

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

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

Computer Science Department, University of Oxford  
Oxford, United Kingdom

[hosein.hasanbeig,natasha.yogananda.jeppu,alessandro.abate,tom.melham,daniel.kroening]@cs.ox.ac.uk

## ABSTRACT

We propose 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 the attainment of an unknown *sequence* of high-level objectives. Our method employs a recently-published algorithm for synthesis of compact automata to uncover this sequential structure. We synthesise an automaton from trace data generated through exploration of the environment by the deep RL agent. A product construction is then used to enrich the state space of the environment so that generation of an optimal control policy by deep RL is guided by the discovered structure encoded in the automaton. Our experiments show that our method is able to achieve training results that are otherwise difficult with state-of-the-art RL techniques unaided by external guidance.

## 1 INTRODUCTION

Reinforcement Learning (RL) is the key enabling technique for a variety of potential applications of artificial intelligence, including robotics [37], resource management [30], traffic management [34], flight control [1], chemistry [48], and playing video games [31]. While RL is very general, many advances in the last decade have been achieved using specific instances of RL that employ a deep neural network to synthesise optimal action policies. A deep RL algorithm, AlphaGo [35], played moves in the game of Go that were initially considered glitches by human experts, but secured victory against the strongest human player. Another recent example is AlphaStar [42], which 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.

Deep RL can autonomously solve many tasks in complex environments. But tasks that feature extremely sparse, non-Markovian rewards or other long-term sequential structures are often difficult or impossible to solve with unaided RL. A well-known example is the Atari game *Montezuma’s Revenge*, in which deep RL methods such as those described in [31] fail to score even once. Interestingly, solving Montezuma’s Revenge and many other hard problems encountered in potential applications often requires learning to attain, possibly in a specific sequence, a set of high-level objectives to obtain the reward. Accomplishment of these objectives can often be identified with passing through designated states of the system, and this insight can be a lever that enables us to obtain a manageable, high-level model of the system’s behaviour and its dynamics.

In this paper we propose a new framework that infers sequential dependencies of a reward on high-level objectives automatically and exploits this to guide a deep RL agent when the reward signal is history-dependent and significantly delayed. Identification of sequential dependence on designated high-level objectives is the key to breaking down a complex task into a sequence of many 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. The automata are generated automatically, using a counterexample-guided inductive synthesis (CEGIS) algorithm [24].

The key contribution of our novel framework is that 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 deep RL. To the best of our knowledge, our work is also the first to enable deep RL to synthesise policies for unknown non-Markovian sequential objectives over *continuous-state* MDPs. Many real world problems require actions to be taken in response to high-dimensional and real-valued state observations. It is also well-known that discretisation schemes generally suffer from the trade off between accuracy and curse of dimensionality [21],

and hence it is more efficient to treat the infinite state space directly.

At the heart of our method is a deep RL algorithm that can be synchronised with an automaton and outputs a policy that follows its high-level structure. The synchronization is achieved by a certain automata-theoretic product construction, which creates a hybrid architecture for the deep neural fitted Q-iteration in RL. The technical details are given in Section 5, where we illustrate our framework with a running example. In Section 6 we evaluate the performance of our framework on a selection of benchmarks with sequential high-level structure. These experiments show that our framework is able to *automatically* infer and formalize unknown, sparse, and non-Markovian high-level reward structures to efficiently synthesise successful policies when standard deep RL fails.

## 2 RELATED WORK

The dependencies of rewards upon objectives that our method targets are often called *options* [36] in the RL literature and can, in general, be hierarchically structured. Options can be embedded into general learning algorithms to address the problem of sparse rewards. But current approaches to hierarchical RL very much depend on state representations and whether they are structured enough for a suitable reward signal to be effectively engineered manually. Hierarchical RL therefore often requires detailed supervision in the form of explicitly specified high-level actions or intermediate supervisory signals [13, 25, 28, 32, 41].

