Journal of Artificial Intelligence Research 80 (2024) 1099-1137 Submitted 7/2022; published 7/2024

Symbolic Task Inference in Deep Reinforcement Learning

Hosein Hasanbeig HOSEIN.HASANBEIG@MICROSOFT.COM
Microsoft Research

Natasha Yogananda Jeppu NATASHA.JEPPU@GMAIL.COM
Department of Computer Science, University of Oxford

Alessandro Abate ALESSANDRO.ABATE@CS.OX.AC.UK
Department of Computer Science, University of Oxford

Tom Melham TOM.MELHAM@CS.OX.AC.UK
Department of Computer Science, University of Oxford

Daniel Kroening DANIEL.KROENING@MAGD.OX.AC.UK
Amazon

Abstract

This paper proposes DeepSynth, a method for effective training of deep reinforcement learning agents when the reward is sparse or non-Markovian, but at the same time progress towards the reward requires achieving an *unknown* sequence of high-level objectives. Our method employs a novel algorithm for synthesis of compact finite state automata to uncover this sequential structure automatically. We synthesise a human-interpretable automaton from trace data collected by exploring the environment. The state space of the environment is then enriched with the synthesised automaton, so that the generation of a control policy by deep reinforcement learning is guided by the discovered structure encoded in the automaton. The proposed approach is able to cope with both high-dimensional, low-level features and unknown sparse or non-Markovian rewards. We have evaluated DeepSynth's performance in a set of experiments that includes the Atari game *Montezuma's Revenge*, known to be challenging. Compared to approaches that rely solely on deep reinforcement learning, we obtain a reduction of two orders of magnitude in the iterations required for policy synthesis, and a significant improvement in scalability.

1. Introduction

Reinforcement Learning (RL) is the key enabling technique for a variety of applications of artificial intelligence, including advanced robotics (Polydoros & Nalpantidis, 2017), resource and traffic management (Mao et al., 2016; Sadigh et al., 2014), drone control (Abbeel et al., 2007), and chemical engineering (Zhou et al., 2017). While RL is a general learning approach, many advances in the last decade have been achieved using specific instances that employ deep neural networks to synthesise approximate optimal policies. A deep RL algorithm, AlphaGo, by Silver et al. (2016), played moves in the game of Go that were initially considered glitches by human experts, but secured victory against the world champion. Similarly, AlphaStar, by Vinyals et al. (2019), was able to defeat the world's best players at the real-time strategy game StarCraft II, and to reach top 0.2% in scoreboards with an "unimaginably unusual" playing style.

While deep RL can autonomously solve problems in many complex environments, tasks that feature sparse, non-Markovian rewards or other long-term sequential structures are often difficult or impossible to solve by unaided learning methods. A well-known example is the Atari game *Montezuma's Revenge*, in which Deep Q-Network (DQN) failed to score (Mnih et al., 2015). Interestingly, Montezuma's Revenge and other hard problems often require learning a strategy that can accomplish, possibly in a specific sequence, a set of high-level objectives to obtain the reward. These objectives can often be identified with passing through designated and semantically distinguished states of the system. This insight can be leveraged to obtain a manageable, high-level model of the system's behaviour and its dynamics.

Contributions In this paper we extend upon DeepSynth (Hasanbeig et al., 2021a), a new algorithm that automatically infers unknown sequential dependencies of a reward on high-level objectives and exploits this to guide a deep RL agent when the reward signal is history-dependent and significantly delayed, as is the case, among others, in Montezuma's Revenge. We assume that these sequential dependencies have a *regular* nature, in formal language theory sense (Gulwani, 2012). The identification of dependency on achieving a sequence of high-level objectives is the key to breaking down a complex task into a series of less sparse Markovian ones. In our work, we use automata expressed in terms of high-level objectives to orchestrate sequencing of low-level actions in deep RL and to guide the learning towards sparse rewards. Furthermore, the automaton representation allows a human observer to interpret the solution computed by deep RL in a high-level manner, and to gain more insight into that solution.

At the heart of DeepSynth is a *model-free* deep RL algorithm that is synchronised in a closed-loop fashion with an automaton inference algorithm, enabling our method to learn a policy that discovers and follows high-level sparse-reward structures. The synchronisation is achieved by a product construction that creates a hybrid architecture for deep RL. This architecture combines neural networks and logic to leverage the strengths of both learning and symbolic reasoning techniques to solve complex problems in RL. Neural-network-based solutions are known for their capabilities in pattern recognition, learning from raw data, and generalizing to new situations; symbolic reasoning and logic-based solutions excel in handling structured knowledge, explicit reasoning, and providing human-readable explanations. The combination of these two approaches can result in algorithms that can learn, reason, and generalize better than either approach alone.

We evaluate the performance of DeepSynth on a wide range of benchmarks with unknown sequential high-level structures. These experiments show that DeepSynth is able to automatically discover and formalise unknown, sparse, and non-Markovian high-level reward structures, and then to efficiently synthesise successful policies in a broad variety of application domains where other related approaches fail. DeepSynth represents a better integration of deep RL and formal automata synthesis than previous approaches, making learning for sparse (and potentially non-Markovian) rewards more scalable. It is worth noting that regardless of whether the reward is Markovian or not, DeepSynth can have an advantage over conventional algorithms whenever progress towards the reward requires achieving an unknown sequence of high-level objectives.

Outline The rest of this paper is organised as follows. We first review the related work in Section 2. We then provide a primer on reinforcement learning in Section 3 and an introduction to automata synthesis in Section 4. The details of DeepSynth are then presented in Section 5, where we illustrate our method with Montezuma's Revenge as a running example. Finally, Section 6 presents an evaluation of the performance of DeepSynth on a set of experiments, including Montezuma's Revenge.

2. Related Work

Our research employs formal methods to deal with sparse reward problems in RL. In the RL literature, the dependency of rewards on objectives is often tackled with *options* (Sutton & Barto, 1998), or, in general, the dependencies are structured *hierarchically*. Current approaches to Hierarchical Reinforcement Learning (HRL) very much depend on state representations and whether they are structured enough for a suitable reward signal to be effectively engineered manually. Therefore, HRL often requires detailed supervision in the form of explicitly specified high-level actions or intermediate supervisory signals (Precup, 2001; Kearns & Singh, 2002; Daniel et al., 2012; Kulkarni et al., 2016; A. Vezhnevets et al., 2016; Bacon et al., 2017). A key difference between our approach and HRL is that our method produces a modular, human-interpretable and succinct graph to represent the sequence of tasks, as opposed to complex and comparatively sample-inefficient structures, e.g., RNNs.

The closest line of work to ours, which aims to avoid HRL requirements, are model-based (Fu & Topcu, 2014; Sadigh et al., 2014; Littman et al., 2017; Fulton & Platzer, 2018; Cai et al., 2021; Brunke et al., 2021) or model-free RL approaches that constrain the agent with a temporal logic property (Hasanbeig et al., 2018; Toro Icarte et al., 2018; Camacho et al., 2019; Hasanbeig, Kantaros, et al., 2019; Yuan et al., 2019; De Giacomo et al., 2019, 2020; Hasanbeig, Abate, & Kroening, 2019; Hasanbeig, Kroening, & Abate, 2023; Hasanbeig, Abate, & Kroening, 2020; Cai et al., 2021; Hasanbeig, Kroening, & Abate, 2020a, 2020b; Lavaei et al., 2020; Cai & Vasile, 2021; Jiang et al., 2021; Giacobbe et al., 2021; Alur et al., 2022; Hasanbeig et al., 2022; Mitta et al., 2022; Nejati et al., 2023; Wang et al., 2023; Mitta et al., 2024). These approaches are either limited to finite-state systems, or more importantly require the temporal logic formula to be known a priori. The latter assumption is relaxed in (Toro Icarte et al., 2019; Rens et al., 2020; Rens & Raskin, 2020; Furelos-Blanco et al., 2020; Gaon & Brafman, 2020; Xu et al., 2020; Abate et al., 2023), by inferring automata from exploration traces.

The automata inference by Toro Icarte et al. (2019) uses a local-search algorithm, Tabu search (Glover & Laguna, 1998). The automata inference algorithm that we employ uses SAT, where the underlying search algorithm is a backtracking search method called DPLL (Davis & Putnam, 1960). In comparison with Tabu search, the DPLL algorithm is complete and explores the entire search space efficiently (Cook & Mitchell, 1996), producing more accurate representations of the trace sets. A detailed comparison of the two approaches is provided in Section 6.5. The Answer Set Programming (ASP) based algorithm used to learn automata by Furelos-Blanco et al. (2020) also uses DPLL but assumes a known upper bound for the maximum finite distance between automaton states. We further relax this restriction and assume that the task and its automaton are entirely unknown.

A classic automata learning technique is the L* algorithm by Angluin (1987). This is used to infer automata in (Rens et al., 2020; Rens & Raskin, 2020; Gaon & Brafman, 2020; Chockler et al., 2020). It employs a series of equivalence and membership queries from an oracle, the results of which are used to construct the automaton. The absence of an oracle in our setting prevents the use of L* in our method.

Another common approach for synthesising automata from traces is *state-merge* (Biermann & Feldman, 1972). Variants of the state-merge algorithm, e.g., Evidence-Driven State Merge (EDSM) by Lang et al. (1998), use both positive and negative instances of behaviour to determine equivalence of states to be merged based on statistical evidence. To avoid over-generalisation in the absence of labelled data, the EDSM algorithm was improved to incorporate inherent temporal behaviour (Walkinshaw et al., 2007; Walkinshaw & Bogdanov, 2008), which needs to be known a priori. State-merge and some of its variants (Lang et al., 1998; Walkinshaw et al., 2007), however, do not always produce the most succinct automaton but generate an approximation that conforms to the trace (Ulyantsev et al., 2018). The comparative succinctness of our inferred automaton allows DeepSynth to be applied to large high-dimensional problems, including Montezuma's Revenge. A detailed comparison of these approaches can be found in Section 6.4.

A number of approaches combine SAT with state-merge to generate minimal automata from traces (Ulyantsev & Tsarev, 2011; Heule & Verwer, 2013; Ulyantsev et al., 2018; Buzhinsky & Vyatkin, 2017; Buzhinsky & Vyatkin, 2017). A similar SAT based algorithm is employed by (Xu et al., 2020; Neider et al., 2021; Xu et al., 2021) to generate reward machines. Although this approach generates succinct automata that accurately capture a rewarding sequence of events, it is not ideal for hard exploration problems such as Montezuma's Revenge where reaching a rewarding state, e.g., collecting the key, requires the agent to follow a sequence of non-rewarding steps that are difficult to discover via exploration. The automata learning algorithm we use is able to capture these non-rewarding sequences and leverage them to guide exploration towards the rewarding states. Inferred automata have been also used to learn strategies for infinite two-person games in which strategies are a function of previously visited states. The construction of *chain automata* for these games provides a means to implement memory-less strategies (Krishnan et al., 1995). However, these chain automata have a disproportionally large number of states when compared to the size of the actual automaton the game is played on.

Further related work is *policy sketching* by Andreas et al. (2017), which learns feasible tasks first and then stitches them together to accomplish a complex task. The key difference to our work is that their method assumes policy sketches, i.e., temporal instructions, to be available to the agent. Memarian et al. (2021) proposed a self-supervised method to infer and encode sparse reward signals as a neural network. Furthermore, it is worth noting that learning optimal policies for infinite-horizon temporal instructions in a probably approximately correct Markov decision process framework is theoretically intractable (Yang et al., 2021). There is also recent work on learning underlying non-Markovian objectives when an optimal policy or human demonstration is available (Koul et al., 2019; Ringstrom et al., 2020; Memarian et al., 2020). A recent, statistical approach to learn Task Automata is presented by Abate et al. (2023) which, however, is not directly comparable to ours, in view of its different nature and the absence of an interleaved policy synthesis part. Recently, the zero-shot planning and logical reasoning capabilities of large language models (LLMs), specifically within the context of latent relational automata underlying reinforcement learning tasks (Momennejad et al., 2017), have been investigated by Momennejad et al. (2024). The authors highlight the importance of symbolic task inference in control policy synthesis using

LLMs as it enables the extraction and formulation of task goals and constraints from natural language descriptions. As shown by Hasanbeig, Sharma, et al. (2023), this integration, which in its simplest form can be achieved via iterative in-context-learning, aids in aligning LLMs with specific tasks, enhancing their reasoning capabilities to generate context-aware actions.

3. Background on Reinforcement Learning

We consider a conventional RL setup, consisting of an agent interacting with an environment, which is modelled as a *black box* Markov Decision Process (MDP), defined next.

Definition 1 (MDP) *The tuple $\mathfrak{M} = (\mathcal{S}, \mathcal{A}, s_0, P, \Sigma, L, R)$ is an MDP over a set of continuous states $\mathcal{S}$, where $\mathcal{A}$ is a finite set of actions and $s_0 \in \mathcal{S}$ is the initial state. $P : \mathcal{B}(\mathcal{S}) \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a Borel-measurable conditional transition kernel that assigns to any pair of state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$ a probability measure $P(\cdot|s, a)$ on the Borel space $(\mathcal{S}, \mathcal{B}(\mathcal{S}))$, where $\mathcal{B}(\mathcal{S})$ is the Borel σ-algebra on the state space (Bertsekas & Shreve, 2004). The set Σ is called the* vocabulary *and it is a finite set of atomic propositions. Ths labelling function $L : \mathcal{S} \to 2^{\Sigma}$ that assigns to each state $s \in \mathcal{S}$ a set of atomic propositions $L(s) \in 2^{\Sigma}$. Further, a random variable $R(s, a) \sim \Upsilon(\cdot|s, a) \in \mathcal{P}(\mathbb{R})$ is defined over the MDP $\mathfrak{M}$, to represent the Markovian reward obtained when action a is taken in a given state s, where $\mathcal{P}(\mathbb{R})$ is the set of probability distributions on subsets of $\mathbb{R}$, and Υ is the reward distribution. A possible realisation of R at time step n is denoted by r_n.*

Definition 2 (Path) *In an MDP $\mathfrak{M}$, an infinite path ρ starting at s_0 is an infinite sequence of state transitions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ ...$ such that every transition $s_i \xrightarrow{a_i} s_{i+1}$ is possible in $\mathfrak{M}$. I.e., s_{i+1} belongs to the smallest Borel set B such that $P(B|s_i, a_i) = 1$. Similarly, a finite path is a finite sequence of state transitions $\rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$. The set of infinite paths is $(\mathcal{S} \times \mathcal{A})^{\omega}$ and the set of finite paths is $(\mathcal{S} \times \mathcal{A})^* \times \mathcal{S}$.*

Similar to a Markovian reward, a non-Markovian reward $\widehat{R} : (\mathcal{S} \times \mathcal{A})^* \times \mathcal{S} \to \mathbb{R}$ is a mapping from the set of finite paths to real numbers, and a realisation of $\widehat{R}$ at time step n is denoted by $\widehat{r}_n$. At each state $s \in \mathcal{S}$, an agent action is determined by a policy π, which is a mapping from states to a probability distribution over the actions. That is, $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$.

Definition 3 (Expected Discounted Return) *For a policy π on an MDP $\mathfrak{M}$, the expected discounted return for a Markovian reward R is defined as (Sutton & Barto, 1998):*

$$V^{\pi}(s) = \mathbb{E}^{\pi}[\sum_{n=0}^{\infty} \gamma^n \ r_n | s_0 = s], \tag{1}$$

where $\mathbb{E}^{\pi}[\cdot]$ denotes the expected value given that the agent follows policy π, and $\gamma \in [0, 1)$ ($\gamma \in [0, 1]$ when episodic) is a discount factor.

