DeepSynth: Automata Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning

Mohammadhosein Hasanbeig, Natasha Yogananda Jeppu, Alessandro Abate, Tom Melham, Daniel Kroening*

Computer Science Department, University of Oxford, Parks Road, Oxford, United Kingdom, OX1 3QD
{hosein.hasanbeig, natasha.yogananda.jeppu, alessandro.abate, tom.melham}@cs.ox.ac.uk and daniel.kroening@magd.ox.ac.uk

Abstract

This paper proposes DeepSynth, a method for effective training of deep Reinforcement Learning (RL) agents when the reward is sparse and 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 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 RL 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 non-Markovian rewards. We have evaluated DeepSynth's performance in a set of experiments that includes the Atari game *Montezuma's Revenge*. Compared to existing approaches, we obtain a reduction of *two* orders of magnitude in the number of iterations required for policy synthesis, and also 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 and Nalpantidis 2017), resource and traffic management (Mao et al. 2016; Sadigh et al. 2014), drone control (Abbeel et al. 2007), chemical engineering (Zhou et al. 2017), and gaming (Mnih et al. 2015). While RL is a very general architecture, many advances in the last decade have been achieved using specific instances of RL that employ a deep neural network to synthesise optimal policies. A deep RL algorithm, AlphaGo (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 (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 many problems in complex environments, tasks that feature extremely sparse, non-Markovian rewards or other long-term sequential structures are often difficult or impossible to solve by unaided RL. A well-known example is the Atari game *Montezuma's Revenge*, in which DQN (Mnih et al. 2015) failed to score. Interestingly, Montezuma's Revenge and other hard problems often require learning to 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.

Contribution: In this paper we propose DeepSynth, 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. We assume that these sequential dependencies have a *regular* nature, in formal language theory sense (Gulwani 2012). The identification of dependency on a sequence of high-level objectives is the key to breaking down a complex task into a series of 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 automata representation allows a human observer to easily interpret the deep RL solution in a high-level manner, and to gain more insight into the optimality of 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 the deep RL. When dealing with raw image input, we assume that an off-the-shelf unsupervised image segmentation method, e.g. (Liu et al. 2019), can provide enough object candidates in order to identify semantically distinguished states. We evaluate the performance of DeepSynth on a selection 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 various domains where other related approaches fail. DeepSynth represents a better integration of deep RL and formal automata synthesis than previous approaches, making learning for non-Markovian rewards more scalable.

*The work in this paper was done prior to joining Amazon.

Related Work: Our research employs formal methods to deal with the sparse reward problem in RL. In the RL literature, the dependency of rewards on objectives is often tackled with *options* (Sutton and Barto 1998), or, in general, the dependencies are structured *hierarchically*. Current approaches to Hierarchical RL (HRL) very much depend on state representations and whether they are structured enough for a suitable reward signal to be effectively engineered manually. HRL therefore often requires detailed supervision in the form of explicitly specified high-level actions or intermediate supervisory signals (Precup 2001; Kearns and Singh 2002; Daniel et al. 2012; Kulkarni et al. 2016; 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 and Topcu 2014; Sadigh et al. 2014; Fulton and Platzer 2018; Cai 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 et al. 2019a; Yuan et al. 2019; De Giacomo et al. 2019, 2020; Hasanbeig et al. 2019d,c, 2020b,c; Kazemi and Soudjani 2020; Lavaei et al. 2020). These approaches are 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 and Raskin 2020; Furelos-Blanco et al. 2020; Gaon and Brafman 2020; Xu et al. 2020), by inferring automata from exploration traces.

Automata inference in (Toro Icarte et al. 2019) uses a local-search based algorithm, Tabu search (Glover and Laguna 1998). The automata inference algorithm that we employ uses SAT, where the underlying search algorithm is a backtracking search method called DPLL (Davis and Putnam 1960). In comparison with Tabu search, the DPLL algorithm is complete and explores the entire search space efficiently (Cook and Mitchell 1996), producing more accurate representations of the trace. The Answer Set Programming (ASP) based algorithm used to learn automata in (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 (Angluin 1987). This is used to infer automata in (Rens et al. 2020; Rens and Raskin 2020; Gaon and 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 and Feldman 1972). State-merge and some of its variants (Lang et al. 1998; Walkinshaw et al. 2007) do not always produce the most succinct automaton but generate an approximation that conforms to the trace (Ulyantsev, Buzhinsky, and Shalyto 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 the extended version of this work (Hasanbeig et al. 2019b).

A number of approaches combine SAT with state-merge to generate minimal automata from traces (Ulyantsev and Tsarev 2011; Heule and Verwer 2013). A similar SAT based algorithm is employed in (Xu et al. 2020) 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 it to guide exploration towards the rewarding states.