The closest line of work to ours, which aims to avoid these requirements, are recent model-based [16, 34] or model-free [14, 19, 20, 38] approaches in RL that constrain the agent with a temporal logic property. But these approaches are limited to finite-state systems, and also require a temporal property that must be known a priori. Further related work is *policy sketching* [2], which learns feasible tasks first and then stitches them together to accomplish a complex task. The key problem is that this method assumes the policy sketches are given, which may be unrealistic. Inferred automata have been used before to learn strategies for infinite two-person games, which is a more general case of an MDP, where the strategies are a function of previously visited states. Construction of Chain automata for these games provided a means to implement memory-less strategies [27]. But the Chain automata contained a significantly large number of states as compared to the  $\omega$ -automaton the game was played on. There has also been recent work on learning underlying objectives in partially observable systems which uses Q-Learning to infer a finite-state reward machine [23]. The automata synthesis based approach proposed in this paper infers underlying sequential objectives from exploration traces without any a priori knowledge.

The most common approach to synthesising automata from traces is state merging [7]. A variant of the state merge algorithm, Evidence-Driven State Merge (EDSM) [29], uses both positive and negative instances of behaviour to determine equivalence of states to be merged based on statistical

evidence. Some SAT-based approaches use SAT together with state merge to generate automata from positive and negative traces [10, 11, 22, 39, 40]. To avoid over-generalisation in the absence of labelled data, the EDSM algorithm was improved to incorporate inherent temporal behaviour [44, 45] which need to be known a priori. These algorithms do not focus on producing the most succinct automata but rather produce a good enough approximation that conforms to the trace [39], which is not the best fit for our framework. The classic automata learning technique, Angluin’s  $L^*$  algorithm [4], employs a series of equivalence and membership queries to an oracle, the results of which are used to construct the automaton. The absence of an oracle restricts the use of this algorithm in our setting.

Finally, we note that in the case of Atari games such as Montezuma’s Revenge, the dynamic nature of underlying system requires a closed-loop, online (reactive) learning algorithm to tackle the policy synthesis problem. One recently-published online approach that is effective in this setting is Go-Explore [15]. Our work, by contrast, deals with pre-recorded traces in a strictly offline manner.

## 3 BACKGROUND

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

*Definition 3.1 (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  $\mathcal{S} = \mathbb{R}^n$ , where  $\mathcal{A}$  is a set of finite actions, and  $s_0 \in \mathcal{S}$  is the initial state.  $P : \mathcal{B}(\mathbb{R}^n) \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$  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  $(\mathbb{R}^n, \mathcal{B}(\mathbb{R}^n))$  [5].  $\Sigma$  is called the *vocabulary set* in this work and is a finite set of atomic propositions for which there exists a labelling function  $L : \mathcal{S} \rightarrow 2^\Sigma$  that assigns to each state  $s \in \mathcal{S}$  a set of atomic propositions  $L(s) \in 2^\Sigma$ .

We assume that the elements of the set  $\Sigma$  are known but their assignment to the states, i.e., the labelling function  $L$ , is unknown.

*Definition 3.2 (Path).* In a general MDP  $\mathfrak{M}$ , an infinite path  $\rho$  starting at  $s_0$  is a sequence of states  $\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots$  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  $P(B | s_i, a_i) = 1$ . A finite path  $\rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n$  is a prefix of an infinite path.

At each state  $s \in \mathcal{S}$ , an agent future action is determined by a policy  $\pi$ , which is a mapping from states to a probability distribution over the actions, i.e.,  $\pi : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$ . Further, a Markovian reward function  $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  is defined to denote the immediate scalar bounded reward received by the agent from the environment after performing action  $a \in \mathcal{A}$  in state  $s \in \mathcal{S}$ .

*Definition 3.3 (Expected Discounted Reward).* For a policy  $\pi$  on an MDP  $\mathfrak{M}$ , the expected discounted reward is defined as [36]:

$$U^\pi(s) = \mathbb{E}^\pi \left[ \sum_{n=0}^{\infty} \gamma^n R(s_n, \pi(s_n)) | s_0 = s \right], \quad (1)$$

where  $\mathbb{E}^\pi[\cdot]$  denotes the expected value given that the agent follows policy  $\pi$ ,  $[0, 1]$  ( $\gamma \in [0, 1]$  when episodic) is a discount factor,  $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  is the reward, and  $s_0, \dots, s_n$  is the sequence of states generated by policy  $\pi$  up to time step  $n$ .