The expected return is also known as the *value function* in the literature. For any state-action pair (s, a) we can also define an action-value function that assigns a quantitative measure $Q^{\pi} : S \times A \to \mathbb{R}$ as follows:

$$Q^{\pi}(s, a) = \mathbb{E}^{\pi}[\sum_{n=0}^{\infty} \gamma^n \ r_n | s_0 = s, a_0 = a]. \tag{2}$$

Q-Learning (QL) by Watkins and Dayan (1992) uses the action-value function and updates state-action pair values upon visitation. QL is *off-policy*, which means that π has no effect on the convergence of the Q-function, as long as every state-action pair is visited infinitely many times. Thus, for simplicity, we may drop the superscript π in (2), and update the Q-function as

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a) + \gamma \max_{a' \in \mathcal{A}}(Q(s',a')) - Q(s,a)], \tag{3}$$

where $0 < \alpha \leq 1$ is the learning rate, γ is the discount factor, and s' is the state reached after performing action a. The learning rate and discount factor in general can be state-dependent. Under mild assumptions, QL converges to a unique limit Q^*, as long as every state action pair is visited infinitely many times (Watkins & Dayan, 1992). Once QL converges, an optimal policy can be distilled from the action-value function:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \ Q^*(s,a),$$

where π^* is the same optimal policy that can be alternatively generated with Bellman iterations (Bertsekas & Tsitsiklis, 1996) if the MDP was fully known, maximising the expected return in (1) at any given state. Thus, the main goal is to synthesise π^* when the MDP is essentially a *black box*. We denote a non-Markovian optimal policy by $\widehat{\pi}^*$, which optimises a memory-dependent Q-function $\widehat{Q}^*$.

In many problems, the MDP can have a continuous or large state space, and thus the recursion in (3) has to be approximated by parameterising Q using a parameter set θ^Q. The parameters are updated by minimizing the following loss function (Riedmiller, 2005):

$$\mathfrak{L}(\theta^Q) = \mathbb{E}_{s \sim pr^\beta}[(Q(s,a|\theta^Q) - y)^2], \tag{4}$$

where pr^β is the probability distribution of state visit over $\mathcal{S}$ under an arbitrary stochastic policy β, and

$$y = R(s,a) + \gamma \max_{a'} Q(s',a'|\theta^Q).$$

The function Q can then be approximated via a deep neural network architecture where the parameter set θ^Q represents the weights of the neural network.

4. Background on Automata Synthesis

This section describes the fundamentals of the algorithm used for automatic symbolic inference of unknown high-level sequential structures as automata. We use an automata synthesis algorithm that extracts information from trace sequences over finite paths (Definition 2) in order to construct a succinct automaton that represents the behaviour exemplified by these traces. Here, a trace is defined as follows:

Definition 4 (Trace) *In an MDP $\mathfrak{M}$, and over a finite path $\rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$, a trace σ is defined as a sequence of labels $\sigma = v_1, v_2, \ldots, v_n$, where $v_i = L(s_i)$ is referred to as a trace event.*

The automata synthesis algorithm used in this work (Jeppu et al., 2020) is an instance of the general *synthesis from examples* approach (Gulwani, 2012). It is a scalable method for learning a finite-state automaton from *positive* trace data only. A positive trace, in the literature, means a sequence of labels exhibiting behaviour that can be produced by the underlying system; a *negative* trace refers to a sequence of labels that can never be produced by the underlying system. The algorithm aims to produce abstract, concise models. It takes as input a set of positive traces and generates an automaton that conforms to the input trace set.

The automaton being constructed is represented symbolically as a set of *transition templates*. Each transition template is a triple (q, p, q') comprising symbolic variables q, p and q' for the state from which the transition occurs, the corresponding transition predicate and the next state, respectively. Boolean constraints defined over these symbolic variables encode the sequence of trace events in each input trace as transition sequences in the automaton. An upper bound on the number of automaton states N is enforced by constraining the symbolic variables q and q' to take values between 1 and N. The Boolean formula obtained as the conjunction of all constraints is then fed to a SAT solver, which looks for a satisfying assignment.

Starting with $N = 1$, the algorithm searches for a satisfying assignment, incrementing N by one each time the search fails. This ensures that the smallest automaton conforming to the input trace is generated. We note, however, that a trivial solution to the encoded SAT problem is a single state automaton that accepts all traces defined over the set of trace events. Our automaton synthesis algorithm generates models using only positive trace samples, and it therefore runs the risk of overgeneralisation. In order to address overgeneralisation and consequently to control the precision of the learned automaton, the algorithm introduces a tuneable parameter l. Once a candidate automaton is generated, the algorithm performs a compliance check of the automaton against the input trace to eliminate any trace event sequences of length l that are admitted by the generated model but do not appear in the input trace. The identified trace event sequences are used to encode blocking constraints on the automaton. These are constraints that restrict the automaton from representing the corresponding sequential behaviour. The search is then repeated. A higher value for l yields more exact representations of the trace. Further details are provided by Jeppu et al. (2020).

For long traces, the Boolean formula encoding all automaton constraints becomes very large, thereby increasing algorithm runtime. To enable the algorithm to scale to long traces, the automaton synthesis method introduces trace segmentation as an optimisation. It leverages the presence of patterns in a trace to reduce a large SAT instance to multiple smaller SAT instances with manageable runtime. Here, a pattern or trace segment is any sequence of labels that are consecutive in the trace. Each such pattern exemplifies sequential behaviour that we wish to capture in the automaton.

With trace segmentation, the constraints for automaton construction are relaxed such that the algorithm learns an automaton that captures all label correlations exemplified by patterns of a specified length w, rather than the entire trace. To this end, the automaton synthesis algorithm uses a sliding window to extract all unique segments of length w from the trace. The segments are used to define Boolean constraints as before, such that the corresponding sequential behaviour is captured by the learned automaton. Experimental evidence indicates a linear growth in runtime for increasing trace length with the optimisation, as compared to exponential growth without the optimisation (Jeppu et al., 2020).

Note that with trace segmentation, the constraints on the automaton do not suffice to guarantee that the generated automaton accepts the input trace. Therefore, once a model is generated the algorithm checks if the learned automaton accepts the input trace. If the check fails, missing trace data is *incrementally* added to refine the generated model, until the check passes. To this end, the algorithm identifies the smallest trace prefix from the input trace set that is not accepted by the learned automaton. Note that the trace prefix could be longer than w. The prefix is used to encode additional constraints on the automaton such that the sequential behaviours exemplified by the trace prefix are additionally represented by the automaton. The constraints ensure that in each iteration of this incremental learning approach, the algorithm generates an automaton whose behaviours are a superset of the behaviours represented by the automaton learned in the previous iteration. The above incremental learning approach can also be applied to the setting where given an automaton and a set of positive traces, we wish to augment the automaton with behaviours exemplified by the traces.

Given a set of traces, our incremental, SAT-based algorithm learns succinct models that are guaranteed to be correct using only trace data. Here, an automaton is said to be correct if it accepts all input traces. The values of the parameters w and l do not affect the correctness of the generated automaton, but merely affect algorithm runtime and the succinctness of the generated models. In this work, we use an open source implementation (Jeppu, 2020) of the automata synthesis algorithm to infer unknown high-level dependencies of the reward. We use the default tool setting of $w = 3$ and $l = 2$ to generate our models. Further details on trace generation and the integration of automaton synthesis with deep RL are given in the next section.

5. DeepSynth

5.1 Overview of the Algorithm and Running Example

A schematic of the DeepSynth algorithm is provided in Figure 1. The algorithm comprises three steps, and we will give details of each step in a separate subsection. We will illustrate the workings of each step using a running example, the first level of Montezuma's Revenge (Figure 2).

Unlike other Atari games in which the primary goal is limited to avoiding obstacles or collecting items in no particular order, Montezuma's Revenge requires the agent to perform a long, complex sequence of actions before receiving any reward (Bellemare et al., 2013). The agent must find a key and open either door in Figure 2.a. To this end, the agent has to climb down the middle ladder, jump on the rope, climb down the ladder on the right, and jump over a skull to reach the key. The reward given by the Atari emulator for collecting the key is 100 and the reward for opening one of the doors is another 300. Owing to the sparsity of the rewards, the existing deep RL algorithms either fail to learn a policy that can even reach the key, e.g., DQN (Mnih et al., 2015), or the learning process is computationally heavy and sample inefficient, e.g., FeUdal by A. S. Vezhnevets et al. (2017) and Go-Explore by Ecoffet et al. (2021).

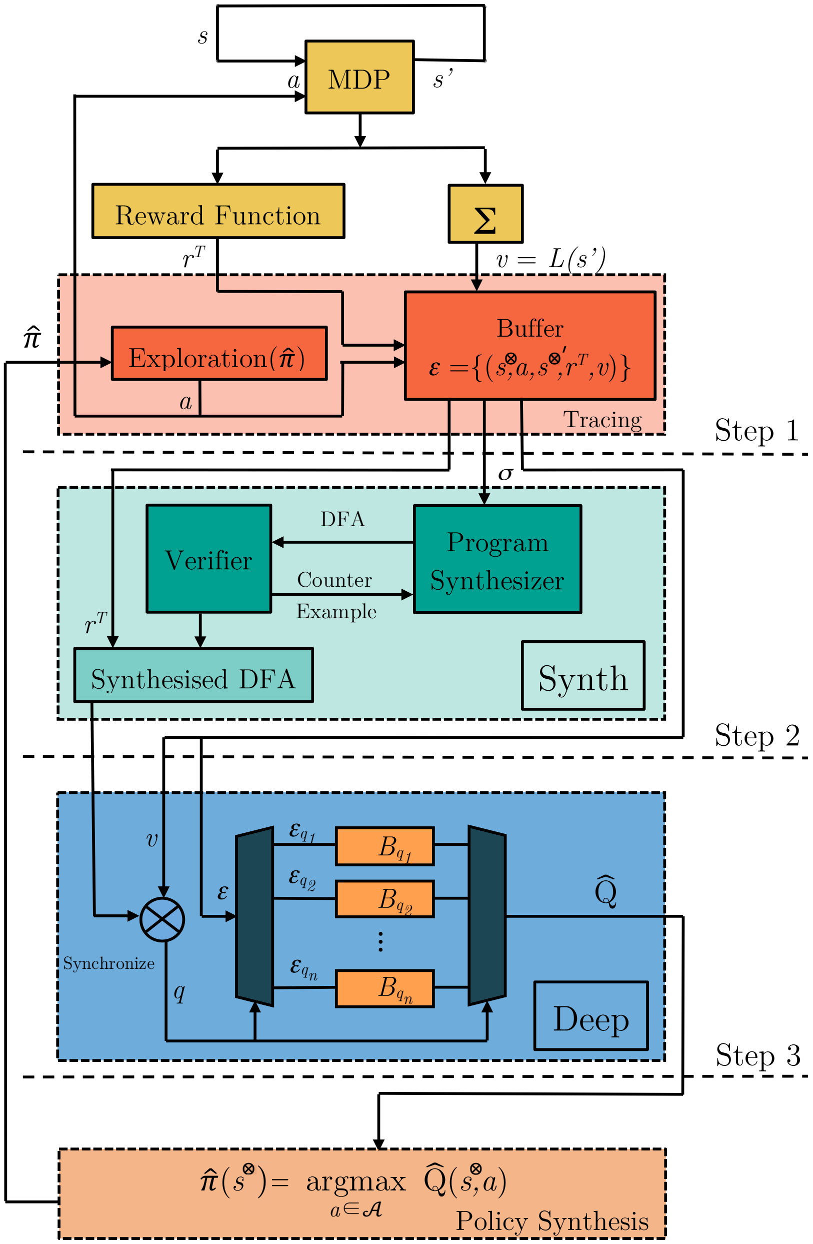
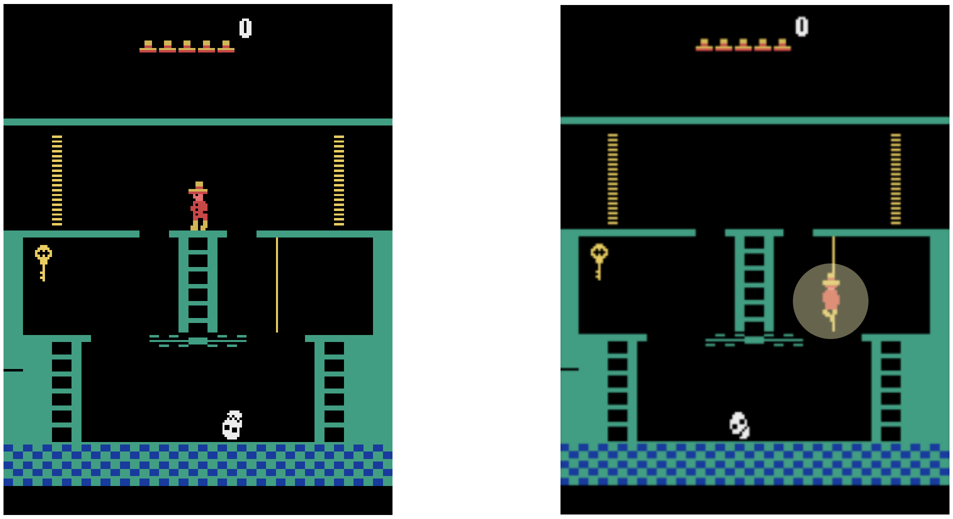


Figure 1: The DeepSynth Algorithm



(a)

(b)

Figure 2: (a) the first level of Atari 2600 Montezuma's Revenge; (b) pixel overlap of two segmented objects.

Existing techniques to solve this problem mostly hinge on intrinsic motivation and object-driven guidance. Unsupervised object detection (or unsupervised semantic segmentation) from raw image input has seen substantial progress in recent years, and became comparable to its supervised counterpart (Liu et al., 2019; Ji et al., 2019; Hwang et al., 2019; Zheng & Yang, 2021). In this work, we assume that an off-the-shelf image segmentation or object detection algorithm can provide plausible object candidates that are semantically distinguishable, e.g., Liu et al. (2019). Exploring image segmentation or object detection methods is very domain-specific and is outside of the scope of this work.

The key to solving a complex task of the type exemplified by Montezuma's Revenge is to find the *semantic correlation* between the objects in the scene. When a human player tries to solve this game, the semantic correlations, such as "keys open doors", are partially known a priori and the player's behaviour is driven by these known correlations when exploring unknown objects. This drive to explore unknown objects has been a subject of study in psychology, where animals and humans seem to have general motivations (often referred to as intrinsic motivations) that push them to explore and manipulate their environment, encouraging curiosity and cognitive growth (Berlyne, 1960; Csikszentmihalyi, 1990; Ryan & Deci, 2000).

As explained later, DeepSynth encodes these correlations as an automaton, which is an intuitive and modular structure, and guides the exploration so that by exploiting learned (or known) correlations, previously unknown correlations are gradually discovered. This exploration scheme imitates cognitive growth in biological organisms, but in a formal and explainable way. The exploration is driven by an intrinsic motivation to discover as many objects as possible in order to find the optimal sequence of extrinsically-rewarding high-level objectives. To showcase the full potential of DeepSynth, in all the experiments and examples in this paper, we assume that the agent is unaware of any semantic correlation. The agent starts with no prior knowledge of the sparse reward task or the correlation of the high-level objects.