Further related work is *policy sketching* (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 the method assumes policy sketches, i.e. temporal instructions, to be available to the agent. There is also recent work on learning underlying non-Markovian objectives when an optimal policy or human demonstration is available (Koul et al. 2019; Memarian et al. 2020).

2 Background on Reinforcement Learning

We consider a conventional RL setup, consisting of an agent interacting with an environment, which is modelled as an unknown general Markov Decision Process (MDP):

Definition 2.1 (General MDP) *The tuple $\mathfrak{M} = (\mathcal{S}, \mathcal{A}, s_0, P, \Sigma, L)$ is a general 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 sigma-algebra on the state space (Bertsekas and Shreve 2004). Σ is called the vocabulary set and it is a finite set of atomic propositions. There exists a labelling function $L : \mathcal{S} \to 2^{\Sigma}$ that assigns to each state $s \in \mathcal{S}$ a set of atomic propositions $L(s) \in 2^{\Sigma}$.*

Definition 2.2 (Path) *In a general MDP $\mathfrak{M}$, an infinite path ρ starting at s_0 is an infinite sequence of state transitions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ 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 $\rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$. The set of infinite paths is $(\mathcal{S} \times \mathcal{A})^{\omega}$ and the set of finite paths is $(\mathcal{S} \times \mathcal{A})^* \times \mathcal{S}$.*

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, $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$. 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. Similarly, 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 one possible realisation of R and $\widehat{R}$ at time step n is denoted by r_n and $\widehat{r}_n$ respectively.

Due to space limitations we present the formal background on RL in (Hasanbeig et al. 2019b) and we only introduce the notation we use. The expected discounted return for a policy π and state s is denoted by $U^\pi(s)$, which is maximised by the optimal policy π^*. Similarly, at each state the optimal policy maximises the Q-function $Q(s, a)$ over the set of actions. The Q-function can be parameterised using a parameter set θ^Q and updated by minimising a loss $\mathfrak{L}(\theta^Q)$.

3 Background on Automata Synthesis

The automata synthesis algorithm extracts information from trace sequences over finite paths in order to construct a succinct automaton that represents the behaviour exemplified by these traces. This architecture is an instance of the general *synthesis from examples* approach (Gulwani 2012; Jeppu et al. 2020). Our synthesis method scales to long traces by employing a segmentation approach, achieving automata learning in close-to-polynomial runtime (Jeppu 2020).

The synthesis algorithm takes as input a trace sequence and generates an N-state automaton conforming to the trace input. Starting with $N = 1$, the algorithm systematically searches for the required automaton, incrementing N by 1 each time the search fails. This ensures that the smallest automaton conforming to the input trace is generated. The algorithm additionally uses a hyper-parameter w to tackle growing algorithm complexity for long trace input. The synthesis algorithm divides the trace into segments using a sliding window of size w and only unique segments are used for further processing. In this way, the algorithm exploits the presence of patterns in traces. Multiple occurrences of these patterns are processed only once, thus reducing the size of the input to the algorithm.

Automata generated using only *positive* trace samples tend to overgeneralise (Gold 1978). This is mitigated by performing a compliance check of the automaton against the trace input to eliminate any transition sequences of length l that are accepted by the generated automaton but do not appear in the trace. The hyper-parameter l therefore controls the degree of generalisation in the generated automaton. A higher value for l yields more exact representations of the trace. The correctness of the generated automaton is verified by checking if the automaton accepts the input trace. If the check fails, missing trace data is incrementally added to refine the generated model, until the check passes. Further details on tuning the hyper-parameters w and l are given in the next section.

4 DeepSynth

A schematic of the DeepSynth algorithm is provided in Fig. 1 and the algorithm is described step-by-step in this section. We begin by introducing the first level of Montezuma's Revenge as a running example (Bellemare et al. 2013). Unlike other Atari games where the primary goal is limited to avoiding obstacles or collecting items with no particular order, Montezuma's Revenge requires the agent to perform a long,