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

$$Q^\pi(s, a) = \mathbb{E}^\pi \left[ \sum_{n=0}^{\infty} \gamma^n R(s_n, \pi(s_n)) | s_0 = s, a_0 = a \right], \quad (2)$$

Q-Learning (QL) [47] employs the action-value function and updates state-action pair values upon visitation as in (3). QL is off-policy, namely policy  $\pi$  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 use  $Q$  only as

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

where  $0 < \mu \leq 1$  is the learning rate,  $\gamma$  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-dependant. Under mild assumptions, QL converges to a unique limit  $Q^*$ , as long as every state action pair is visited infinitely many times [47]. Once QL converges, an optimal policy can be obtained as follows

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

and  $\pi^*$  is the same optimal policy that can be alternatively generated with Bellman iterations [6] if the MDP was fully known, maximising (1) at any given state. Thus, the main goal in RL is to synthesise  $\pi^*$  when the MDP is essentially a black box.

In deep RL, the MDP in general can have a continuous state space, and thus the recursion in (3) has to be approximated by parameterising  $Q$  using  $\theta^Q$  and by minimizing the following loss function [33]:

$$\mathcal{L}(\theta^Q) = \mathbb{E}_{s \sim p^{r^\beta}} [(Q(s, a | \theta^Q) - y)^2], \quad (4)$$

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

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

In this work  $Q$  is approximated via a deep neural network architecture and the parameter set  $\theta^Q$  represents the weights of such a neural network.

## 4 AUTOMATA SYNTHESIS

The automatic inference of the high-level sequential structure that is used to guide learning is achieved using automata synthesis. The automata synthesis algorithm uses trace sequences to construct an automaton that represents the behaviour exemplified by them. The required automaton is generated using a *synthesis from examples* [18] approach as described in [24]. It is a scalable method for learning finite-state models from trace data that produce abstract, concise models. The automata synthesis framework we use makes an algorithmic improvement towards scaling to long traces by means of a segmentation approach, thus achieving automata learning in close-to-polynomial runtime as supported by empirical evidence. The resulting automaton also features informative transition-edge predicates that are not explicit in the traces, however we do not exploit this feature in the present work.

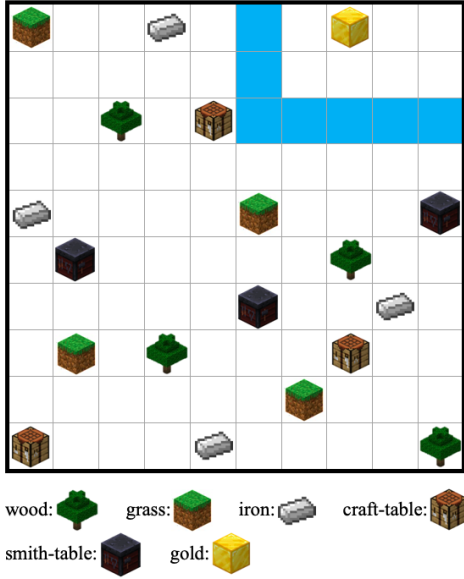
The synthesis framework takes as input a trace sequence and an optional parameter  $N$  for the number of automaton states. By default,  $N$  is initialised to two states. An overview of the synthesis framework is provided next:

- The automaton is constructed by challenging a model checker [12] with the hypothesis that there does *not* exist an  $N$ -state automaton that conforms to the input trace sequence.
- If the hypothesis check fails, the model checker in turn generates an automaton as a counterexample to the hypothesis. When the check is successful, the framework repeats the search with  $N = N + 1$  states.
- The generated automaton is refined by checking it against the trace. This compliance check eliminates transition sequences that are allowed by the learned automaton but do not appear in the trace.