5.2 Tracing (Step 1 in Figure 1)

The purpose of tracing is to gather execution sequences from the model, and associate a reward to them. The task is unknown initially and the extrinsic reward is extremely sparse (and potentially non-Markovian[1]). The agent receives a reward $\widehat{R} : (\mathcal{S} \times \mathcal{A})^* \times \mathcal{S} \rightarrow \mathbb{R}$ only when a correct sequence of state-action pairs and their associated object correlations are visited. (Note that a Markovian reward is a special case here.) In order to guide the agent to find the optimal sequence, DeepSynth uses the following reward transformation:

$$r^T = \widehat{r} + \mu \, r^i, \tag{5}$$

where $\widehat{r}$ is the extrinsic reward (which can be Markovian), $\mu > 0$ is a positive regulatory coefficient, and r^i is the intrinsic reward. The role of the intrinsic reward is to guide the exploration and also to drive the exploration towards the discovery of unknown object correlations. The underlying mechanism of intrinsic rewards depends on the inferred automaton and is explained in detail later.

Using the labelling function (Definition 1), Tracing (Step 1) records the label trace $L(s_i)L(s_{i+1})\ldots$ as the agent explores the environment. All transitions with their corresponding labels and rewards are stored and the set of traces is denoted by $\mathcal{T}$. The tracing scheme is the Tracing box in Figure 1.

Application to Running Example The only extrinsic rewards in Montezuma's Revenge are the reward for reaching the key $\widehat{r}_{key}$ and for reaching one of the doors $\widehat{r}_{door}$. As we will argue in Section 6, the lack of intrinsic motivation prevents other methods, e.g., (Toro Icarte et al., 2019; Rens et al., 2020; Gaon & Brafman, 2020; Xu et al., 2020), to succeed in extremely-spare reward, high-dimensional and large problems such as Montezuma's Revenge.

In Montezuma's Revenge, the agent observes raw pixel images, and the input state is a stack of four consecutive frames $84 \times 84 \times 4$ that are preprocessed to reduce the dimensionality (Mnih et al., 2015). As mentioned above, we use unsupervised object detection on the raw image output from the simulator. Let us write Σ for the set of detected objects. Note that the semantics of the names for individual objects is of no relevance to the algorithm and Σ can thus contain any distinct identifiers, e.g., $\Sigma = \{\texttt{obj}_1, \texttt{obj}_2, \ldots\}$. But for the sake of exposition we name the objects according to their appearance in Figure 2.a, i.e., $\Sigma = \{\texttt{red\_character}, \texttt{middle\_ladder}, \texttt{rope}, \texttt{right\_ladder}, \texttt{left\_ladder}, \texttt{key}, \texttt{door}\}$. Note that there can be any number of detected objects, as long as the input image is segmented into enough objects whose correlation can guide the agent to achieve the task. We

[1] Note that in the case of Montezuma's Revenge, it remains uncertain whether the reward system is truly Markovian or not, primarily due to the lack of access to the internal state of the game's emulator. Recall that a Markovian reward structure is dependent only on the current state and action, independent of previous states or actions, while a non-Markovian reward structure depends on sequences of states or actions, not solely on the current state or action. As Montezuma's Revenge is frequently used as a benchmark for evaluating the performance of AI agents, it is crucial to acknowledge the potential non-Markovian nature of its reward and the implications it may have on the development and assessment of reinforcement learning techniques.

remark that agent interaction with the skull results in death and episode termination. Thus, reading the label `skull` or falling from the ladders, or any action that results in terminating the game run, is not useful for driving exploration towards rewarding states. Hence, for the sake of simplicity we have omitted the label `skull` from Σ.

The labelling function employs the object vocabulary set Σ to detect object pixel overlap in a particular state frame. For example, if the pixels of `red_character` collide with the pixels of `rope` in any of the stacked frames, the labelling function for that particular state s is $L(s)$={`red_character`, `rope`} (Figure 2.b). In this specific example, the only moving object is the character. So for sake of succinctness, we omit the character from the label set, i.e., the above label is $L(s)$ = {`rope`}.

Given this labelling function, the Tracing step records the sequence of detected objects $L(s_i)L(s_{i+1})\ldots$ as the agent explores the game. We would like to stress that the labels considered in this work are general and not particularly dependent on object collisions. The labelling function, as per Definition 1, is a mapping from the state space to the power set of objects in the vocabulary $L : \mathcal{S} \rightarrow 2^\Sigma$ and thus, the label of a state could be the empty set or any set of objects from Σ.

5.3 Synth (Step 2 in Figure 1)

The automata synthesis algorithm described in Section 4 is used to generate an automaton that conforms to the trace sequences generated by Tracing (Step 1). Given a trace sequence $\sigma = v_1, v_2, \ldots, v_n$, the labels v_i serve as transition predicates in the generated automaton. The synthesis algorithm further constrains the construction of the automaton so that no two transitions from a given state in the generated automaton have the same predicates. The automaton obtained by the synthesis algorithm is thus deterministic.

The learned automaton follows the standard definition of a Deterministic Finite Automaton (DFA). The alphabet is $\Sigma_\mathfrak{A}$, containing symbols $v \in \Sigma_\mathfrak{A}$ given by the labelling function $L : \mathcal{S} \rightarrow 2^\Sigma$ defined earlier. Thus, for a trace sequence $\sigma = v_1, v_2, \ldots, v_n$ over a finite path $\rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ in the decision process, the symbol $v_i \in \Sigma_\mathfrak{A}$ is given by $v_i = L(s_i)$.

Definition 5 (Deterministic Finite Automaton) *A DFA $\mathfrak{A} = (\mathcal{Q}, q_0, \Sigma_\mathfrak{A}, F, \delta)$ is a 5-tuple, where $\mathcal{Q}$ is a finite set of states, $q_0 \in \mathcal{Q}$ is the initial state, $\Sigma_\mathfrak{A}$ is the alphabet, $F \subset \mathcal{Q}$ is the set of accepting states, and $\delta : \mathcal{Q} \times \Sigma_\mathfrak{A} \rightarrow \mathcal{Q}$ is the transition function.*

Let $\Sigma_\mathfrak{A}^*$ be the set of all finite words over $\Sigma_\mathfrak{A}$. A finite word $w = v_1, v_2, \ldots, v_m \in \Sigma_\mathfrak{A}^*$ is accepted by a DFA $\mathfrak{A}$ if there exists a finite run $\vartheta \in \mathcal{Q}^*$ starting from $\vartheta_0 = q_0$, where $\vartheta_{i+1} = \delta(\vartheta_i, v_{i+1})$ for $i \geq 0$ and $\vartheta_m \in F$. Given the collected traces $\mathcal{T}$ we construct a DFA using the method described in Section 4.

The generated automaton provides deep insight into the correlation of the objects detected in Step 1 and shapes the intrinsic reward. The output of this stage is a DFA, which is updated iteratively as the agent explores and gathers more trace sequences.

Application to Running Example Figure 3 illustrates the evolution of the synthesised automata for Montezuma's Revenge. Most of the deep RL approaches are able to reach the states that correspond to the DFA up to state q_4 in Figure 3 via random exploration. But

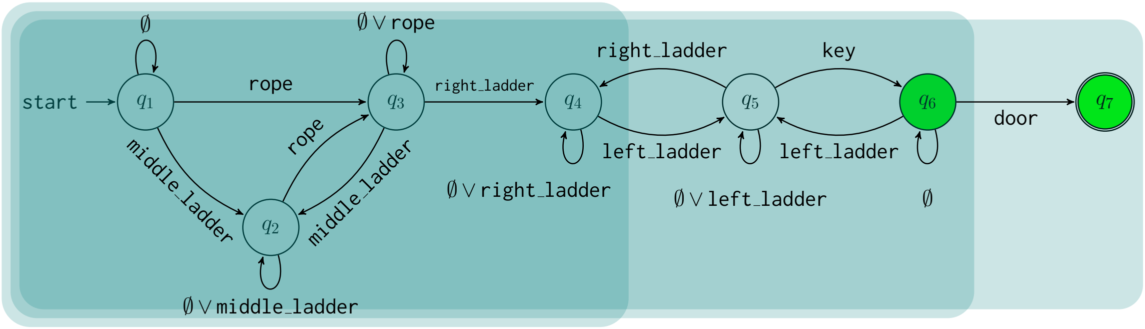


Figure 3: Illustration of the evolution of the automaton synthesised for Montezuma's Revenge (extrinsically-rewarding states are in green). The right ladder is often discovered by random exploration (state q_4). Note that the agent found a short-cut to reach the key by skipping the middle ladder and directly jumping over the rope, which is not obvious even to a human player (state q_1 to q_3). Such observations are difficult to extract from other hierarchy representations, e.g., LSTMs. The key is found with an extrinsic reward of $\widehat{r}_{\texttt{key}} = +100$ (state q_6) and the door is unlocked with an extrinsic reward of $\widehat{r}_{\texttt{door}} = +300$ (state q_7).

reaching the key and further the doors is challenging and is achieved by DeepSynth using a hierarchical curiosity-driven learning method described next.

In the following, in order to explain the core ideas underpinning the algorithm, we temporarily assume that the MDP structure and the associated transition probabilities are fully known. Later we relax these assumptions, and we stress that the algorithm can be run *model-free* over any black-box MDP environment. Specifically, we relate the black-box MDP and the automaton by synchronising them *on-the-fly* to create a new structure that breaks down a sparse and potentially non-Markovian task into a set of Markovian, history-independent sub-goals.

Definition 6 (Product MDP) *Given an MDP* $\mathfrak{M} = (\mathcal{S}, \mathcal{A}, s_0, P, \Sigma, L, R)$ *and a DFA* $\mathfrak{A} = (\mathcal{Q}, q_0, \Sigma_{\mathfrak{A}}, F, \delta)$*, the product MDP is defined as* $(\mathfrak{M} \otimes \mathfrak{A}) = \mathfrak{M}_{\mathfrak{A}} = (\mathcal{S}^{\otimes}, \mathcal{A}, s_0^{\otimes}, P^{\otimes}, \Sigma^{\otimes}, F^{\otimes}, L, R)$*, where* $\mathcal{S}^{\otimes} = \mathcal{S} \times \mathcal{Q}$*,* $s_0^{\otimes} = (s_0, q_0)$*,* $\Sigma^{\otimes} = \mathcal{Q}$*, and* $F^{\otimes} = \mathcal{S} \times F$*. The transition kernel* $P^{\otimes}$ *is such that given the current state* (s_i, q_i) *and action* a*, the new state* (s_j, q_j) *is given by* $s_j \sim P(\cdot|s_i, a)$ *and* $q_j = \delta(q_i, L(s_j))$*.*

By synchronising the MDP states with the DFA states by means of the product MDP, we can quantify the degree of satisfaction of the associated high-level task. Most importantly, as shown by Brafman et al. (2018), for any MDP $\mathfrak{M}$ with finite-horizon non-Markovian reward, there exists a Markov reward MDP $\mathfrak{M}' = (\mathcal{S}, \mathcal{A}, s_0, P, \Sigma, L, R)$ that is equivalent to $\mathfrak{M}$ such that the states of $\mathfrak{M}$ can be mapped into those of $\mathfrak{M}'$. The corresponding states yield the same transition probabilities, and corresponding traces have the same rewards. Based on this result, De Giacomo et al. (2019) showed that the product MDP $\mathfrak{M}_{\mathfrak{A}}$ is $\mathfrak{M}'$ defined above. Therefore, by synchronising the DFA with the original MDP the non-Markovian extrinsic reward becomes Markovian. Secondly, the DFA succinctly expresses the history of the state labels (i.e., traces) that have been discovered by the agent.

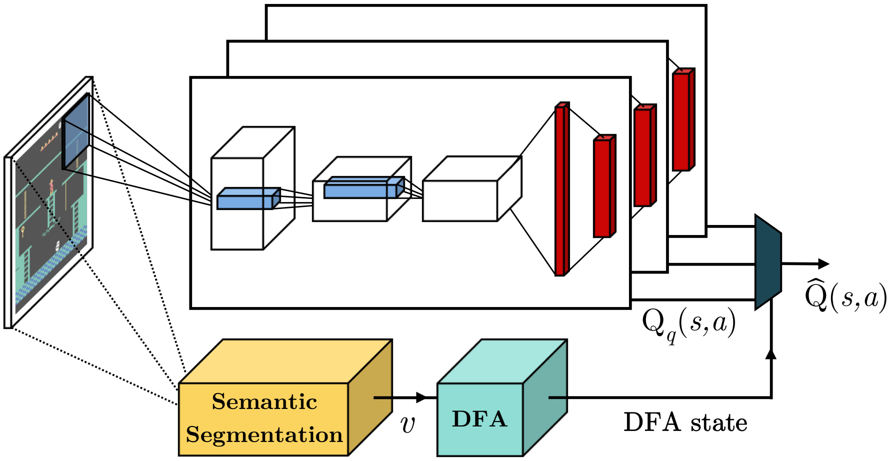


Figure 4: DeepSynth for Montezuma's Revenge: each DQN module is forced by the DFA to focus on the correlation of semantically distinct objects. The input to the first layer of the DQN modules is the input image which is convolved by 32 filters of 8×8 with stride 4 and a ReLU. The second hidden layer convolves 64 filters of 4×4 with stride 2, followed by a ReLU. This is followed by another convolutional layer that convolves 64 filters of 3×3 with stride 1 followed by a rectifier. The final hidden layer is fully connected and consists of 512 ReLUs and the output layer is a fully-connected linear layer with a single output for each action.

Note that the DFA transitions can be taken just by observing the labels of the visited states, which makes the agent aware of the automaton state without explicitly constructing the product MDP. This means that the proposed approach can run *model-free*, and as such it does not require a priori knowledge about the MDP.

5.4 Deep Temporal Neural Fitted RL (Step 3 in Figure 1)

Each state of the DFA in the synchronised product MDP divides the general sequential task so that each transition between the states represents an achievable Markovian sub-task. Thus, given a synthesised DFA $\mathfrak{A} = (\mathcal{Q}, q_0, \Sigma_{\mathfrak{A}}, F, \delta)$, we propose a hybrid architecture of $n = |\mathcal{Q}|$ separate deep RL modules (Figure 4 and box Deep in Figure 1). For each state in the DFA, there is a dedicated deep RL module, where each deep RL module is an instance of a deep RL algorithm with distinct neural networks and replay buffers. A general replay buffer $\mathcal{E}$ stores all the gathered samples as 5-tuples $\langle s^{\otimes}, a, s^{\otimes\prime}, r^T, L(s')\rangle$, where $s^{\otimes}$ is the current product state, a is the executed action, $s^{\otimes\prime}$ is the resulting product state, r^T is the total reward received after performing action a at state $s^{\otimes}$, and $L(s')$ is the label corresponding to the set of atomic propositions in Σ that hold in state s', where $s^{\otimes\prime} = (s', q')$. The modules are interconnected, in the sense that the modules act as a global *hybrid* deep RL architecture to approximate the Q-function in the product MDP. As described next, this allows the agent to jump from one sub-task to another by just switching between these modules as prescribed by the DFA.

The specifics of Step 3 depend on the input to the agent provided by the environment. For high-dimensional inputs, such as raw images, we propose an architecture that is based on deep-RL and is inspired by DQN (Mnih et al., 2015). When instead the input is low dimensional and in vector form, our algorithm is based on Neural Fitted Q-iteration. For both setups, DeepSynth synthesises a policy whose traces are accepted by the DFA and it encourages the agent, through intrinsic motivation, to explore the environment under the guidance of the DFA. More importantly, under this guidance the agent is encouraged to expand the DFA itself, as described in the following.

In the running example (Montezuma's Revenge), the Atari emulator provides the number of lives left in the game, which is used to reset $\mathcal{Q}_E \subseteq \mathcal{Q} \setminus \mathcal{N}$, where $\mathcal{Q}_E$ is the set of automaton states that are visited so far in the learning episode, and $\mathcal{N}$ is the set of non-accepting sink states in the automaton. Upon losing a life in the running example, or generally upon episode termination, $\mathcal{Q}_E$ is reset to q_0, marking the start of a new learning episode.