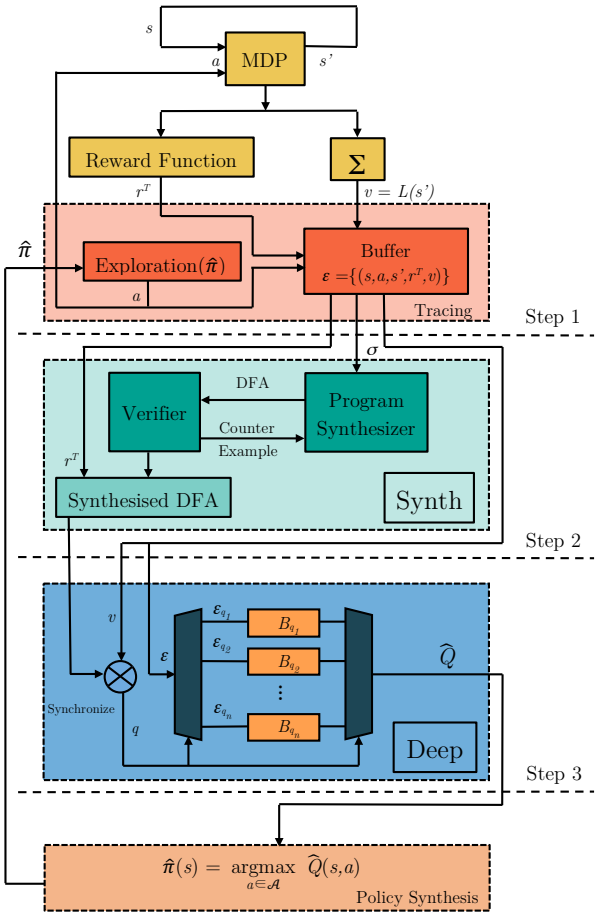
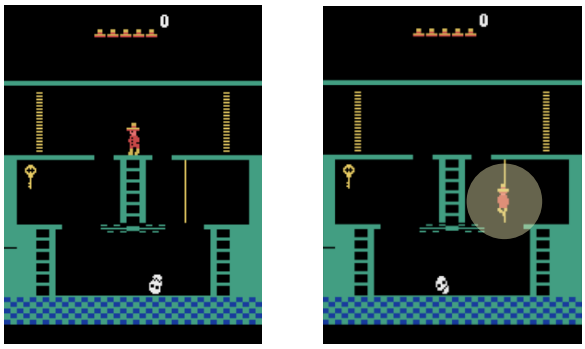


Figure 1: The DeepSynth Algorithm

complex sequence of actions before receiving any reward. The agent must find a key and open either door in Fig. 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 (Vezhnevets et al. 2017), and Go-Explore (Ecoffet et al. 2021).

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, Henriques, and Vedaldi 2019; Hwang et al. 2019; Zheng and Yang 2021). In this work, we assume that an off-the-shelf image segmentation algorithm can provide plausible object candidates, e.g. (Liu et al. 2019). The key to solving a complex task such as Montezuma's Revenge is finding the semantic correlation between the objects



(a) (b)

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

in the scene. When a human player tries to solve this game the semantic correlations, such as "keys open doors", are partially known and the player's behaviour is driven by exploiting 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 and 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 previously unknown correlations are discovered. This exploration scheme imitates biological cognitive growth in a formal and explainable way, and is driven by an intrinsic motivation to explore 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 of this paper we assume that semantic correlations are unknown to the agent. The agent starts with no prior knowledge of the sparse reward task or the correlation of the high-level objects.

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 Fig. 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.

Tracing (Step 1 in Fig. 1): Note that the task is unknown initially and the extrinsic reward is non-Markovian and extremely sparse. The agent receives a reward $\widehat{R} : (\mathcal{S} \times \mathcal{A})^* \times \mathcal{S} \to \mathbb{R}$ only when a correct sequence of state-action pairs and their associated object correlations are visited. In order to guide the agent to find the optimal sequence, DeepSynth

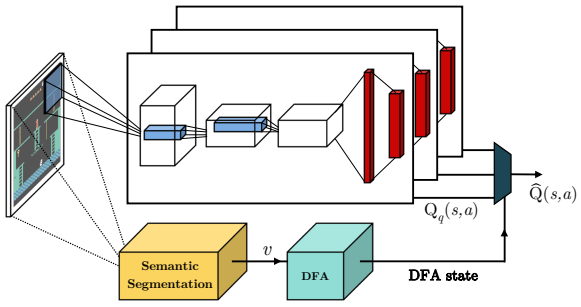


Figure 3: DeepSynth for Montezuma's Revenge: each DQN module is forced by the DFA to focus on the correlation of semantically distinct objects. The input to the first layer of the DQN modules is the input image which is convolved by 32 filters of 8×8 with stride 4 and a ReLU. The second hidden layer 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 (Mnih et al. 2015).

uses the following reward transformation:

$$r^T = \widehat{r} + \mu\, r^i, \tag{1}$$

where $\widehat{r}$ is the extrinsic reward, $\mu > 0$ 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. 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 $\widehat{r}_{door}$. Note that the lack of intrinsic motivation as shown in Section 5, prevents other methods, e.g. (Toro Icarte et al. 2019; Rens et al. 2020; Gaon and 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, states consist of raw pixel images. Each state is a stack of four consecutive frames $84 \times 84 \times 4$ that are preprocessed to reduce input dimensionality (Mnih et al. 2015). 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`$\}$ (Fig. 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, e.g., the above label is $L(s)$=$\{$`rope`$\}$.

