics
- in addition to ensuring that the policy learned satisfies the LTL property, reward machines can also be more sample efficient than classical RL approaches

Example: sample efficiency



From Toro Icarte et al. Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning. ICML 2019

Specifying norms using reward machines

Norms

- agents operating in a multi-agent environment often need to *coordinate* their activities with other agents in order to achieve their goals
- for example, two vehicles approaching an intersection must decide who has right of way
- in human societies, such coordination is often effected using norms — patterns of desired (or undesired) behaviour
- we focus on norms to allow effective coordination between agents in multi-party interactions, e.g., right of way, turn taking, etc.

Learning norms

- most of the work on normative multi-agent systems has been in the context of programmed agents
- there has been relatively little work on how norms can be implemented in agents developed using machine learning techniques.
- there has been work on how norms can emerge or be “learned” by a society of interacting agents; however our concern is how an existing norm, e.g., one already is use by humans, can be implemented in a learning agent
- how can the rule-like representations typically used for norms be incorporated into a reward signal for a reinforcement learning agent?
- key idea: represent the norm as a reward machine

Norms as reward machines

- reward machines are used to represent norms necessary for coordination *independently* of the agents' individual tasks and their associated reward functions
- the role of the reward machine is to ensure that the way the agent learns to perform its particular task is consistent with the norm
- for example, a norm specifying right of way is independent of the particular goals of the agents, or where they will go next, etc.
- agents can be trained independently in a decentralised manner to follow the norm

Conditional obligations with deadlines

Definition (Conditional obligation)

Let *cond*, φ, *d* be boolean combinations of propositional variables from $\overline{P}$. A *conditional obligation* is represented by a tuple $(cond, \varphi, d)$.

- conditional norms are triggered (detached) in states satisfying *cond*
- a detached norm is satisfied if the behaviour of the agent(s) brings about φ before a state in which the deadline condition *d* is true, otherwise it is violated
- norms are evaluated on sequences of states from the MDPRM

From norms to reward machines

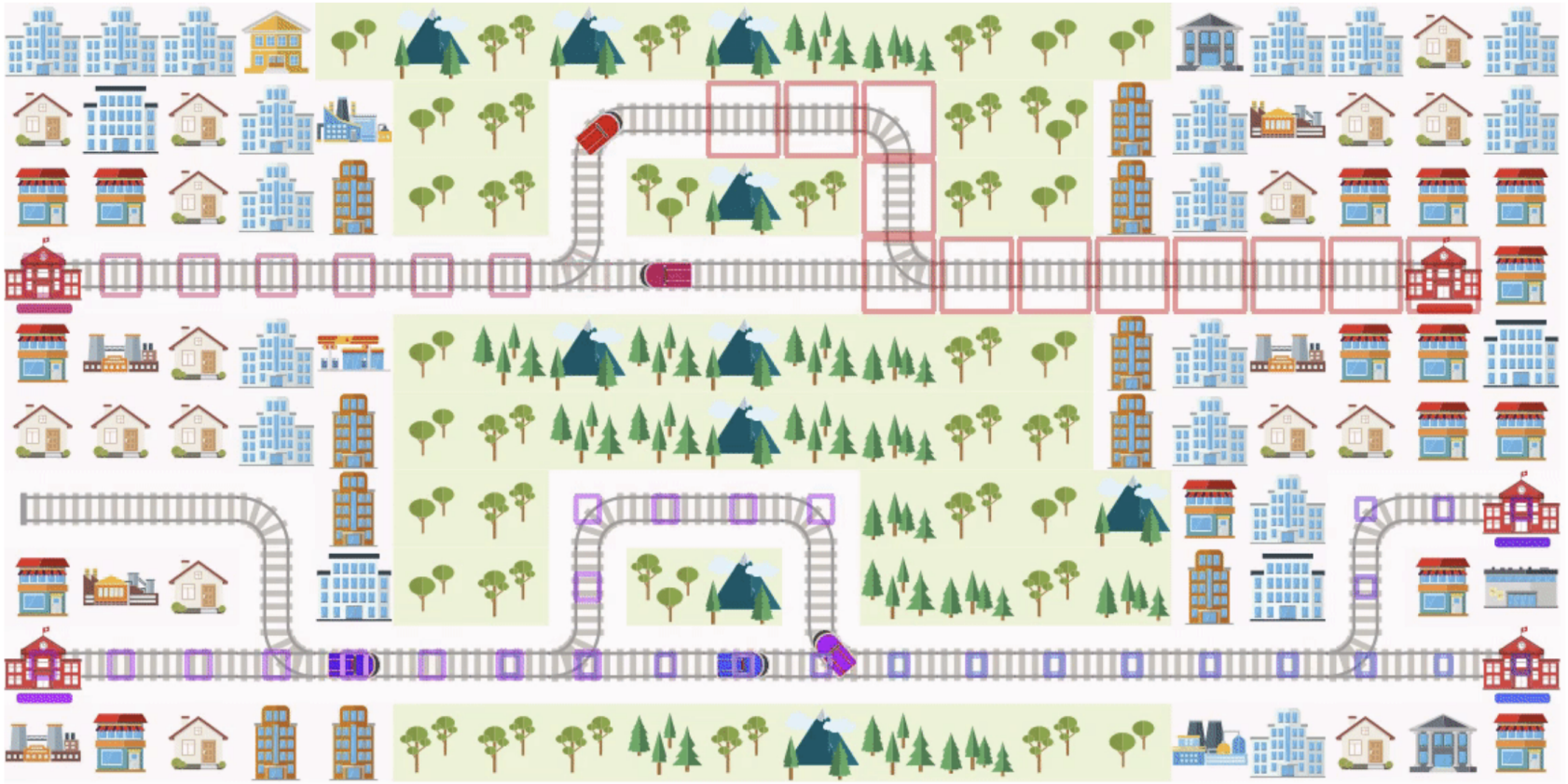
- conditional obligations with deadlines can be re-expressed in a fragment of LTL
- LTL specification can be translated into a reward machine in a straightforward way using reactive synthesis techniques