At each time step during the learning episode, given the constructed DFA, if a new DFA state is observed during exploration, the intrinsic reward in (5) becomes positive. Namely,

$$R^i(s^{\otimes}, a) = \begin{cases} \eta & \text{if } q' \notin \mathcal{Q}_E, \\ 0 & \text{otherwise,} \end{cases} \tag{6}$$

where η is an arbitrarily finite and positive reward that is strictly less than the extrinsic reward $\widehat{r}$, and $\mathcal{Q}_E$ is the set of automaton states that the agent has observed *in the current learning episode*. Further, once a new label that does not belong to $\Sigma_{\mathfrak{A}}$ is observed during exploration (Step 1) it is then passed to the automaton synthesis step (Step 2). The automaton synthesis algorithm then synthesises a new DFA that complies with the new label.

In the running example (Montezuma's Revenge), the agent receives a positive intrinsic reward every time it reaches the middle ladder for the first time within each learning episode. Specifically, the intrinsic reward is used to motivate exploration, expand the DFA, and to discover new objects. The overall task is initially unknown, thus *bad* high-level actions cannot be inferred solely from the labels. Note that whenever the agent is intrinsically rewarded towards a bad behaviour, the contribution from the extrinsic reward is much higher than that from the intrinsic reward (adjusted by μ in (6)). Thus, the agent is able to escape local intrinsic reward optima and back-propagate the extrinsic reward to rule out bad behaviours and find the extrinsic optimal policy.

Pre-training Our algorithm does not rely on any a-priori knowledge about the task, and thus is bootstrapped using a two-step training procedure. In the first step, the agent explores randomly, and a single DQN module is set up. The experience samples that are gathered via random exploration are recorded with their associated labels. This pre-training phase ends when the set of experience samples reaches a pre-defined size. The traces gathered in the pre-training phase are used by the automata synthesis module to construct an initial DFA, which captures what the agent has discovered by random exploration. Once the initial DFA is constructed, we set up a separate DQN module and a separate replay buffer for each state of the DFA.

Algorithm 1: An Episode of Temporal DQN in DeepSynth

```
input  : automaton $\mathfrak{A}$ from the Synth step
output : approximate $Q$-function
$t = 0$
initialise the state to $(s_0, q_0)$
repeat
    $a_t = \arg\max_a B_{q_t}$ with $\epsilon$-greedy
    execute action $a_t$ and observe the total reward $r_t^T$
    observe the next image $x_{t+1}$
    semantic segmentation outputs $L(x_{t+1})$
    preprocess images $x_t$ and $x_{t+1}$
    store transition $(x_t, a_t, x_{t+1}, r_t^T, L(x_{t+1}))$ in $\mathcal{E}_{q_t}$
    sample minibatch from $\mathcal{E}_{q_t}$
    target value from target network: $\mathcal{P}_{q_t} = \{(input_l, target_l),\ l = 1, ..., |\mathcal{E}_{q_t}|)\}$
            $input_l = (s_l, a_l)$
            $target_l = r_l^T + \gamma \max_{a'} Q(s_{l+1}, a'|\theta^{\hat{B}_{q_{l+1}}})$
    $B_{q_t} \leftarrow$ RMSprop$(\mathcal{P}_{q_t})$
    every $TC$ steps clone the current network $B_{q_i}$ to $\hat{B}_{q_t}$
    update automaton state from $q_t$ to $q_{t+1}$ (call Synth if a new trace is observed to
      update the automaton $\mathfrak{A}$)
    $t = t + 1$
until end of trial
```

Algorithm 2: An Episode of Temporal NFQ in DeepSynth

```
input  : automaton $\mathfrak{A}$ from the Synth step
output : approximate $Q$-function
$t = 0$
initialise the state to $(s_0, q_0)$
repeat
    $a_t = \arg\max_a B_{q_t}$ with $\epsilon$-greedy
    execute action $a_t$ and observe the total reward $r_t^T$
    observe the next state $s_{t+1}$
    observe the next state label $L(s_{t+1})$
    store transition $(s_t, a_t, s_{t+1}, r_t^T, L(s_{t+1}))$ in $\mathcal{E}_{q_t}$
    target value from target network: $\mathcal{P}_{q_t} = \{(input_l, target_l),\ l = 1, ..., |\mathcal{E}_{q_t}|)\}$
            $input_l = (s_l, a_l)$
            $target_l = r_l^T + \gamma \max_{a'} Q(s_{l+1}, a'|\theta^{\hat{B}_{q_{l+1}}})$
    $B_{q_t} \leftarrow$ Adam $(\mathcal{P}_{q_t})$
    update automaton state from $q_t$ to $q_{t+1}$ (call Synth if a new trace is observed to
      update the automaton $\mathfrak{A}$)
    $t = t + 1$
until end of trial
```

In the main training phase, we then train these DQN modules together and apply automata synthesis to further improve the DFA. Whenever the automata synthesis adds a new state to the DFA, a new DQN module is added correspondingly, along with a new replay buffer.

When the state is in vector form and no convolutional layer is in use, we resort to NFQ-based deep-RL modules, instead of DQN modules, and the pre-training step might not be required. Similar to DQN, NFQ uses experience replay in order to efficiently approximate the Q-function in MDPs with continuous state spaces as in Algorithm 2. The only major difference is that in the Temporal NFQ, we do not maintain two instances of the same networks for periodic updates due to the simpler neural construction.

Application to Running Example In the running example, the agent exploration scheme is ϵ-greedy with diminishing ϵ where the rate of decrease also depends on the DFA state so that each module has enough chance to explore. For each automaton state $q_i \in \mathcal{Q}$ in the the product MDP, the associated deep RL module is called $B_{q_i}(s,a)$. Once the agent is at state $s^{\otimes} = (s, q_i)$, the neural net B_{q_i} is active and explores the MDP. Note that the modules are interconnected, as discussed above. For example, assume that by taking action a in state $s^{\otimes} = (s, q_i)$ the label $v = L(s')$ has been observed and as a result the agent is moved to state $s^{\otimes\prime} = (s', q_j)$, where $q_i \neq q_j$. By minimising the loss function $\mathfrak{L}$ the network parameters of B_{q_i} are updated such that $B_{q_i}(s,a)$ has minimum possible error to $R^T(s,a) + \gamma \max_{a'} B_{q_j}(s', a')$ while $B_{q_i} \neq B_{q_j}$. As such, the output of B_{q_j} directly affects B_{q_i} when the automaton state is changed. This allows the extrinsic reward to back-propagate efficiently, e.g., from modules B_{q_7} and B_{q_6} associated with q_7 and q_6 in Figure 3, to the initial module B_{q_1}. An overview of the algorithm is presented as Algorithm 1.

Define $\mathcal{E}_{q_i}$ as the projection of the general replay buffer $\mathcal{E}$ onto q_i. The size of the replay buffer for each module is limited and in the case of our running example $|\mathcal{E}_{q_i}| = 15000$. This includes the most recent frames that are observed when the product MDP state was $s^{\otimes} = (s, q_i)$. In the running example we used RMSProp for each module with uniformly sampled mini-batches of size 32.

In DeepSynth, each state of the automaton divides the general sequential task so that each transition between the automaton states represents an achievable Markovian sub-task. This allows the agent to jump from one sub-task to another by just switching between these modules as prescribed by the automaton. Thus, when employing neural networks, each network is a highly-specialized (sub-) agent that can efficiently achieve a particular sub-task. We remark that the sub-agents can potentially be used in transfer learning scenarios.

Input Embeddings We have experimented with a variety of different input embeddings, such as the one-hot encoding (Harris & Harris, 2010) and the integer encoding, in order to approximate the global Q-function with a single network. However, we have observed poor performance with all setups of this kind. We hypothesize that these encodings allow the network to assume an ordinal relationship between the states of the automaton. This means that by assigning integer numbers or one-hot codes, the automaton states are given a ranking. This disrupts Q-function generalisation since some states in the product MDP (the synchronised structure between the MDP and the automaton) are closer to each other. Consequently, we have turned to the use of n separate networks, which work together in

Task	Sequence							
Task1	Σ^*	wood	Σ^*	craft table				
Task2	Σ^*	grass	Σ^*	craft table				
Task3	Σ^*	wood	Σ^*	grass	Σ^*	iron	Σ^*	craft table
Task4	Σ^*	wood	Σ^*	smith table				
Task5	Σ^*	grass	Σ^*	smith table				
Task6	Σ^*	iron	Σ^*	wood	Σ^*	smith table		
Task7	Σ^*	wood	Σ^*	iron	Σ^*	craft table	Σ^*	gold

Table 1: High-level sequence for each task in the Minecraft environment, where Σ^* indicates that any other finite sequence of labels from the set Σ.

a hybrid fashion, meaning that the agent can switch between these neural networks as it jumps from one automaton state to another.

6. Experimental Results

6.1 Benchmarks and Setup

We evaluate the performance of DeepSynth on a comprehensive set of benchmarks, given in Table 3. The Minecraft environment (`minecraft-tX`) taken from Andreas et al. (2017) requires solving challenging low-level control tasks, and features sequential high-level goals. The two `mars-rover` benchmarks are taken from Hasanbeig (2020), and the models have uncountably infinite state spaces. Similarly, we treat Montezuma's Revenge (`montezuma`) as an infinite state experiment owing to its large and high-dimensional state space. The example `robot-surve` is adopted from Sadigh et al. (2014), and the task is to visit two regions in sequence. Models `slp-easy` and `slp-hard` are inspired by the noisy MDPs of Chapter 6 in (Sutton & Barto, 1998). The goal in `slp-easy` is to reach a particular region of the MDP and the goal in `slp-hard` is to visit four distinct regions sequentially in proper order. The `frozen-lake` benchmarks are similar: the first three are simple reachability problems and the last three require sequential visits of four regions, except that now there exist unsafe regions as well. The `frozen-lake` MDPs are stochastic and are adopted from the OpenAI Gym (Brockman et al., 2016).

All simulations have been carried out on a machine with an Intel Xeon 3.5 GHz processor, Nvidia Tesla V100 GPU and 16 GB of RAM, running Ubuntu 18. Full instructions on how to run DeepSynth are provided on a GitHub page that accompanies the distribution (Hasanbeig et al., 2021b):

`www.github.com/grockious/deepsynth (@018e606)`

In the following, we further elaborate on the Minecraft and Montezuma's Revenge experiments.

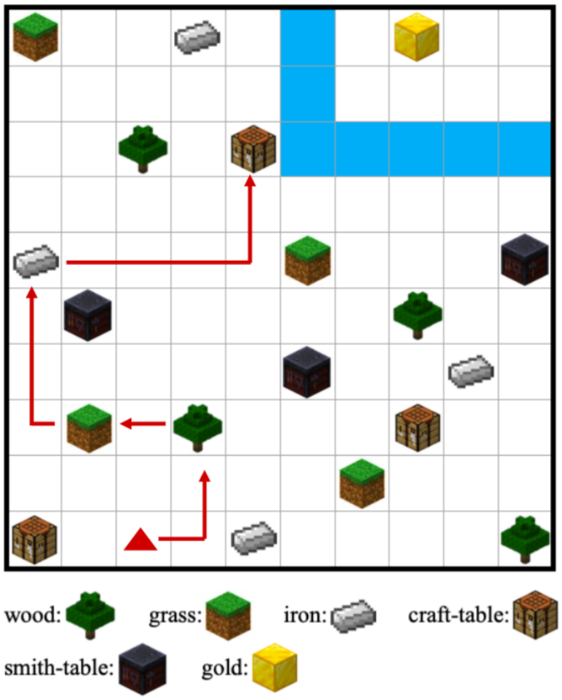


Figure 5: Example policy learnt by DeepSynth for Task 3 of the Minecraft environment with given vocabulary $\Sigma = \{$`wood, grass, iron, craft-table, smith-table, gold`$\}$.

6.2 Implementation Details

6.2.1 Minecraft

The Minecraft environment consists of 7 crafting tasks taken from (Andreas et al., 2017) that require the agent to execute optimal low-level actions in order to accomplish high-level objectives in proper order. The extrinsic reward of +1 is given to the agent only when the whole task is achieved and the tasks are fully unknown at the beginning. A few tasks include a long sequence of high-level objectives, and thus the associated reward is extremely sparse and non-Markovian (Table 1).

In this example, the agent location in the grid (Figure 5) is the MDP state $s \in \mathcal{S}$. At each state $s \in \mathcal{S}$ the agent has a set of actions $\mathcal{A} = \{left, right, up, down\}$ by which it is able to move to a neighbouring location $s' \in \mathcal{S}$ unless stopped by the boundary or by an obstacle. Obstacle locations are the blue cells and represent a river in the game. We assume the elements of the vocabulary set $\Sigma = \{\texttt{wood}, \texttt{grass}, \texttt{iron}, \texttt{craft-table}, \texttt{smith-table}, \texttt{gold}\}$ to be known.

Recall that the reward in this game is generally sparse and non-Markovian: the agent will receive a positive extrinsic reward only when a correct sequence is performed in each (high-level) task. Namely, the reward $\widehat{R} : (\mathcal{S} \times \mathcal{A})^* \times \mathcal{S} \to \mathbb{R}$ is a function over the set of finite paths. A trace-dependent reward is associated to the accomplishment of a given task: for example, performing a high-level task

Task 1: **wood** → grass → **craft-table** results in an extrinsic reward $\widehat{R_1}$, and for another high-level task, such as

$$\texttt{Task 4: grass} \rightarrow \textbf{\texttt{wood}} \rightarrow \texttt{iron} \rightarrow \textbf{\texttt{smith-table}}$$

the extrinsic reward is $\widehat{R_4}$. Further, these temporal orderings are initially unknown, and unlike in the work of Andreas et al. (2017) the agent is not equipped with any instructions to accomplish them. In these scenarios, existing exploration schemes fail, and prior work such as (Sadigh et al., 2014; Hasanbeig, Abate, & Kroening, 2019) requires the temporal ordering to be known in advance.

As mentioned in Section 5, for experiments where the state is already in vector form we employ NFQ modules instead of DQN ones. Similar to DQN, NFQ uses experience replay in order to efficiently approximate the Q-function in MDPs with continuous state spaces. Let $q_i \in \mathcal{Q}$ be a state in the DFA $\mathfrak{A}$. Then define $\mathcal{E}_{q_i}$ as the projection of $\mathcal{E}$ onto q_i. Each NFQ module B_{q_i} is trained by its associated experience set $\mathcal{E}_{q_i}$. At each iteration a pattern set $\mathcal{P}_{q_i}$ is generated based on $\mathcal{E}_{q_i}$:

$$\mathcal{P}_{q_i} = \{(input_l, target_l), l = 1, ..., |\mathcal{E}_{q_i}|)\},$$

where

$$input_l = (s_l, a_l),$$

and

$$target_l = r^T + \gamma \max_{a' \in \mathcal{A}} Q({s_l}', a'),$$

such that $\langle s_l, a_l, {s_l}', r^T, L(s_l')\rangle$. This pattern set is then used to train the neural net B_{q_i} as in Algorithm 2.

We use the Adam optimiser (Kingma & Ba, 2015) to update the network parameters in each module (line 12). Similar to the DQN modules, the back-propagation of the reward in NFQ modules starts from ones that are associated with extrinsically-rewarding states of the automaton, and eventually moves backward until it reaches the networks that are associated to the initial states. In this way we back-propagate the Q-value through the networks one by one. Through the extrinsic reward back-propagation those automaton paths that are not optimal with respect to the expected reward are eventually not chosen by the agent. This naturally leads to discovery of the high-level optimal paths, similar to the discovery of low-level action policies via RL. Once the Q-value has converged, the optimal policy is synthesised by maximising over this value at any given state.