Given this labelling function, Tracing (Step 1) records the sequence of detected objects $L(s_i)L(s_{i+1})\ldots$ as the agent explores the MDP. The labelling function, as per Definition 2.1, is a mapping from the state space to the power

set of objects in the vocabulary $L : \mathcal{S} \to 2^{\Sigma}$ and thus, the label of a state could be the empty set or a set of objects.

All transitions with their corresponding labels are stored as 5-tuples $\langle s, a, s', r^T, L(s') \rangle$, where s is the current state, a is the executed action, s' is the resulting state, r^T is the total reward received after performing action a at state s, and $L(s')$ is the label corresponding to the set of atomic propositions in Σ that hold in state s'. The set of past experiences is called the experience replay buffer $\mathcal{E}$. The exploration process generates a set of *traces*, defined as follows:

Definition 4.1 (Trace) *In a general MDP $\mathfrak{M}$, and over a finite path $\rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n$, a trace σ is defined as a sequence of labels $\sigma = v_1, v_2, \dots, v_n$, where $v_i = L(s_i)$ is a trace event.*

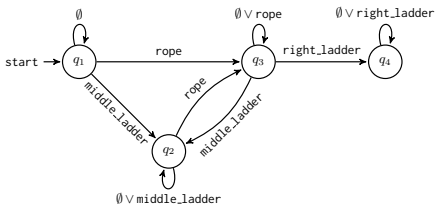
The set of traces associated with $\mathcal{E}$ is denoted by $\mathcal{T}$. The tracing scheme is the Tracing box in Fig. 1.

Synth (Step 2 in Fig. 1): The automata synthesis algorithm described in Section 3 is used to generate an automaton that conforms to the trace sequences generated by Tracing (Step 1). Given a trace sequence $\sigma = v_1, v_2, \dots, v_n$, 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) with the alphabet $\Sigma_{\mathfrak{A}}$, where a symbol of the alphabet $v \in \Sigma_{\mathfrak{A}}$ is given by the labelling function $L : \mathcal{S} \to 2^{\Sigma}$ defined earlier. Thus, given a trace sequence $\sigma = v_1, v_2, \dots, v_n$ over a finite path $\rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n$ in the MDP, the symbol $v_i \in \Sigma_{\mathfrak{A}}$ is given by $v_i = L(s_i)$.

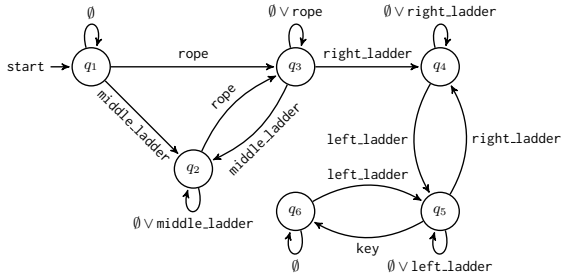
The Atari emulator provides the number of lives left in the game, which is used to reset $\Sigma_{\mathfrak{a}} \subseteq \Sigma_{\mathfrak{A}}$, where $\Sigma_{\mathfrak{a}}$ is the set of labels that appeared in the trace so far. Upon losing a life, $\Sigma_{\mathfrak{a}}$ is reset to the empty set.

Definition 4.2 (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 $\delta : \mathcal{Q} \times \Sigma_{\mathfrak{A}} \to \mathcal{Q}$ is the transition function.*

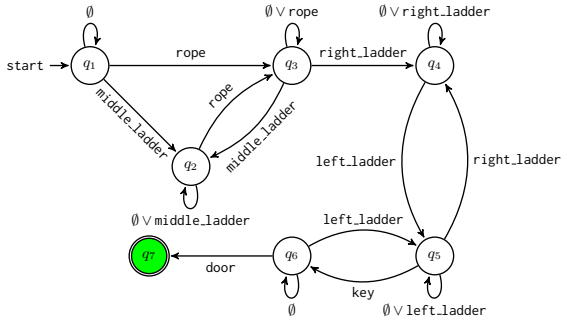
Let $\Sigma^*_{\mathfrak{A}}$ be the set of all finite words over $\Sigma_{\mathfrak{A}}$. A finite word $w = v_1, v_2, \dots, v_m \in \Sigma^*_{\mathfrak{A}}$ is accepted by a DFA $\mathfrak{A}$ if there exists a finite run $\theta \in \mathcal{Q}^*$ starting from $\theta_0 = q_0$, where $\theta_{i+1} = \delta(\theta_i, v_{i+1})$ for $i \geq 0$ and $\theta_m \in F$. Given the collected traces $\mathcal{T}$ we construct a DFA using the method described in Section 3. The algorithm first divides the trace into segments using a sliding window of size equal to the hyper-parameter w introduced earlier. This determines the size of the input to the search procedure, and consequently the algorithm runtime. Note that choosing $w = 1$ will not capture any sequential behaviour. In DeepSynth, we would like to have a value for w that results in the smallest input size. In our experiments, we tried different values for w in increasing order, ranging within $1 < w \leq |\sigma|$, and have obtained the same automaton in all setups.