The automata synthesis framework additionally employs two hyperparameters,  $w$  and  $l$ , that can be tuned based on the requirement. The hyperparameter  $w$  is used to tackle growing algorithm complexity for long trace input. The synthesis framework divides the trace input into segments using a sliding window of size  $w$  and only unique segments are used for further processing. It leverages the presence of patterns in the trace to significantly improve algorithm runtime. Multiple occurrences of these patterns in the trace require to be processed only once, thus reducing size of the input to the algorithm. The second hyperparameter  $l$  is used to control the degree of generalisation in the generated automaton. Automata generated using only ‘positive’ trace samples tend to overgeneralise [17]. This is handled by performing a compliance check of the automaton against the trace input to eliminate any transition sequences of length  $l$  that are present in the generated automaton but do not appear in the trace. Further details regarding choice of hyperparameter values is provided in Section 5.2.

## 5 DEEPSYNTH

We begin by introducing a running example of a Minecraft-inspired game, in which an agent must find a number of raw ingredients, combine them together in proper order,

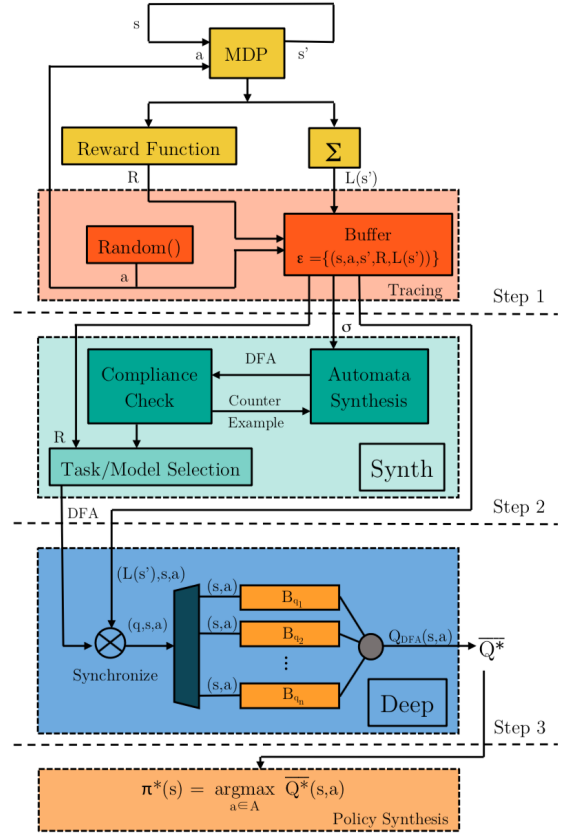


**Figure 1: Minecraft environment with given vocabulary  $\Sigma = \{\text{wood, grass, iron, craft-table, smith-table, gold}\}$**

and craft intermediate tools to alter the environment later (Fig. 1). These crafting tasks, which are taken from [3] for comparison, require the agent to execute optimal low-level actions in order to accomplish high-level objectives in proper order. The reward is given to the agent only when the whole task is achieved and the tasks are fully unknown from the beginning. A number of tasks include a long sequence of high-level objectives, and thus the associated reward is extremely sparse and non-Markovian (Table 1).

In this setup, the agent location is the MDP state  $s \in \mathcal{S}$ . At each state  $s \in \mathcal{S}$  the agent has a set of actions  $\mathcal{A} = \{\text{left, right, up, down}\}$  by which it is able to move to a neighbour state  $s' \in \mathcal{S}$  unless stopped by the boundary of  $\mathcal{S}$  or by an obstacle. Recall that we assumed the elements of the vocabulary set  $\Sigma = \{\text{wood, grass, iron, craft-table, smith-table, gold}\}$  to be known but ungrounded, namely their mapping  $L$  to the states is initially unknown to the agent. As shown in the experiments, our algorithm can handle a stochastic environment with continuous state spaces; the running example of the deterministic Minecraft environment is chosen for the sake of exposition and to enable comparison with [3].

Recall that the reward in this game is generally sparse and non-Markovian: the agent will receive a positive reward only when a correct sequence is performed in each (high-level) task. Namely, the reward  $\bar{R} : (\mathcal{S} \times \mathcal{A})^* \rightarrow \mathbb{R}$  is a function over finite state-action sequences. Further, these temporal orderings are initially unknown and the agent is not equipped with any instructions to accomplish them. In these scenarios, existing RL algorithms fail, and prior work such as [2, 19, 20] requires the temporal ordering to be known in advance. DeepSynth is a formal and intuitive framework to tackle such complex and



**Figure 2: DeepSynth framework**

practical problems. A schematic of the DeepSynth framework is provided in Fig. 2 and described next.