6.2.2 Montezuma's Revenge

In Montezuma's Revenge, the emulator's internal state is not visible to the agent. The agent only observes an image, which is a matrix of pixel values that represent the current screen. Inspired by Mnih et al. (2015), we apply a basic preprocessing step to the Atari 2600 frames, designed to reduce the dimensionality of the input. We extract the image brightness and luminance from the RGB frame and rescale it to 84 × 84. Further, a grey-scaling pre-processing is applied and then the four most recent frames are stacked together to form the input to a DQN module.

The agent selects and executes actions according to an ϵ-greedy policy. Similar to the NFQ, the DQN module also uses experience replay, which averages the behaviour distribution over

Hyperparameter	Value	Description
minibatch size	32	Number of training cases over which each Stochastic Gradient Descent (SGD) update is computed
replay memory size	150000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames observed by the agent that are given as input to the Q network
target network update frequency (TC)	10000	The frequency (measured in the number of parameter updates) with which the target network is updated
discount factor	0.99	Discount factor gamma used in the Q-learning update
learning rate	0.00025	The learning rate used by RMSProp
initial exploration par	1	Initial value of ϵ in ϵ-greedy exploration
final exploration par	0.1	Final value of ϵ in ϵ-greedy exploration
final exploration frame	150,000	The number of frames over which the initial value of ϵ is linearly annealed to its final value
replay start size	8000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.

Table 2: Hyper-parameters of the DQN Modules for Montezuma's Revenge.

many of the previous states, smoothing out learning and avoiding oscillations or divergence in the parameters (Riedmiller, 2005; Mnih et al., 2015). Each DQN module stores a finite number of last experience tuples in the replay memory, and samples minibatches uniformly at random. Further, to improve the stability of the deep nets, we use a separate network in each module for generating the target y values in (4). Specifically, after a finite number of steps we clone the module network to obtain a target network, which is used to generate the Q-learning targets y. This modification makes the algorithm more stable compared to standard Q-Learning and NFQ updates (Mnih et al., 2015).

An overview of the algorithm is presented as Algorithm 1. The values and descriptions of all the hyper-parameters are provided in Table 2.

6.3 Results

Column three in Table 3 gives the number of states in the automaton that can be generated from the high-level objective sequences of the ground-truth task. Column four gives the

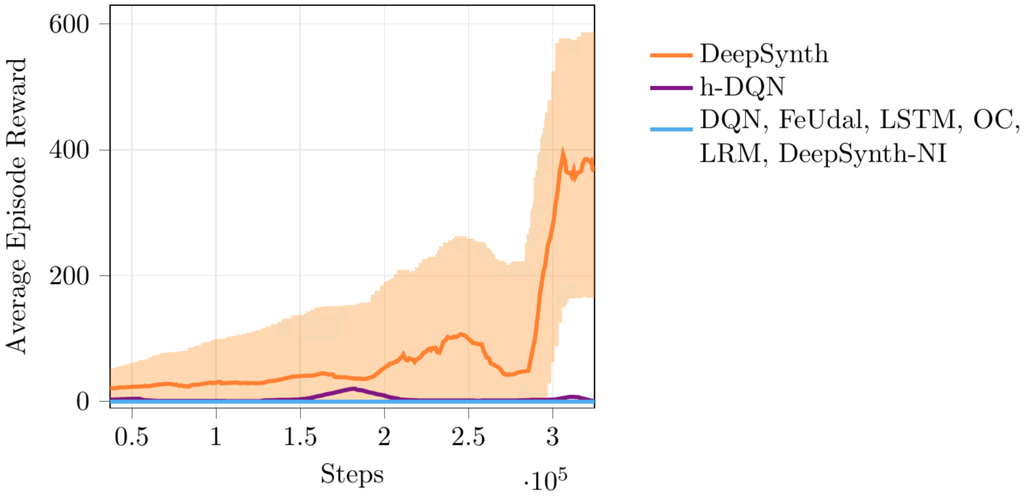


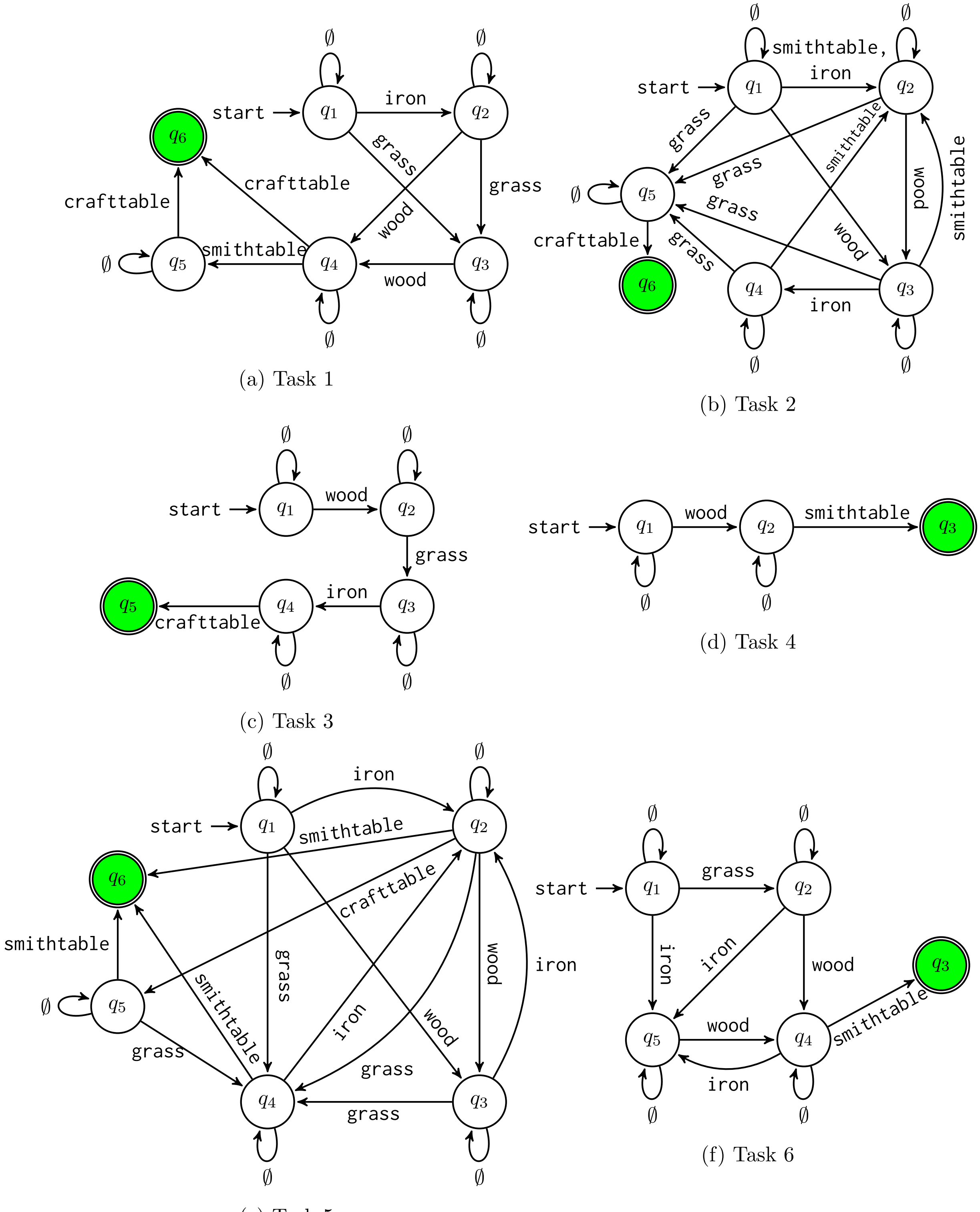
Figure 6: Average episode reward progress (over 5 runs) and its variance in Montezuma's Revenge with DeepSynth, h-DQN (Kulkarni et al., 2016), DQN (Mnih et al., 2015), FeUdal-LSTM (A. S. Vezhnevets et al., 2017), Option-Critic (OC) (Bacon et al., 2017), Learning Reward Machines (LRM) (Toro Icarte et al., 2019) and **DeepSynth with No Intrinsic reward (DeepSynth-NI)**. h-DQN opens a door but only after 2M steps. FeUdal and LSTM open a door after 100M and 200M steps, respectively. DQN, OC, LRM and **DeepSynth-NI** remain flat.

number of states of the automaton synthesised by DeepSynth[2], and column *product MDP* gives the number of states in the resulting product MDP (Definition 6). Finally, *max sat. prob. at* s_0 is the maximum probability of achieving the extrinsic reward from the initial state. In all experiments the high-level objective sequences are initially unknown to the agent. Furthermore, the extrinsic reward is only given when completing the task and reaching the objectives in the correct order.

For each task in the Minecraft environment, as shown in Table 1, we construct a DFA from trace sequences gathered by intrinsically-motivated exploration. The generated DFAs are presented in Figure 7a–7f and Figure 8, where the green state denotes task satisfaction.

The training progress for Montezuma's Revenge, and also Task 1 and Task 3 in Minecraft is plotted in Figure 6 and Figure 9, respectively. In Figure 6, we compare DeepSynth's average episode reward with several well-known baselines. In order to showcase the importance of the proposed intrinsic reward in (6), we test the performance of DeepSynth without any intrinsic reward. Namely, we set $\mu = 0$ in (5), and we call this special instance "DeepSynth with No Intrinsic reward (DeepSynth-NI)". It is interesting to observe that DeepSynth-NI was not able to obtain the key or open a door in Montezuma's revenge, highlighting the significant effect of the intrinsic reward on the performance of DeepSynth. Another interesting

[2] The difference between the size of synthesised DFAs and ground-truth DFAs is due to the assumption that the tasks are unknown to the agent and have to be uncovered via exploration. Specifically, from the agent's perspective, all the observed labels before a rewarding state is achieved are required to accomplish the task. Thus, synthesised DFAs can be larger than the ground-truth DFAs.



(a) Task 1

(b) Task 2

(c) Task 3

(d) Task 4

(e) Task 5

(f) Task 6

Figure 7: Automata generated for the Minecraft Tasks 1 to 6.

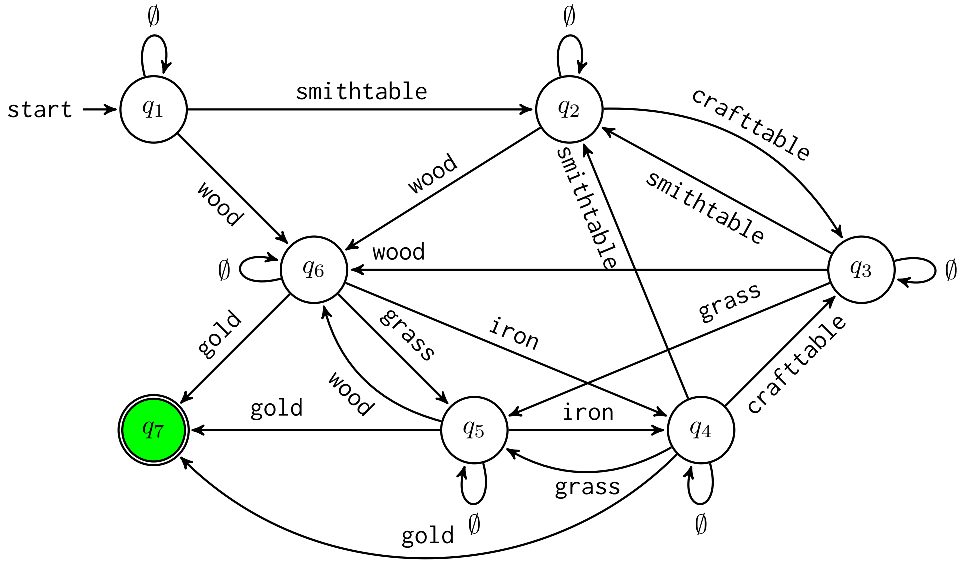


Figure 8: Automaton generated for the Minecraft Task 7.

observation is that, compared to other methods that are able to solve Montezuma's Revenge, DeepSynth proved to be much more sample efficient. Specifically, DeepSynth obtained a reduction of two orders of magnitude in the number of iterations required for policy synthesis. An important contributing factor to this efficiency is the explicit modelling of the high-level objectives and their semantic correlations in an automaton. Therefore, no matter how rare the required sequence of actions, they can still be delineated abstractly in terms of an automaton over high-level objectives that can guide the agent throughout the environment. These semantic correlations are implicitly learned in other methods by a much deeper network, hence the requirement for more samples.

Figure 10 presents the evolution of the loss functions in the Minecraft experiment for Task 1 and Task 3. For instance, in Figure 10.b, the orange line gives the loss for the very first deep net associated to the initial state of the DFA, the red and blue ones are of the intermediate states in the DFA and the green line is associated to the final state. This demonstrates an efficient back-propagation of the extrinsic reward from the final high-level state to the initial state. Once the last deep net converges, the expected reward is back-propagated to the second and so on. Each NFQ module has two hidden layers and 128 ReLUs. Note that the synthesised DFAs are guaranteed to encode all sequential label dependencies that are exemplified by the exploration traces (Jeppu et al., 2020). So, there may be a number of ways to accomplish a particular task in the synthesised DFAs. This, however, causes no harm since when the extrinsic reward is back-propagated, the non-optimal options fall out.

The crafting environment outputs a reward for Task 1 when the agent brings "wood" to the "craft table". Figure 9.a illustrates the results of training for Task 1. Note that with the very same training set $\mathcal{E}$ of 4500 training samples DeepSynth is able to converge while NFQ fails. Task 3 has a more complicated sequential structure, as given in Table 1. An

experiment	$\vert\mathcal{S}\vert$	# DFA states		product	max sat.	# episodes to convergence	
		task	synth.	MDP	prob. at s_0	DeepSynth	DQN
`minecraft-t1`	100	3	6	600	1	25	40
`minecraft-t2`	100	3	6	600	1	30	45
`minecraft-t3`	100	5	5	500	1	40	t/o
`minecraft-t4`	100	3	3	300	1	30	50
`minecraft-t5`	100	3	6	600	1	20	35
`minecraft-t6`	100	4	5	500	1	40	t/o
`minecraft-t7`	100	5	7	800	1	70	t/o
`mars-rover-1`	∞	3	3	∞	n/a	40	50
`mars-rover-2`	∞	4	4	∞	n/a	40	t/o
`montezuma`	∞	6	7	∞	n/a	300e3	t/o
`robot-surve`	25	3	3	75	1	10	10
`slp-easy-sml`	120	2	2	240	1	10	10
`slp-easy-med`	400	2	2	800	1	20	20
`slp-easy-lrg`	1600	2	2	3200	1	30	30
`slp-hard-sml`	120	5	5	600	1	80	t/o
`slp-hard-med`	400	5	5	2000	1	100	t/o
`slp-hard-lrg`	1600	5	5	8000	1	120	t/o
`frozen-lake-1`	120	3	3	360	0.9983	100	120
`frozen-lake-2`	400	3	3	1200	0.9982	150	150
`frozen-lake-3`	1600	3	3	4800	0.9720	150	150
`frozen-lake-4`	120	6	6	720	0.9728	300	t/o
`frozen-lake-5`	400	6	6	2400	0.9722	400	t/o
`frozen-lake-6`	1600	6	6	9600	0.9467	450	t/o

Table 3: Comparison between DeepSynth and DQN.