(a) The right ladder is often discovered by random exploration



(b) The key is found with an extrinsic reward of $\widehat{r}_{key} = +100$



(c) The door is unlocked with an extrinsic reward of $\widehat{r}_{door} = +300$

Figure 4: Illustration of the evolution of the automaton synthesised for Montezuma's Revenge. 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. Such observations are difficult to extract from other hierarchy representations, e.g. LSTMs.

As discussed in Section 3, the hyper-parameter l controls the degree of generalisation in the learnt automaton. Learning exact automata from trace data is known to be NP-complete (Gold 1978). Thus, a higher value for l increases the algorithm runtime. We optimise over the hyper-parameters and choose $w = 3$ and $l = 2$ as the best fit for our setting. This ensures that the automata synthesis problem is not too complex for the synthesis algorithm to solve but at the same time it does not over-generalise to fit the trace.

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, from the set of succinct DFAs obtained earlier. Fig. 4 gives three exemplars of 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 in Fig. 4.a

via random exploration. However, reaching the key and further the doors as in Fig. 4.b and Fig. 4.c is challenging and is achieved by DeepSynth using a hierarchical curiosity-driven learning method described next. The automata synthesis is the Synth box in Fig. 1 and implementation details can be found in (Hasanbeig et al. 2019b).

Deep Temporal Neural Fitted RL (Step 3 in Fig. 1): We propose a deep-RL-based architecture inspired by DQN (Mnih et al. 2015) and Neural Fitted Q-iteration (NFQ) (Riedmiller 2005) when the input is in vector form, not a raw image. DeepSynth is able to synthesise a policy whose traces are accepted by the DFA and it encourages the agent to explore under the DFA guidance. More importantly, the agent is guided and encouraged to expand the DFA towards task satisfaction.

Given the constructed DFA, at each time step during the learning episode, if a new label is observed during exploration, the intrinsic reward in (1) becomes positive. Namely,

$$R^i(s,a) = \begin{cases} \eta & \text{if } L(s') \notin \Sigma_{\mathfrak{a}}, \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

where η is an arbitrarily finite and positive reward, and $\Sigma_{\mathfrak{a}}$, as discussed in the Synth step, is the set of labels 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.

Theorem 4.1 (Formalisation of the Intrinsic Reward) *The optimal policies are invariant under the reward transformation in* (1) *and* (2)*.*

The proof of Theorem 4.1 is presented in (Hasanbeig et al. 2019b). In the following, in order to explain the core ideas underpinning the algorithm, we temporarily assume that the MDP graph 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 non-Markovian task into a set of Markovian, history-independent sub-goals.

Definition 4.3 (Product MDP) *Given an MDP* $\mathfrak{M} = (\mathcal{S}, \mathcal{A}, s_0, P, \Sigma)$ *and a DFA* $\mathfrak{A} = (\mathcal{Q}, q_0, \Sigma_{\mathfrak{A}}, F, \delta)$*, the product MDP is defined as* $(\mathfrak{M} \otimes \mathfrak{A}) = \mathfrak{M}_{\mathfrak{A}} = (\mathcal{S}^{\otimes}, \mathcal{A}, s_0^{\otimes}, P^{\otimes}, \Sigma^{\otimes}, F^{\otimes})$*, 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 MDP states with the DFA states by means of the product MDP, we can evaluate the satisfaction of the associated high-level task. Most importantly, as shown in (Brafman et al. 2018), for any MDP $\mathfrak{M}$ with finite-horizon non-Markovian reward, e.g. Montezuma's Revenge, there exists a Markov reward MDP $\mathfrak{M}' = (\mathcal{S}, \mathcal{A}, s_0, P, \Sigma)$ 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, the non-Markovianity of the extrinsic reward is resolved by synchronising the DFA with the original MDP, where the DFA represents the history of state labels that has led to that reward.

Note that the DFA transitions can be executed 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.

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 (Fig. 3 and Deep in Fig. 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. The modules are interconnected, in the sense that modules act as a global *hybrid* deep RL architecture to approximate the Q-function in the product MDP. As explained in the following, this allows the agent to jump from one sub-task to another by just switching between these modules as prescribed by the DFA.

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 weights 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 Fig. 4.c, to the initial module B_{q_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. When the state is in vector form and no convolutional layer is involved we resort to NFQ deep RL modules instead of DQN modules.