### 5.1 Tracing (Step 1 in Fig. 2)

The agent is made to explore the unknown MDP randomly. The tracing framework begins with a predefined vocabulary set of atomic propositions,  $\Sigma$ , that we record as the agent randomly explores the environment. All the transitions with the corresponding atomic proposition are stored as tuples  $(s, a, s', R(s, a), L(s'))$ . Here,  $s$  is the current state,  $a$  is the chosen action,  $s'$  is the resulting state,  $R(s, a)$  is the immediate reward received after performing action  $a$  at state  $s$ , and  $L(s')$  is the label corresponding to the set of atomic propositions in  $\Sigma$  that hold in state  $s'$ . The set of past experiences is called the experience replay buffer  $\mathcal{E}$ .

This random exploration process generates a set of *traces*:

*Definition 5.1 (Trace).* In a general MDP  $\mathcal{M}$ , 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  $\sigma$  is defined as a sequence of labels  $\sigma = \{v_i\}_{i=1}^n$  where  $v_i = L(s_i)$ .

As we will see later, the recorded sequences play a critical role in breaking down the non-Markovian, trace-dependent

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

Table 1: High-level sequence for each task

reward  $\bar{R}$  into a set of Markovian, history-independent rewards  $R$ . Recall that a trace-dependent reward is associated to the accomplishment of a given task: for example, performing a high-level task  $\text{wood} \rightarrow \text{iron} \rightarrow \text{craft-table}$  results in a reward  $\bar{R}_1$ , and for another high-level task such as  $\text{grass} \rightarrow \text{wood} \rightarrow \text{iron} \rightarrow \text{smith-table}$  the reward is  $\bar{R}_2$ . However, once a high-level task is completed, what the agent records into the replay buffer  $\mathcal{E}$  is just the final reward  $R$ . As stated before, along the way of performing that high-level task, the agent also records state-action pairs and their corresponding label. The sequence of labels acts as a memory for the trace-dependent reward and allows one to convert it to a Markovian reward with which we can employ RL. Further, the final reward categorises the traces into different sets, each associated to a high-level task. The tracing framework is represented by the “Tracing” box in Fig. 2.

## 5.2 Tuning Parameters for Synthesis (Step 2 in Fig. 2)

The synthesis framework described in Section 4 is used to generate an automaton that conforms to the trace sequence for a given task obtained in the previous step. Given a trace sequence  $\sigma = \{v_i\}_{i=1}^n$ , the labels  $v_i$  serve as transition predicates in the generated automaton. The synthesis framework further constrains automata construction such that no two transitions from a given state in the generated automaton have the same predicates. The automaton obtained by the synthesis framework 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  $L : \mathcal{S} \rightarrow 2^{\Sigma}$  defined earlier. So given a trace sequence  $\sigma = \{v_i\}_{i=1}^n$  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  $v_i = L(s_i)$ .

*Definition 5.2 (Deterministic Finite Automaton).* A DFA  $\mathfrak{A} = (\mathcal{Q}, q_0, \Sigma_{\mathfrak{A}}, F, \delta)$  is a state machine, 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 a 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  $\theta_m \in F$ .

For each task in the Minecraft environment, we construct a DFA from a trace sequence  $\sigma$  using the approach described in Section 4. Starting from  $N = 2$ , we systematically search for an automaton that conforms to the trace sequence. This ensures that we generate the smallest automaton that conforms to the input trace. The algorithm first divides the trace into segments using a sliding window of size equal to the hyperparameter  $w$  introduced earlier. The hyperparameter  $w$  determines the input size, and consequently the algorithm runtime. Choosing  $w = 1$  will not capture any sequential behaviour, but will only ensure that all trace events appear in the automaton. For the automata synthesis in our setting, we would like to choose a value for  $w$  that results in the smallest input size but is not trivial ( $w = 1$ ).