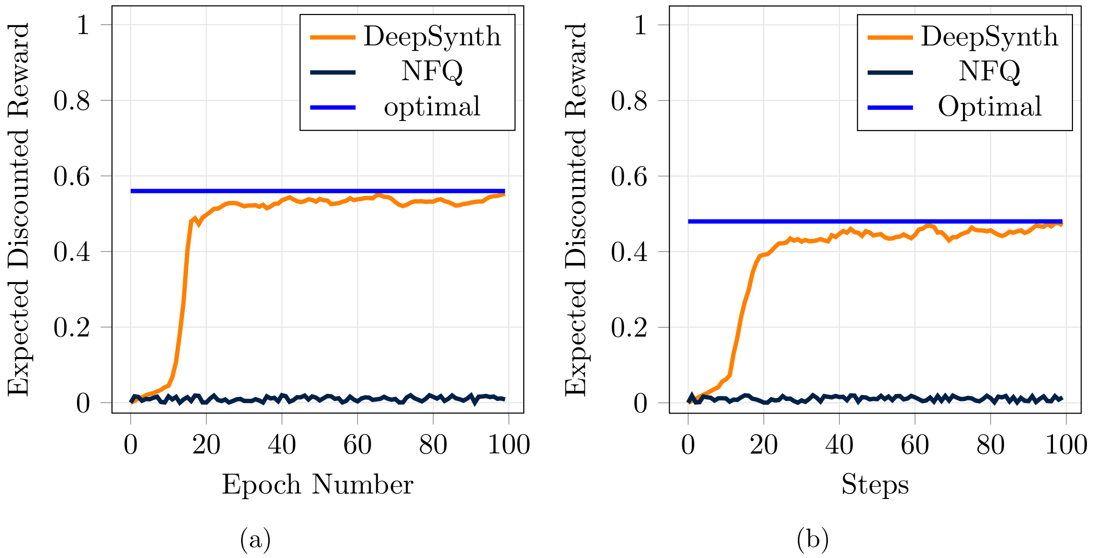
example policy learnt by DeepSynth for Task 3 is provided in Figure 5. Figure 9.b gives the result of training for Task 3 using DeepSynth and DQN where the training set $\mathcal{E}$ has 6000 training samples. However, for Task 3, NFQ failed to converge even after we increased the training set by an order of magnitude to 60000.

Lastly, we present the automata (Figure 11–15) that DeepSynth synthesised for the rest of the benchmarks in Table 3.

6.4 SAT-based Automata Synthesis vs. State Merge

We compare the automata synthesis algorithm we use to algorithms based on state merging. State merge is the established approach for model generation from traces. Traces are first converted into a Prefix Tree Acceptor (PTA). Model inference techniques are then used to identify pairs of equivalent states to be merged in the hypothesis model. Starting from the traditional *kTails* algorithm for state merging by Biermann and Feldman (1972), several alternatives to determine state equivalence have been proposed over the years (Walkinshaw & Bogdanov, 2008). For our experiment we used the MINT (Model INference Technique) tool by Walkinshaw (2018), which implements different variants of the state merge algorithm, including data classifiers (Walkinshaw et al., 2016) to check state equivalence for merging.

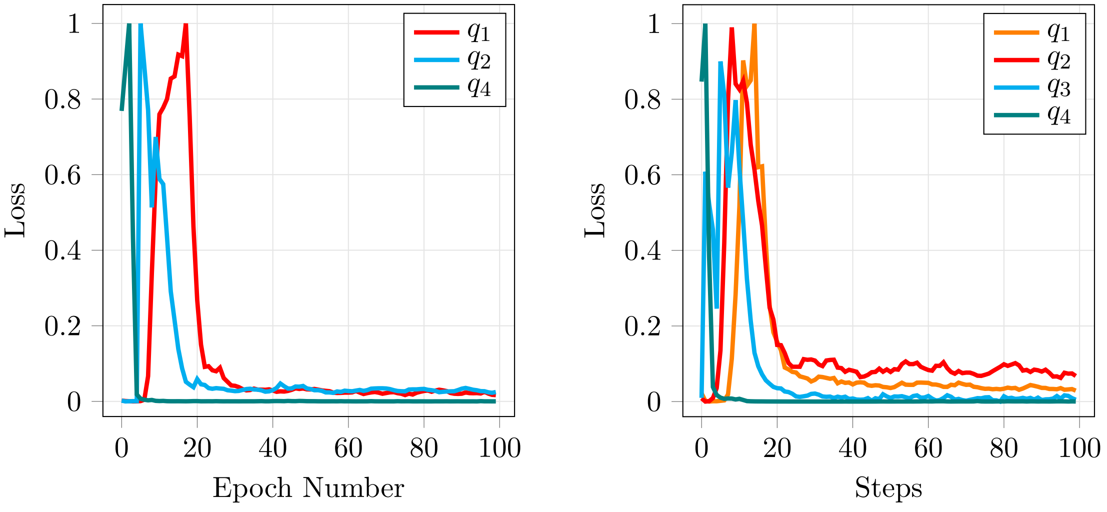
We generated models using MINT for all seven tasks for the Minecraft environment and explored different tool configurations to generate a model that best fits the input trace.



(a)

(b)

Figure 9: Training progress with DeepSynth and NFQ on the same training set $\mathcal{E}$ in the Minecraft Experiment for (a) Task 1 and (b) Task 3. The expected return is over state $s_0 = [4, 4]$ with origin being the bottom left corner cell.



(a) Task1

(b) Task3

Figure 10: Training progress for Tasks 1 and 3 with three and four active hybrid deep NFQ modules coupled together, respectively.

We observed that although MINT is faster, the automata generated by the tool are either too big (meaning they have a large number of states) or are over-generalised (in extreme cases they only have a single state), depending on the tool configurations. For instance, the smallest model that best fits the traces for Task 5 includes 49 states (Figure 16), and 14 states for Task 6 (Figure 17). Here, the 'start' label signifies the beginning of a new trace

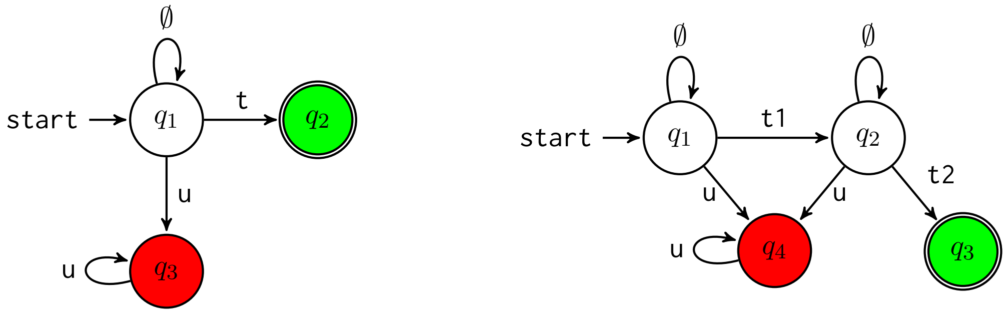


Figure 11: Automata synthesised for `mars-rover-1` and `mars-rover-2`.

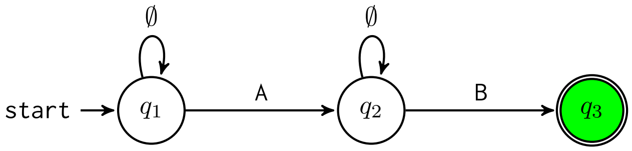


Figure 12: Automaton synthesised for `robot-surve`.

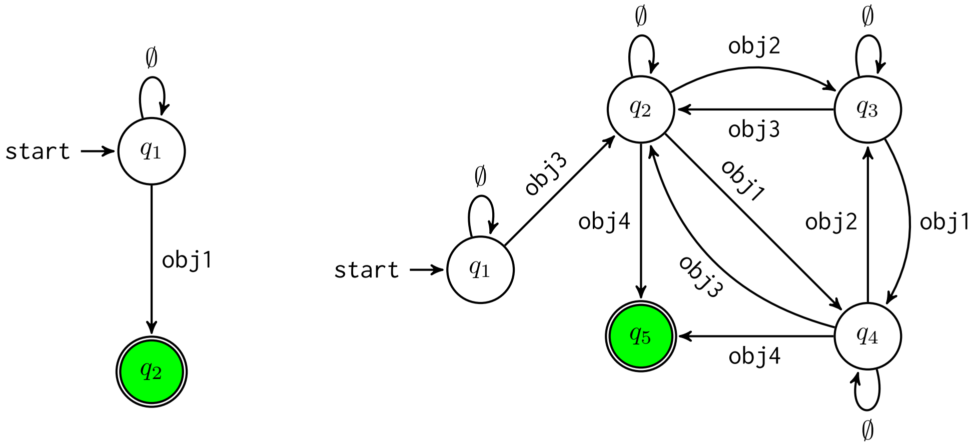


Figure 13: Automata synthesised for `slp-easy` and `slp-hard`.

obtained from another instance of random exploration. DeepSynth requires automata that are succinct and accurately represent sequential behaviour observed in the exploration trace to ensure fast and efficient learning. Since state merge algorithms do not produce the most succinct model that fits a given trace, we prefer using the automata synthesis algorithm described in Section 4.

6.5 Comparison with Tabu Search

The line of work presented by Toro Icarte et al. (2019), referred to as LRM here, uses reward machines learnt from trace data. It is important to note that the decision process considered in Toro Icarte et al. (2019) is partially observable. In particular, the task of the learned machines is to record what to remember so that there is Markovian predictability

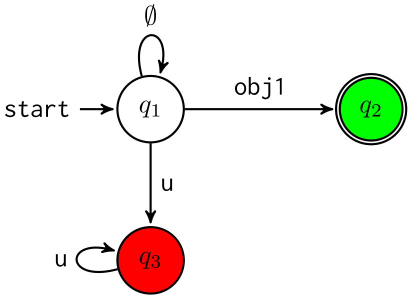


Figure 14: Automaton synthesised for `frozen-lake-1,2,3`.

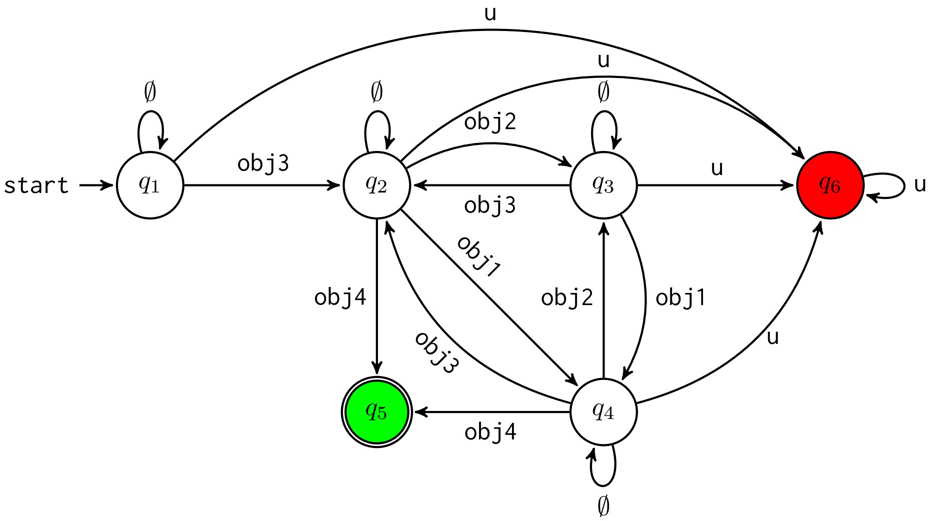


Figure 15: Automaton synthesised for `frozen-lake-4,5,6`.

from the observations and the reward machine state. Hence, the function of the generated automata is different from the function of the learned automata in this paper. Having this functional difference in mind, our interest persisted in comparing the two approaches and in the following we compare the two automata-learning methodologies.

The automata synthesis algorithm (Synth) used in this paper implements online learning. Synth converts the model learning problem into a Boolean Satisfiability (SAT) problem and uses a SAT solver to generate models. SAT solvers, in turn, use a backtracking search algorithm, DPLL (Davis & Putnam, 1960), to look for a satisfying assignment to the variables in the Boolean formula. LRM, on the other hand, uses Tabu search to explore and find reward machines from trace data. DPLL is complete and explores the entire search space, as opposed to Tabu search, which is local (Cook & Mitchell, 1996). Model completeness and accuracy is important in DeepSynth as the agent relies on the automaton to guide learning. The automata generated by Synth are complete in the sense that they capture all sequential behaviours seen in the trace.

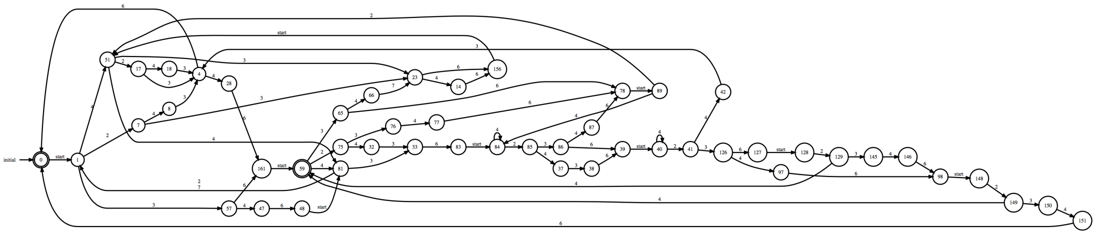


Figure 16: Best fit model for Task 5 generated by the MINT tool.

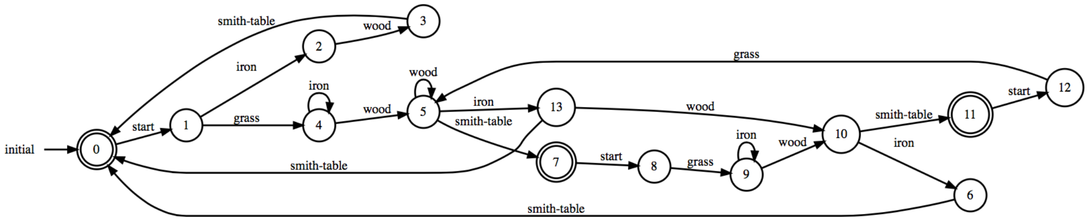


Figure 17: Best fit model for Task 6 generated by the MINT tool.

DeepSynth uses traces from random exploration to generate an automaton. Note that DeepSynth uses positive trace samples only. This allows us to add new traces incrementally to the existing automaton. In each episode sequential behaviours observed over iterations of exploration are incorporated into the automaton without changing the structure of the automaton inferred in previous episodes. More specifically, when the automaton is updated, it will always be a superset of the previous automaton. Consequently, the Q-networks can continue to be used and updated after an automaton is updated. Namely, the only required change is that more Q-networks need to be added; the old Q networks can be reused. This is not the case in LRM where Q-networks must be reset and relearned from scratch each time the automaton changes. Also, in contrast to LRM, Synth does not require an initial model.

For empirical evaluation, we applied Synth to generate automata from traces for the three examples used to benchmark the LRM approach—Cookie Domain, 2-Keys Domain and Symbol Domain. Details on the benchmarks can be found in (Toro Icarte et al., 2019).

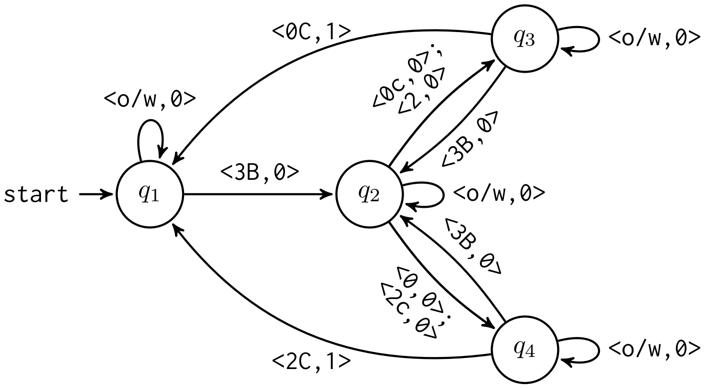


Figure 18: Perfect reward machine for Cookie domain (Toro Icarte et al., 2019).