5 Experimental Results

Benchmarks and Setup: We evaluate and compare the performance of DeepSynth with DQN on a comprehensive set of benchmarks, given in Table 1. The Minecraft environment

experiment	$\|S\|$	task DFA	synth DFA	prod. MDP	max sat. prob. at s_0	DeepSynth conv. ep.*	DQN conv. ep.*
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
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

* average number of episodes to convergence over 10 runs

Table 1: Comparison between DeepSynth and DQN

(`minecraft-tX`) taken from (Andreas et al. 2017) requires solving challenging low-level control tasks, and features highly sequential high-level goals. The two `mars-rover` benchmarks are taken from (Hasanbeig et al. 2019d), and the models have uncountably infinite state spaces. 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 and 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).

The *task DFA* column in Table 1 gives the number of states in the automaton that can be generated from the high-level objective sequences of the ground-truth task. The *synth DFA* column gives the number of states of the automaton synthesised by DeepSynth, and *prod. MDP* gives the number of states in the resulting product MDP (Definition 4.3). 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. The details of all experiments, including hyperparameters, are given in (Hasanbeig et al. 2019b, 2020a).

Results:

The training progress for Montezuma's Revenge and Task 3 in Minecraft is plotted in Fig. 5 and Fig. 6. In Fig. 6.a 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 shows an efficient back-propagation of extrinsic reward from the final high-level state to the initial state, namely once the last deep net converges the expected reward is back-propagated to the second and so

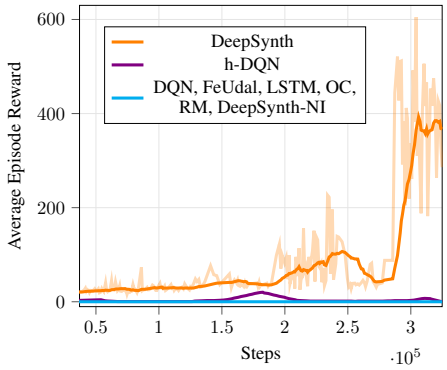
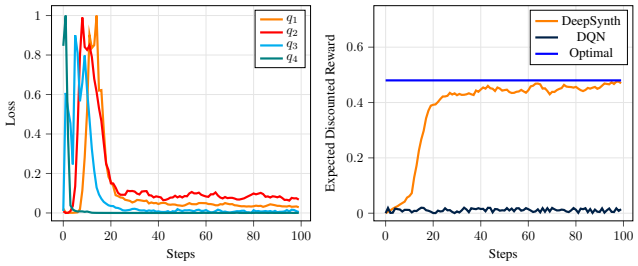


Figure 5: Average episode reward progress in Montezuma's Revenge with h-DQN (Kulkarni et al. 2016), DQN (Mnih et al. 2015), FeUdal-LSTM (Vezhnevets et al. 2017), Option-Critic (OC) (Bacon et al. 2017), inferring reward models (RM) (Toro Icarte et al. 2019; Rens et al. 2020; Gaon and Brafman 2020; Xu et al. 2020) and DeepSynth with no intrinsic reward (DeepSynth-NI). h-DQN (Kulkarni et al. 2016) finds the door but only after 2M steps. FeUdal and LSTM find the door after 100M and 200M steps respectively. DQN, OC, RM and DeepSynth-NI remain flat.



(a) (b)

Figure 6: Minecraft Task 3 Experiment: (a) Training progress with four hybrid deep NFQ modules, (b) Training progress with DeepSynth and DQN on the same training set $\mathcal{E}$.

on. Each NFQ module has 2 hidden layers and 128 ReLUs. Note that 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.

6 Conclusions

We have proposed a fully-unsupervised approach for training deep RL agents when the reward is extremely sparse and 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 easy 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. Another major contribution 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.

Acknowledgements

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 Balliol College, Jason Hu scholarship.

References

Abbeel, P.; Coates, A.; Quigley, M.; and Ng, A. Y. 2007. An Application of Reinforcement Learning to Aerobatic Helicopter Flight. In *NeurIPS*, 1–8. MIT Press.

Andreas, J.; et al. 2017. Modular Multitask Reinforcement Learning with Policy Sketches. In *ICML*, volume 70, 166–175.

Angluin, D. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75(2): 87–106.

Bacon, P.-L.; et al. 2017. The Option-Critic Architecture. In *AAAI*, 1726–1734.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *JAIR* 47: 253–279.

Berlyne, D. E. 1960. *Conflict, Arousal, and Curiosity*. McGraw-Hill Book Company.

Bertsekas, D. P.; and Shreve, S. 2004. *Stochastic Optimal Control: The Discrete-time Case*. Athena Scientific.

Biermann, A. W.; and 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.; et al. 2018. LTLf/LDLf Non-Markovian Rewards. In *AAAI*, 1771–1778.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym. *arXiv* 1606.01540.