Once a candidate automaton is generated, the automata synthesis framework performs a compliance check by verifying if all transition sequences in the automaton of a given length, equal to the hyperparameter  $l$ , are subsequences of the trace. A higher value for  $l$  implies tighter constraints on the automaton, moving towards a more exact representation. Learning exact automata from trace data is known to be NP-complete [17]. For our experiments we incrementally tried different values for  $w$ , ranging within  $1 < w \leq |\sigma|$ , and have obtained the same automaton in all scenarios. We optimise over the hyperparameters 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 framework to solve but at the same time it does not over-generalise to fit the trace.

The obtained automaton provides insight into the behaviour of the agent as it explores the environment to obtain reward for a given task. The output of this stage is a DFA, chosen based on the task for which we wish to obtain a policy, from the set of succinct DFAs obtained earlier. The automata synthesis phase is represented by the “Synth” box in Fig. 2.

## 5.3 Deep Temporal Neural Fitted Q-iteration (Step 3 in Fig. 2)

In order to exploit the structure of the chosen DFA, we propose a deep RL scheme inspired by Neural Fitted  $Q$ -iteration (NFQ) [33] that is able to synthesize a policy whose traces are accepted by a DFA. In order to explain the core ideas underpinning the algorithm, we assume in what follows that the MDP graph and the associated transition probabilities are fully known. Later we relax these assumptions, and we

---

**Algorithm 1: Deep Temporal NFQ**


---

**input** : Automaton  $\mathfrak{A}$ , a set of transition samples  $\mathcal{E}$   
**output** : Approximated optimal  $Q$ -function:  $\bar{Q}^*$   
1 initialize all neural nets  $B_{q_i}$   
2 **repeat**  
3   **for**  $q_i = |\mathcal{Q}|$  **to** 1 **do**  
4      $\mathcal{P}_{q_i} = \{(input_l, target_l), l = 1, \dots, |\mathcal{E}_{q_i}|\}$   
5      $input_l = (s_l^\otimes, a_l)$   
6      $target_l = R(s_l^\otimes, a_l) + \gamma \max_{a'} Q(s_l^\otimes, a')$   
7     where  $(s_l^\otimes, a_l, s_l^{\otimes'}, R(s_l^\otimes, a_l), q_i) \in \mathcal{E}_{q_i}$   
8      $B_{q_i} \leftarrow \text{Adam}(\mathcal{P}_{q_i})$   
9   **end**  
10 **until** end of trial

---

stress that the algorithm can be run model-free over any black-box MDP environment.

We relate the black-box MDP and the automaton by synchronizing them *on-the-fly* (Remark 1) to create a new structure that is both compatible with RL and embraces the DFA temporal structure.

*Definition 5.3 (Product MDP).* Given an MDP  $\mathfrak{M} = (\mathcal{S}, \mathcal{A}, s_0, P, \Sigma)$  and a DFA  $\mathfrak{A} = (\mathcal{Q}, q_0, \Sigma, F, \delta)$ , 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 through the product MDP we can evaluate the satisfaction of the associated high-level task. Most importantly, it is shown in [8] that for any MDP  $\mathfrak{M}$  with non-Markovian reward (e.g. Minecraft trace-dependent reward MDP), 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  $\mathfrak{M}'$  where the corresponding states yield the same transition probabilities, and also corresponding traces have same rewards. Based on this result, [14] showed that the product MDP  $\mathfrak{M}_{\mathfrak{A}}$  is indeed  $\mathfrak{M}'$  defined above. Therefore, the non-Markovianity of the 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. This allows one to run RL over the product MDP and to find the optimal policy that maximises the corresponding Markovian reward.

**REMARK 1.** 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 automaton in the product MDP is a task segmentation and each transition between these states represents an achievable sub-task. Thus, once a DFA  $\mathfrak{A} = (\mathcal{Q}, q_0, \Sigma, F, \Delta)$  is generated, we propose a hybrid architecture of  $n = |\mathcal{Q}|$  separate deep neural networks. As in Fig. 2 in the “Deep” box, each deep net is associated with a state in the chosen task DFA and together the deep nets act as a global

hybrid deep RL architecture to approximate the  $Q$ -function in the product MDP. This allows the agent to jump from one sub-task to another by just switching between these nets.