Traces for each benchmark are obtained by executing the implementation of the LRM algorithm taken from (Icarte, 2020), where each trace is a sequence of (label, reward) pairs.

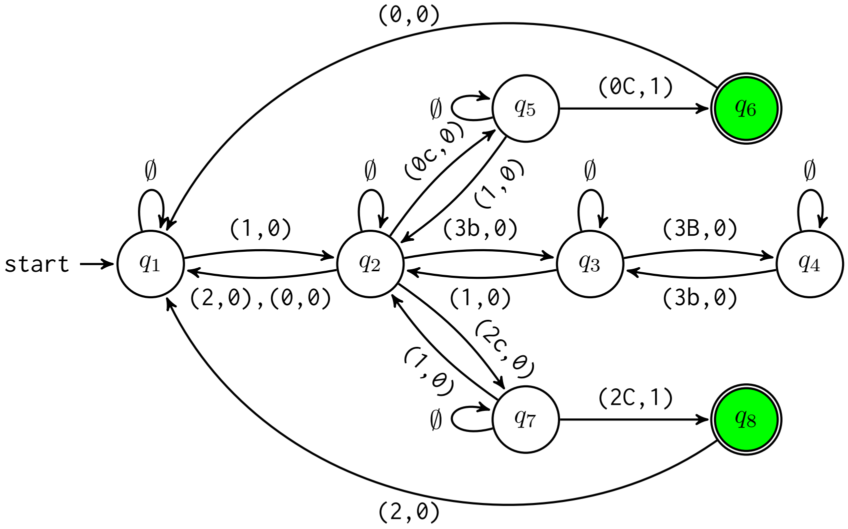


Figure 19: Automaton synthesised for Cookie domain.

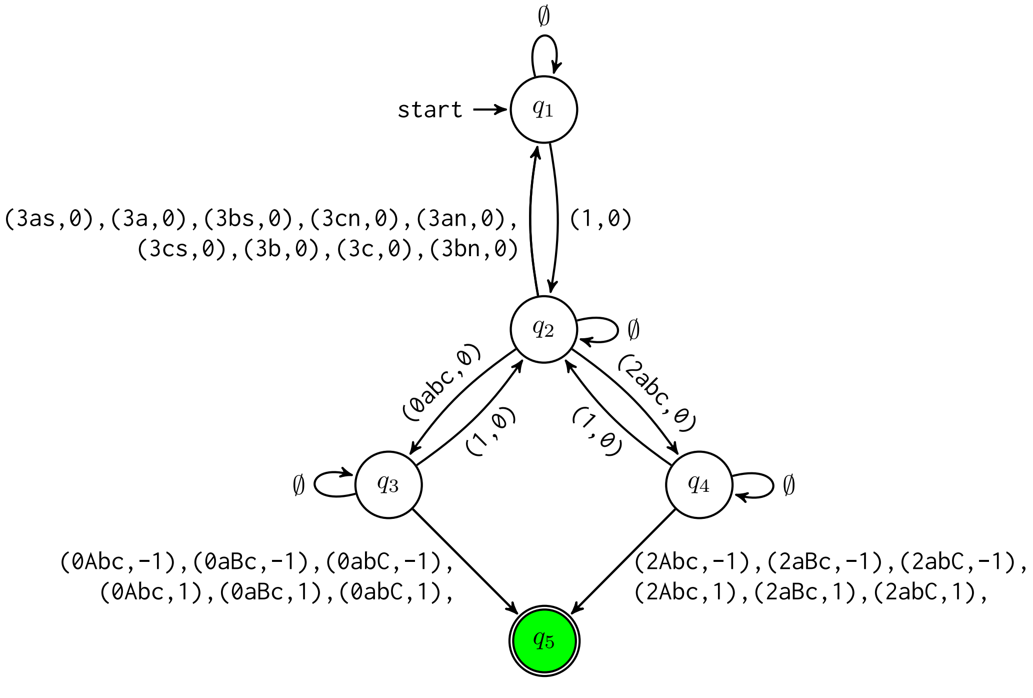

∅
start
q_1
(3as,0),(3a,0),(3bs,0),(3cn,0),(3an,0),
(3cs,0),(3b,0),(3c,0),(3bn,0)
(1,0)
q_2
∅
(0abc,0)
(1,0)
(1,0)
(2abc,0)
∅
q_3
q_4
∅
(0Abc,-1),(0aBc,-1),(0abC,-1),
(0Abc,1),(0aBc,1),(0abC,1),
(2Abc,-1),(2aBc,-1),(2abC,-1),
(2Abc,1),(2aBc,1),(2abC,1),
q_5

Figure 20: Automaton synthesised for Symbol domain.

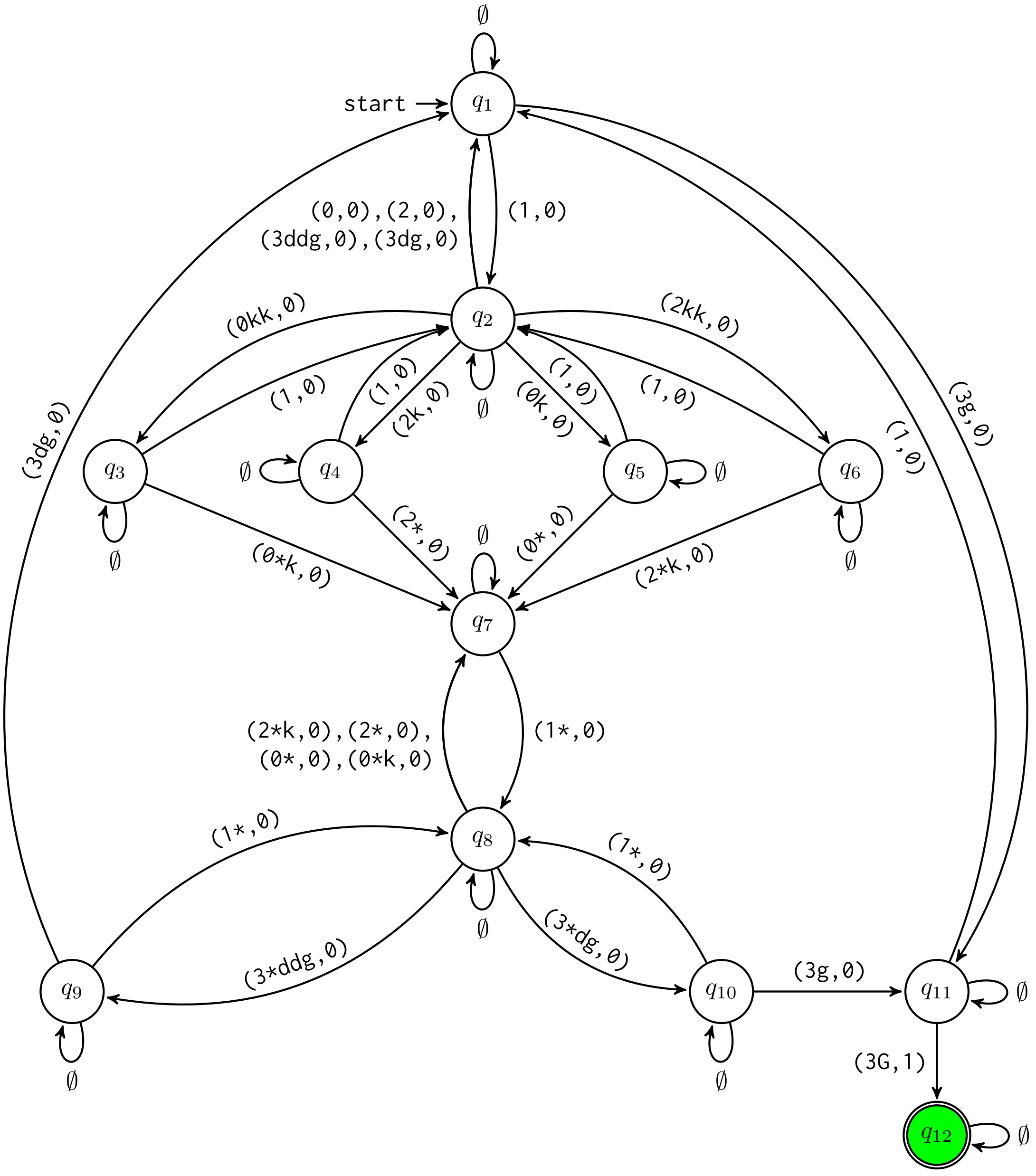
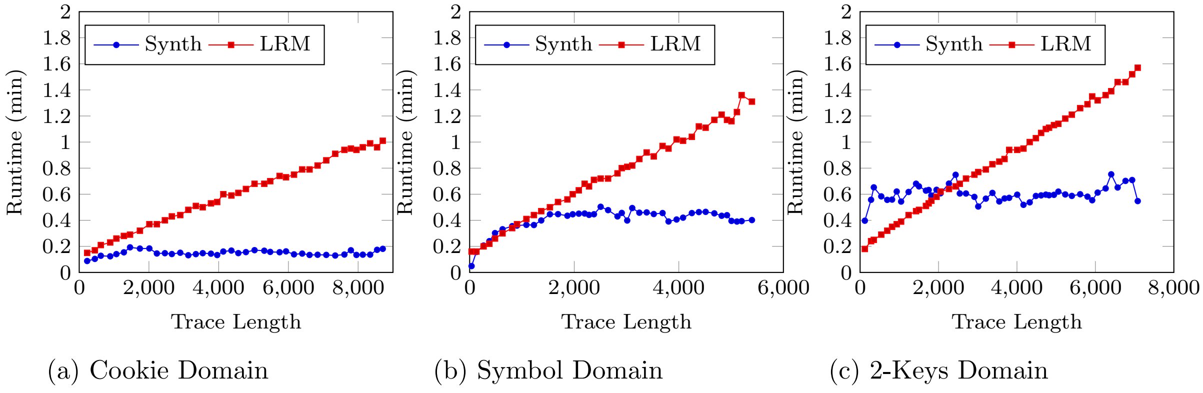


Figure 21: Automaton synthesised for 2-Keys domain.

Here we compare the automaton generated by Synth for the Cookie Domain (Figure 19) with its perfect reward machine (Figure 18) obtained from Toro Icarte et al. (2019).

The Cookie Domain consists of three rooms labelled 0, 2 and 3 connected by a hallway labelled 1. There is a button b in room 3 that, when pressed, causes a cookie c to randomly appear in either room 0 or room 2. Note that there is no cookie present at the beginning of an episode. In the trace, the button press is indicated by B while C indicates that the agent has eaten the cookie. The agent receives a reward of 1 for eating the cookie and 0 otherwise,



(a) Cookie Domain (b) Symbol Domain (c) 2-Keys Domain

Figure 22: Runtime comparison of Tabu search vs. the Synth algorithm.

as captured by the perfect reward machine in Figure 18. The automaton generated by Synth in Figure 19 accurately captures this behaviour—while the path $q_1 \to q_2 \to q_3 \to q_4$ captures the agent discovering and pressing the button in room 3, paths $q_2 \to q_5 \to q_6$ and $q_2 \to q_7 \to q_8$ capture the agent discovering the cookie and eating it in either room 0 or room 2 respectively. The automata generated by Synth for the other two benchmarks are provided in Figure 20–21.

For the sake of completeness, we also present the results of a runtime comparison of Synth and the LRM algorithm. The plots in Figure 22 provide a runtime comparison for increasing trace length on the three benchmarks. The plots show a linear increase in runtime for LRM as trace length increases. The runtime for Synth, on the other hand, saturates after an initial increase in runtime, and is much lower compared to LRM.

Given a set of observed traces, Synth generates a model from the first trace in the set, then incrementally adds any new behaviour seen in subsequent traces. For smaller trace sets, we see new behaviours being added to the model with each trace sequence in the set, but for larger trace sets subsequent trace sequences do not exhibit any new behaviours that are not already captured by the automaton. Therefore, the runtime saturates once all behaviours are captured by the generated automaton. The minimal noise observed in the runtime plots can be attributed to trace processing and plotting the generated automaton.

Increasing the parameters w and l for Synth can result in longer algorithm runtime. For w, the effect on runtime will depend on the nature of the label sequences, specifically the frequency of repeating patterns. Increasing l values instead leads to tighter constraints and a more complex SAT problem to solve.

7. Conclusions and Future Work

We have proposed a fully-unsupervised approach for training deep RL agents when the reward is extremely sparse or non-Markovian. We automatically infer a high-level structure from observed exploration traces using automata synthesis. The inferred automaton is a formal, un-grounded, human-interpretable representation of a complex task and its steps. We showed that we are able to efficiently learn policies that achieve complex high-level objectives using fewer training samples as compared to alternative algorithms.

Owing to the modular structure of the automaton, the overall task can be segmented into easier Markovian sub-tasks. Therefore, any segment of the proposed network that is associated with a sub-task can be used as a separate trained module in transfer learning. An advantage of the proposed method is that in problems where domain knowledge is available, this knowledge can be easily encoded as an automaton to guide learning. This enables the agent to solve complex tasks and saves the agent from an exhaustive exploration in the beginning. In particular, if the task DFA is known or even just partially known a priori, then our Synth step can exploit this partial automaton by incrementally adding any new labels or subtask sequences discovered during exploration. As an example, if the automaton in the second stage of Figure 3 is given initially, the agent is able to utilise the semantic correlation of the objects to facilitate its explorations and find the key faster.

An interesting direction to explore is the integration of widely-used automata synthesis methods such as MINT with the DeepSynth reward and deep RL architecture. A performance comparison could quantify the benefit of particular synthesis methods.

Another interesting research direction is to compare DQN with augmented input. For instance, the input to the DNN could include the segmented objects and their locations in the frame to determine whether the agent is able to learn an implicit correlation between the high-level objects.

Finally, wherever applicable, comparing the performance of DeepSynth with other methods that integrate automata learning and RL is an interesting direction to pursue.

Acknowledgments

The authors would like to thank Hadrien Pouget for interesting discussions and the anonymous reviewers for their insightful suggestions. This work was supported by a grant from the UK NCSC, and the Balliol College, Jason Hu scholarship. The work reported in this paper was done while Hosein Hasanbeig and Daniel Kroening were at the University of Oxford.

References

Abate, A., Almulla, Y., Fox, J., Hyland, D., & Wooldridge, M. (2023). Learning task automata for RL using hidden Markov models. In *European Conference on AI.*

Abbeel, P., Coates, A., Quigley, M., & Ng, A. Y. (2007). An application of reinforcement learning to aerobatic helicopter flight. In *NeurIPS* (pp. 1–8). MIT Press.

Alur, R., Bansal, S., Bastani, O., & Jothimurugan, K. (2022). A framework for transforming specifications in reinforcement learning. , 604–624.

Andreas, J., Klein, D., & Levine, S. (2017). Modular multitask reinforcement learning with policy sketches. In *ICML* (Vol. 70, pp. 166–175).

Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Inf. Comput.*, *75*(2), 87–106.

Bacon, P.-L., Harb, J., & Precup, D. (2017). The option-critic architecture. In *AAAI.*

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The Arcade learning environment: An evaluation platform for general agents. *Artificial Intelligence Research*, *47*, 253–279.

Berlyne, D. E. (1960). *Conflict, arousal, and curiosity.* McGraw-Hill Book Company.

Bertsekas, D. P., & Shreve, S. (2004). *Stochastic optimal control: The discrete-time case.* Athena Scientific.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-dynamic programming* (Vol. 1). Athena Scientific.

Biermann, A. W., & Feldman, J. A. (1972). On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, *21*(6), 592–597.

Brafman, R. I., De Giacomo, G., & Patrizi, F. (2018). LTLf/LDLf non-Markovian rewards. In *AAAI.*

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI gym. *arXiv*, *1606.01540*.