Cai, M.; Peng, H.; Li, Z.; Gao, H.; and Kan, Z. 2021. Receding Horizon Control-Based Motion Planning With Partially Infeasible LTL Constraints. *IEEE Control Systems Letters* 5(4): 1279–1284.

Camacho, A.; Toro Icarte, R.; Klassen, T. Q.; Valenzano, R.; and McIlraith, S. A. 2019. LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning. In *IJCAI*, 6065–6073.

Chockler, H.; Kesseli, P.; Kroening, D.; and Strichman, O. 2020. Learning the Language of Software Errors. *Artificial Intelligence Research* 67: 881–903.

Cook, S.; and 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*, volume 1990. Harper & Row.

Daniel, C.; et al. 2012. Hierarchical Relative Entropy Policy Search. In *Artificial Intelligence and Statistics*, 273–281.

Davis, M.; and Putnam, H. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7(3): 201–215.

De Giacomo, G.; Favorito, M.; Iocchi, L.; and Patrizi, F. 2020. Imitation Learning over Heterogeneous Agents with Restraining Bolts. *ICAPS* 30(1): 517–521.

De Giacomo, G.; et al. 2019. Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications. In *ICAPS*, volume 29, 128–136.

Ecoffet, A.; Huizinga, J.; Lehman, J.; Stanley, K. O.; and Clune, J. 2021. First Return, Then Explore. *Nature* 590(7847): 580–586.

Fu, J.; and Topcu, U. 2014. Probably Approximately Correct MDP Learning and Control With Temporal Logic Constraints. In *Robotics: Science and Systems X*.

Fulton, N.; and Platzer, A. 2018. Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning. In *AAAI*, 6485–6492.

Furelos-Blanco, D.; Law, M.; Russo, A.; Broda, K.; and Jonsson, A. 2020. Induction of Subgoal Automata for Reinforcement Learning. In *AAAI*, 3890–3897.

Gaon, M.; and Brafman, R. 2020. Reinforcement Learning with Non-Markovian Rewards. In *AAAI*, volume 34, 3980–3987.

Glover, F.; and Laguna, M. 1998. Tabu Search. In *Handbook of Combinatorial Optimization*, volume 1–3, 2093–2229. Springer.

Gold, E. M. 1978. Complexity of Automaton Identification from Given Data. *Information and Control* 37: 302–320.

Gulwani, S. 2012. Synthesis from Examples. In *WAMBSE*.

Hasanbeig, M.; Kantaros, Y.; Abate, A.; Kroening, D.; Pappas, G. J.; and Lee, I. 2019a. Reinforcement Learning for Temporal Logic Control Synthesis with Probabilistic Satisfaction Guarantees. In *CDC*, 5338–5343. IEEE.

Hasanbeig, M.; Yogananda Jeppu, N.; Abate, A.; Melham, T.; and Kroening, D. 2019b. DeepSynth: Program Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning [Extended Version]. *arXiv* 1911.10244.

Hasanbeig, M.; Yogananda Jeppu, N.; Abate, A.; Melham, T.; and Kroening, D. 2020a. DeepSynth Code Repository. https://github.com/grockious/deepsynth.

Hasanbeig, M.; et al. 2018. Logically-Constrained Reinforcement Learning. *arXiv* 1801.08099.

Hasanbeig, M.; et al. 2019c. Certified Reinforcement Learning with Logic Guidance. *arXiv* 1902.00778.

Hasanbeig, M.; et al. 2019d. Logically-Constrained Neural Fitted Q-Iteration. In *AAMAS*, 2012–2014. International Foundation for Autonomous Agents and Multiagent Systems.

Hasanbeig, M.; et al. 2020b. Cautious Reinforcement Learning with Logical Constraints. In *AAMAS*, 483–491. International Foundation for Autonomous Agents and Multiagent Systems.

Hasanbeig, M.; et al. 2020c. Deep Reinforcement Learning with Temporal Logics. In *FORMATS*, 1–22. Springer.

Heule, M. J. H.; and 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.; and Chen, L.-C. 2019. SegSort: Segmentation by Discriminative Sorting of Segments. In *ICCV*, 7334–7344.

Jeppu, N. Y. 2020. *Trace2Model Github repository*. URL https://github.com/natasha-jeppu/Trace2Model.

Jeppu, N. Y.; Melham, T.; Kroening, D.; and O'Leary, J. 2020. Learning Concise Models from Long Execution Traces. In *Design Automation Conference*, 1–6. ACM/IEEE.

Ji, X.; Henriques, J. F.; and Vedaldi, A. 2019. Invariant Information Clustering for Unsupervised Image Classification and Segmentation. In *ICCV*, 9865–9874.

Kazemi, M.; and Soudjani, S. 2020. Formal Policy Synthesis for Continuous-Space Systems via Reinforcement Learning. *arXiv* 2005.01319.