Example: Flatland

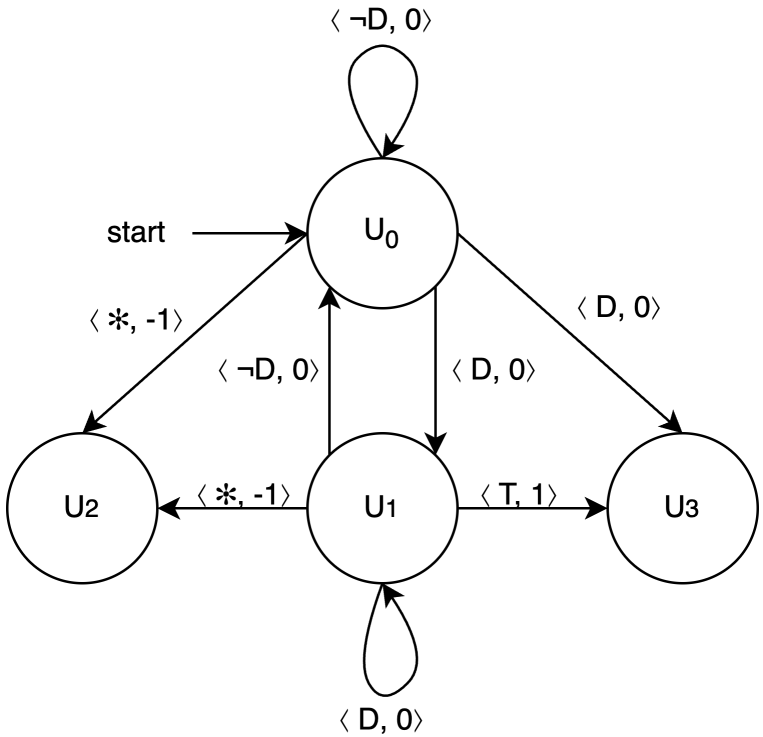
- Flatland is a railway simulation grid world where the agents' (trains) objective is to get to their destination as soon as possible
- agents move along tracks connected by switches and can take three actions: move forward, turn left and turn right.
- if two agents moving in opposite directions enter the same stretch of track a 'conflict' occurs
- to ensure coordination agents should follow a norm which intuitively states "when a potential conflict is detected, one of the agents should turn onto another track which still allows it to reach its goal before the conflict occurs"

Flatland Challenge (flatland.aicrowd.com)

Example: Flatland



Example: “avoid conflicts” reward machine



- u_0, u_1, u_2 and u_3 are the states of the reward machine
- $\Sigma = \{D, *, T\}$ is the set of environment events, where event D occurs when the agent detects a conflict in the future, $*$ occurs when the agent gets stuck in a conflict, and T occurs when the agent reaches its target

References

- Sutton & Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2020
- Leike et al. AI Safety Gridworlds. arXiv:1711.09883
- Camacho et al. LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning. IJCAI 2019: 6065-6073
- Toro Icarte et al. Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning JAIR Vol. 73, 2022, 173–208
- De Giacomo & Vardi. Synthesis for LDL and LTL on Finite Traces. IJCAI 2015: 1558–1564
- Toro Icarte et al. Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning. ICML 2019