Brunke, L., Greeff, M., Hall, A. W., Yuan, Z., Zhou, S., Panerati, J., & Schoellig, A. P. (2021). Safe learning in robotics: From learning-based control to safe reinforcement learning. *arXiv preprint arXiv:2108.06266*.

Buzhinsky, I., & Vyatkin, V. (2017). Automatic inference of finite-state plant models from traces and temporal properties. *IEEE Trans. Ind. Informat.*, *13*(4), 1521–1530.

Buzhinsky, I., & Vyatkin, V. (2017). Modular plant model synthesis from behavior traces and temporal properties. In *Emerging Technologies and Factory Automation* (pp. 1–7). IEEE.

Cai, M., Hasanbeig, H., Xiao, S., Abate, A., & Kan, Z. (2021). Modular deep reinforcement learning for continuous motion planning with temporal logic. In *International Conference on Intelligent Robots and Systems (IROS) and IEEE Robotics and Automation Letters (RAL).*

Cai, M., Peng, H., Li, Z., Gao, H., & Kan, Z. (2021). Receding horizon control-based motion planning with partially infeasible LTL constraints. *IEEE Control Systems Letters*, *5*(4), 1279–1284.

Cai, M., & Vasile, C.-I. (2021). Safe-critical modular deep reinforcement learning with temporal logic through Gaussian processes and control barrier functions. *arXiv preprint arXiv:2109.02791*.

Camacho, A., Toro Icarte, R., Klassen, T. Q., Valenzano, R., & McIlraith, S. A. (2019). LTL and beyond: Formal languages for reward function specification in reinforcement learning. In *IJCAI* (pp. 6065–6073).

Chockler, H., Kesseli, P., Kroening, D., & Strichman, O. (2020). Learning the language of software errors. *Artificial Intelligence Research*, *67*, 881–903.

Cook, S., & Mitchell, D. (1996). Finding hard instances of the satisfiability problem: A survey. In *Satisfiability problem: Theory and applications.*

Csikszentmihalyi, M. (1990). *Flow: The psychology of optimal experience.* Harper & Row New York.

Daniel, C., Neumann, G., & Peters, J. (2012). Hierarchical relative entropy policy search. In *Artificial Intelligence and Statistics* (pp. 273–281).

Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. *J. ACM*, *7*(3), 201–215.

De Giacomo, G., Favorito, M., Iocchi, L., & Patrizi, F. (2020). Imitation learning over heterogeneous agents with restraining bolts. In *International Conference on Automated Planning and Scheduling* (pp. 517–521).

De Giacomo, G., Iocchi, L., Favorito, M., & Patrizi, F. (2019). Foundations for restraining bolts: Reinforcement learning with LTLf/LDLf restraining specifications. In *International Conference on Automated Planning and Scheduling* (Vol. 29, pp. 128–136).

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., & Clune, J. (2021, Feb 01). First return, then explore. *Nature*, *590*(7847), 580-586.

Fu, J., & Topcu, U. (2014). Probably approximately correct MDP learning and control with temporal logic constraints. In *Robotics: Science and Systems X.*

Fulton, N., & Platzer, A. (2018). Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *AAAI* (pp. 6485–6492).

Furelos-Blanco, D., Law, M., Russo, A., Broda, K., & Jonsson, A. (2020). Induction of subgoal automata for reinforcement learning. In *AAAI* (pp. 3890–3897).

Gaon, M., & Brafman, R. (2020). Reinforcement learning with non-Markovian rewards. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 34, pp. 3980–3987).

Giacobbe, M., Hasanbeig, H., Kroening, D., & Wijk, H. (2021). Shielding Atari games with bounded prescience. In *Autonomous Agents and Multiagent Systems* (pp. 1507–1509).

Glover, F., & Laguna, M. (1998). Tabu search. In *Handbook of Combinatorial Optimization* (Vols. 1–3, pp. 2093–2229). Springer.

Gulwani, S. (2012). Synthesis from examples. In *WAMBSE.*

Harris, D., & Harris, S. (2010). *Digital design and computer architecture.* Morgan Kaufmann.

Hasanbeig, H. (2020). *Safe and certified reinforcement learning with logical constraints* (PhD Dissertation). University of Oxford.

Hasanbeig, H., Abate, A., & Kroening, D. (2018). Logically-constrained reinforcement learning. *arXiv*, *1801.08099*.

Hasanbeig, H., Abate, A., & Kroening, D. (2019). Logically-constrained neural fitted Q-iteration. In *Autonomous Agents and Multiagent Systems* (pp. 2012–2014).

Hasanbeig, H., Abate, A., & Kroening, D. (2020). Cautious reinforcement learning with logical constraints. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems* (pp. 483–491).

Hasanbeig, H., Jeppu, N. Y., Abate, A., Melham, T., & Kroening, D. (2021a). DeepSynth: Automata synthesis for automatic task segmentation in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 35, pp. 7647–7656).

Hasanbeig, H., Jeppu, N. Y., Abate, A., Melham, T., & Kroening, D. (2021b). *DeepSynth: Automata synthesis for automatic task segmentation in deep reinforcement learning code repository.* `https://github.com/grockious/deepsynth`. GitHub.

Hasanbeig, H., Kantaros, Y., Abate, A., Kroening, D., Pappas, G. J., & Lee, I. (2019). Reinforcement learning for temporal logic control synthesis with probabilistic satisfaction guarantees. In *CDC* (pp. 5338–5343).

Hasanbeig, H., Kroening, D., & Abate, A. (2020a). Deep reinforcement learning with temporal logics. In *Formal Modeling and Analysis of Timed Systems* (Vol. 12288, pp. 1–22).

Hasanbeig, H., Kroening, D., & Abate, A. (2020b). Towards verifiable and safe model-free reinforcement learning. In *Workshop on Artificial Intelligence and Formal Verification, Logics, Automata and Synthesis (OVERLAY).*

Hasanbeig, H., Kroening, D., & Abate, A. (2022). LCRL: Certified policy synthesis via logically-constrained reinforcement learning. In *International Conference on Quantita-*

tive Evaluation of SysTems.

Hasanbeig, H., Kroening, D., & Abate, A. (2023). Certified reinforcement learning with logic guidance. *Artificial Intelligence*, 103949. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0004370223000954`

Hasanbeig, H., Sharma, H., Betthauser, L., Frujeri, F. V., & Momennejad, I. (2023). ALLURE: A systematic protocol for auditing and improving LLM-based evaluation of text using iterative in-context-learning. *arXiv preprint arXiv:2309.13701*.

Heule, M. J. H., & Verwer, S. (2013). Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, *18*(4), 825–856.

Hwang, J.-J., Yu, S. X., Shi, J., Collins, M. D., Yang, T.-J., Zhang, X., & Chen, L.-C. (2019). SegSort: Segmentation by discriminative sorting of segments. In *ICCV* (pp. 7334–7344).

Icarte, T. (2020). *LRM Learning reward machines for partially observable reinforcement learning.* `https://bitbucket.org/RToroIcarte/lrm/src/master/`. Bitbucket.

Jeppu, N. Y. (2020). *Trace2model code repository.* https://github.com/natasha-jeppu/Trace2Model.

Jeppu, N. Y., Melham, T., Kroening, D., & O'Leary, J. (2020). Learning concise models from long execution traces. In *Design Automation Conference* (pp. 1–6). ACM/IEEE.

Ji, X., Henriques, J. F., & Vedaldi, A. (2019). Invariant information clustering for unsupervised image classification and segmentation. In *ICCV* (pp. 9865–9874).

Jiang, Y., Bharadwaj, S., Wu, B., Shah, R., Topcu, U., & Stone, P. (2021). Temporal-logic-based reward shaping for continuing learning tasks. , *35*(9), 7995–8003.

Kearns, M., & Singh, S. (2002). Near-optimal reinforcement learning in polynomial time. *Machine Learning*, *49*(2-3), 209–232.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations.*

Koul, A., Fern, A., & Greydanus, S. (2019). Learning finite state representations of recurrent policy networks. In *International Conference on Learning Representations.*

Krishnan, S. C., Puri, A., Brayton, R. K., & Varaiya, P. P. (1995). The Rabin index and chain automata, with applications to automata and games. In *Computer-Aided Verification (CAV)* (pp. 253–266). Springer.

Kulkarni, T. D., Narasimhan, K., Saeedi, A., & Tenenbaum, J. (2016). Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *NeurIPS* (pp. 3675–3683).

Lang, K. J., Pearlmutter, B. A., & Price, R. A. (1998). Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In *Grammatical Inference* (Vol. 1433, pp. 1–12). Springer.

Lavaei, A., Somenzi, F., Soudjani, S., Trivedi, A., & Zamani, M. (2020). Formal controller synthesis for continuous-space MDPs via model-free reinforcement learning. In *International Conference on Cyber-Physical Systems* (pp. 98–107).

Littman, M. L., Topcu, U., Fu, J., Isbell, C., Wen, M., & MacGlashan, J. (2017). Environment-independent task specifications via GLTL. *arXiv preprint arXiv:1704.04341*.

Liu, W., Wei, L., Sharpnack, J., & Owens, J. D. (2019). Unsupervised object segmentation with explicit localization module. *arXiv*, *1911.09228*.

Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. In *ACM Workshop on Networks* (pp. 50–56).

Memarian, F., Goo, W., Lioutikov, R., Topcu, U., & Niekum, S. (2021). Self-supervised online reward shaping in sparse-reward environments. *arXiv preprint arXiv:2103.04529*.

Memarian, F., Xu, Z., Wu, B., Wen, M., & Topcu, U. (2020). Active task-inference-guided deep inverse reinforcement learning. In *Conference on Decision and Control, CDC* (pp. 1932–1938). IEEE.

Mitta, R., Hasanbeig, H., Kroening, D., & Abate, A. (2022). Risk-aware Bayesian reinforcement learning for cautious exploration. In *NeurIPS 2022 MLSW.*

Mitta, R., Hasanbeig, H., Wang, J., Kroening, D., Kantaros, Y., & Abate, A. (2024). Safeguarded progress in reinforcement learning: Safe Bayesian exploration for control policy synthesis. In *Proceedings of the aaai conference on artificial intelligence* (Vol. 38, pp. 21412–21419).

Mnih, V., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Momennejad, I., Hasanbeig, H., Vieira Frujeri, F., Sharma, H., Jojic, N., Palangi, H., ... Larson, J. (2024). Evaluating cognitive maps and planning in large language models with CogEval. *Advances in Neural Information Processing Systems*, *36*.

Momennejad, I., Russek, E., Cheong, J., Botvinick, M., Daw, N., & Gershman, S. (2017). The successor representation in human reinforcement learning. *Nature Human Behaviour*, *1*(9), 680–692.

Neider, D., Gaglione, J.-R., Gavran, I., Topcu, U., Wu, B., & Xu, Z. (2021). Advice-guided reinforcement learning in a non-Markovian environment. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 35, pp. 9073–9080).

Nejati, A., Lavaei, A., Jagtap, P., Soudjani, S., & Zamani, M. (2023). Formal verification of unknown discrete-and continuous-time systems: A data-driven approach. *IEEE Transactions on Automatic Control*.

Polydoros, A. S., & Nalpantidis, L. (2017). Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, *86*(2), 153–173.

Precup, D. (2001). *Temporal abstraction in reinforcement learning* (PhD Dissertation). University of Massachusetts Amherst.

Rens, G., & Raskin, J.-F. (2020). Learning non-Markovian reward models in MDPs. *arXiv*, *2001.09293*.

Rens, G., Raskin, J.-F., Reynouad, R., & Marra, G. (2020). Online learning of non-Markovian reward models. *arXiv*, *2009.12600*.

Riedmiller, M. (2005). Neural fitted Q iteration – first experiences with a data efficient neural reinforcement learning method. In *ECML* (Vol. 3720, pp. 317–328). Springer.

Ringstrom, T. J., Hasanbeig, H., & Abate, A. (2020). Jump operator planning: Goal-conditioned policy ensembles and zero-shot transfer. *arXiv preprint arXiv:2007.02527*.

Ryan, R. M., & Deci, E. L. (2000). Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary Educational Psychology*, *25*(1), 54–67.

Sadigh, D., Kim, E. S., Coogan, S., Sastry, S. S., & Seshia, S. A. (2014). A learning based approach to control synthesis of Markov decision processes for linear temporal logic specifications. In *Conference on Decision and Control* (pp. 1091–1096).

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., ...

Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*, 484–503.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction* (Vol. 1). MIT Press Cambridge.

Toro Icarte, R., Klassen, T. Q., Valenzano, R., & McIlraith, S. A. (2018). Teaching multiple tasks to an RL agent using LTL. In *Autonomous Agents and Multiagent Systems* (pp. 452–461).

Toro Icarte, R., Waldie, E., Klassen, T., Valenzano, R., Castro, M., & McIlraith, S. (2019). Learning reward machines for partially observable reinforcement learning. In *NeurIPS* (pp. 15497–15508).

Ulyantsev, V., Buzhinsky, I., & Shalyto, A. (2018, Feb 01). Exact finite-state machine identification from scenarios and temporal properties. *International Journal on Software Tools for Technology Transfer*, *20*(1), 35–55.

Ulyantsev, V., & Tsarev, F. (2011). Extended finite-state machine induction using SAT-solver. In *International Conference on Machine Learning and Applications and Workshops* (pp. 346–349).

Vezhnevets, A., Mnih, V., Osindero, S., Graves, A., Vinyals, O., Agapiou, J., et al. (2016). Strategic attentive writer for learning macro-actions. In *NeurIPS* (pp. 3486–3494).

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., & Kavukcuoglu, K. (2017). FeUdal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning* (p. 3540–3549).

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., ... Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 1–5.

Walkinshaw, N. (2018). *MINT framework code repository.* https://github.com/neilwalkinshaw/mintframework.

Walkinshaw, N., & Bogdanov, K. (2008). Inferring finite-state models with temporal constraints. In *Automated Software Engineering* (p. 248-257). IEEE.

Walkinshaw, N., Bogdanov, K., Holcombe, M., & Salahuddin, S. (2007). Reverse engineering state machines by interactive grammar inference. In *Working Conference on Reverse Engineering* (pp. 209–218). IEEE.

Walkinshaw, N., Taylor, R., & Derrick, J. (2016). Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, *21*(3), 811–853.

Wang, J., Hasanbeig, H., Tan, K., Sun, Z., & Kantaros, Y. (2023). Mission-driven exploration for accelerated deep reinforcement learning with temporal logic task specifications. *arXiv preprint arXiv:2311.17059*.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3-4), 279–292.

Xu, Z., Gavran, I., Ahmad, Y., Majumdar, R., Neider, D., Topcu, U., & Wu, B. (2020). Joint inference of reward machines and policies for reinforcement learning. In *AAAI* (Vol. 30, pp. 590–598).

Xu, Z., Wu, B., Ojha, A., Neider, D., & Topcu, U. (2021). Active finite reward automaton inference and reinforcement learning using queries and counterexamples. In *International Cross-domain Conference for Machine Learning and Knowledge Extraction* (pp. 115–135).

Yang, C., Littman, M., & Carbin, M. (2021). Reinforcement learning for general LTL objectives is intractable. *arXiv preprint arXiv:2111.12679*.

Yuan, L. Z., Hasanbeig, H., Abate, A., & Kroening, D. (2019). Modular deep reinforcement learning with temporal logic specifications. *arXiv*, *1909.11591*.

Zheng, Z., & Yang, Y. (2021). Rectifying pseudo label learning via uncertainty estimation for domain adaptive semantic segmentation. *International Journal of Computer Vision*, *129*, 1106—1120.

Zhou, Z., et al. (2017). Optimizing chemical reactions with deep reinforcement learning. *ACS Central Science*, *3*(12), 1337–1344.