NFQ uses a technique called *experience replay* in order to efficiently approximate the  $Q$ -function in general MDPs with continuous state spaces. Experience replay needs a set of experience samples to efficiently approximate the action-value function. Recall that we have already stored all the required transitions in the replay buffer  $\mathcal{E}$  before the synthesis phase.

For each automaton state  $q_i \in \mathcal{Q}$  the associated deep net is called  $B_{q_i} : \mathcal{S}^\otimes \times \mathcal{A} \rightarrow \mathbb{R}$ . Once the agent is at state  $s^\otimes = (s, q_i)$  the neural net  $B_{q_i}$  is active for the local  $Q$ -function approximation. Hence, the set of deep nets acts as a global hybrid  $Q$ -function approximator  $Q : \mathcal{S}^\otimes \times \mathcal{A} \rightarrow \mathbb{R}$ . Note that the neural nets are not fully decoupled. For example, assume that by taking action  $a$  in state  $s^\otimes = (s, q_i)$  the label  $v = L(s')$  has been seen and as a result the agent is moved to state  $s^{\otimes'} = (s', q_j)$  where  $q_i \neq q_j$ . According to (4) the weights of  $B_{q_i}$  are updated such that  $B_{q_i}(s^\otimes, a)$  has minimum possible error to  $R(s^\otimes, a) + \gamma \max_{a'} B_{q_j}(s^{\otimes'}, a')$ . Therefore, the value of  $B_{q_j}(s^{\otimes'}, a')$  affects  $B_{q_i}(s^\otimes, a)$ .

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 deep net  $B_{q_i}$  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  $\mathcal{E}_{q_i}$ :

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

where

$$input_l = (s_l^\otimes, a_l),$$

and

$$target_l = R(s_l^\otimes, a_l) + \gamma \max_{a' \in \mathcal{A}} Q(s_l^{\otimes'}, a'),$$

such that  $(s_l^\otimes, a_l, s_l^{\otimes'}, R(s_l^\otimes, a_l)) \in \mathcal{E}_{q_i}$ . This pattern set is then used to train the neural net  $B_{q_i}$  as in Algorithm 1. We use the Adam optimizer [26] to update the weights in each neural net (line 8). Within each fitting epoch (lines 2–10), the training schedule starts from networks that are associated with accepting states of the automaton and goes 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. Later, once the  $Q$ -value has converged to the approximated optimal  $\bar{Q}^*$ , the policy is synthesised by maximising the  $\bar{Q}^*$ .

## 6 EXPERIMENTAL RESULTS

### 6.1 Benchmarks and Setup

We evaluate the performance of our framework on a comprehensive set of benchmarks, given in Table 2. The Minecraft environment (**minecraft-tx**) involves various kinds of challenging low-level control tasks, and related joint high-level goals. We stress that rewards are provided only after the agent has completed a task in the appropriate sequence, without any intermediate goals to indicate progress towards completion (Table 1). For each task in Table 1 the generated DFAs are presented in Fig. 3–9.

experiment	$ S $	task DFA	synth DFA	prod. MDP	max sat. prob. at $s_0$	DeepSynth conv. ep.*	DeepQN 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	$\infty$	3	3	$\infty$	n/a	40	50
mars-rover-2	$\infty$	4	4	$\infty$	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	720	1	80	t/o
slp-hard-med	400	5	5	2400	1	100	t/o
slp-hard-lrg	1600	5	5	9600	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	840	0.9728	300	t/o
frozen-lake-5	400	6	6	2800	0.9722	400	t/o
frozen-lake-6	1600	6	6	11200	0.9467	450	t/o

Table 2: Learning results with DeepSynth and deep reinforcement learning

\* average number of epochs to convergence over 10 runs

The two **mars-rover** benchmarks are taken from [21], where the models have uncountably infinite (continuous) state spaces. Example **robot-surve** is adopted from [34] where the task is to visit two regions sequentially while avoiding an unsafe area. Models **slp-easy** and **slp-hard** inspired by noisy MDPs of Chapter 6 in [36]. 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. Both logics are also used in the **frozen-lake** benchmarks, where the first three are simple reachability and the last three are sequential visits of four regions, except that now there exist unsafe regions as well. The **frozen-lake** MDPs are stochastic and are adopted from [9]. Recall that all these sequences are initially unknown to the agent and the agent has to infer them as a DFA.