Kearns, M.; and Singh, S. 2002. Near-optimal Reinforcement Learning in Polynomial Time. *Machine learning* 49(2-3): 209–232.

Koul, A.; et al. 2019. Learning Finite State Representations of Recurrent Policy Networks. In *International Conference on Learning Representations*.

Kulkarni, T. D.; et al. 2016. Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation. In *NeurIPS*, 3675–3683.

Lang, K. J.; et al. 1998. Results of the Abbadingo One DFA Learning Competition and a new Evidence-driven State Merging Algorithm. In *Grammatical Inference*, 1–12. Springer.

Lavaei, A.; Somenzi, F.; Soudjani, S.; Trivedi, A.; and Zamani, M. 2020. Formal Controller Synthesis for Continuous-space MDPs via Model-free Reinforcement Learning. In *ICCPS*, 98–107. IEEE.

Liu, W.; Wei, L.; Sharpnack, J.; and Owens, J. D. 2019. Unsupervised Object Segmentation with Explicit Localization Module. *arXiv* 1911.09228.

Mao, H.; Alizadeh, M.; Menache, I.; and Kandula, S. 2016. Resource Management with Deep Reinforcement Learning. In *ACM Workshop on Networks*, 50–56. ACM.

Memarian, F.; Xu, Z.; Wu, B.; Wen, M.; and Topcu, U. 2020. Active Task-Inference-Guided Deep Inverse Reinforcement Learning. In *59th IEEE Conference on Decision and Control, CDC*, 1932–1938. IEEE.

Mnih, V.; et al. 2015. Human-level Control Through Deep Reinforcement Learning. *Nature* 518(7540): 529–533.

Polydoros, A. S.; and 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*. Ph.D. thesis, University of Massachusetts Amherst.

Rens, G.; and Raskin, J.-F. 2020. Learning Non-Markovian Reward Models in MDPs. *arXiv* 2001.09293.

Rens, G.; Raskin, J.-F.; Reynouad, R.; and 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*, volume 3720, 317–328. Springer.

Ryan, R. M.; and 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.; and Seshia, S. A. 2014. A Learning Based Approach to Control Synthesis of Markov Decision Processes for Linear Temporal Logic Specifications. In *CDC*, 1091–1096. IEEE.

Silver, D.; et al. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529: 484–503.

Sutton, R. S.; and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*, volume 1. MIT Press Cambridge.

Toro Icarte, R.; Klassen, T. Q.; Valenzano, R.; and McIlraith, S. A. 2018. Teaching Multiple Tasks to an RL Agent using LTL. In *AAMAS*, 452–461.

Toro Icarte, R.; Waldie, E.; Klassen, T.; Valenzano, R.; Castro, M.; and McIlraith, S. 2019. Learning Reward Machines for Partially Observable Reinforcement Learning. In *NeurIPS*, 15497–15508.

Ulyantsev, V.; Buzhinsky, I.; and Shalyto, A. 2018. Exact Finite-state Machine Identification from Scenarios and Temporal Properties. *International Journal on Software Tools for Technology Transfer* 20(1): 35–55.

Ulyantsev, V.; and Tsarev, F. 2011. Extended Finite-State Machine Induction Using SAT-Solver. In *ICMLA*, 346–349.

Vezhnevets, A.; et al. 2016. Strategic Attentive Writer for Learning Macro-actions. In *NeurIPS*, 3486–3494.

Vezhnevets, A. S.; et al. 2017. FeUdal Networks for Hierarchical Reinforcement Learning. In *ICML*, 3540–3549.

Vinyals, O.; et al. 2019. Grandmaster Level in StarCraft II Using Multi-agent Reinforcement Learning. *Nature* 575: 1–5.

Walkinshaw, N.; Bogdanov, K.; Holcombe, M.; and Salahuddin, S. 2007. Reverse Engineering State Machines by Interactive Grammar Inference. In *WCRE*, 209–218. IEEE.

Xu, Z.; Gavran, I.; Ahmad, Y.; Majumdar, R.; Neider, D.; Topcu, U.; and Wu, B. 2020. Joint Inference of Reward Machines and Policies for Reinforcement Learning. In *AAAI*, volume 30, 590–598.

Yuan, L. Z.; et al. 2019. Modular Deep Reinforcement Learning with Temporal Logic Specifications. *arXiv* 1909.11591.

Zheng, Z.; and Yang, Y. 2021. Rectifying Pseudo Label Learning via Uncertainty Estimation for Domain Adaptive Semantic Segmentation. *International Journal of Computer Vision* (to appear).

Zhou, Z.; et al. 2017. Optimizing Chemical Reactions with Deep Reinforcement Learning. *ACS Central Science* 3(12): 1337–1344.