All simulations have been carried out on a machine with an Intel Xeon 3.5 GHz processor and 16 GB of RAM, running Ubuntu 18.

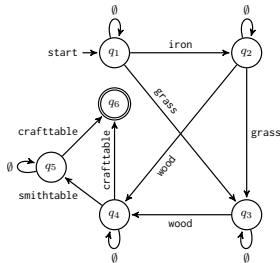


Figure 3: Task 1

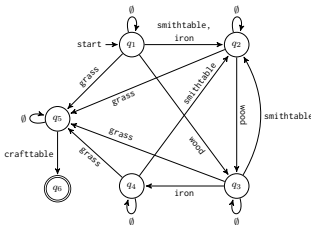


Figure 4: Task 2

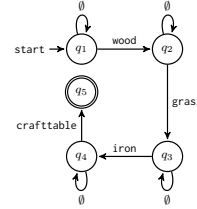


Figure 5: Task 3



Figure 6: Task 4

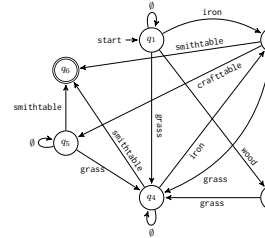


Figure 7: Task 5

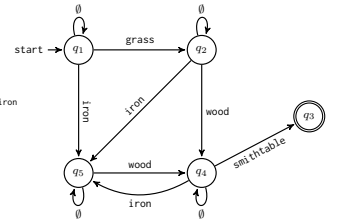


Figure 8: Task 6

As discussed before, we let the agent randomly explore the environment to find possible rewards. Each episode of exploration starts with the agent initialised. Every time we see a reward, e.g.  $R_i$ , we save the observed trace in the buffer under the task  $i$ . The *Synth* box then outputs a DFA for each of the discovered rewards. This means that even a single

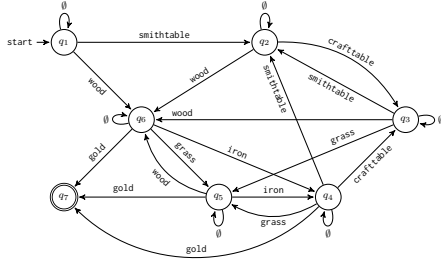


Figure 9: Task 7

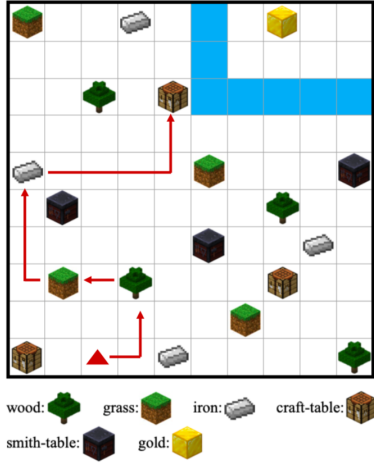


Figure 10: Example policy learnt by DeepSynth for Task 3.

occurrence of task completion is enough for our framework to find a policy that accomplishes this task.

The resulting DFA is then employed to guide the learning process. Note that there may a number of ways to accomplish a particular task in the synthesised DFAs, as in Fig. 3–9. This phenomenon however causes no harm to the learning since there is only one valid way to receive a positive reward. Hence, once the reward is back-propagated the non-optimal options automatically fall out. Additionally, since the initial position during the training is random, once the training is done at any given state the agent is able to find the optimal policy to satisfy the property.

## 6.2 Results

For the sake of exposition, we pick Task 1 and Task 3, and we train a hybrid deep architecture with six and five deep feedforward nets as in Fig. 2 to learn the correct policy. The training progress is illustrated in Fig. 11 and Fig. 12. The red line in Fig. 11 shows the very first deep net associated to  $q_1$ , the blue one is of the intermediate state  $q_2$  in the DFA and the green line is associated to  $q_4$ . This figure shows sequential back-propagation from  $q_4$  to  $q_1$ , namely once the last deep net converges the expected reward is back-propagated to

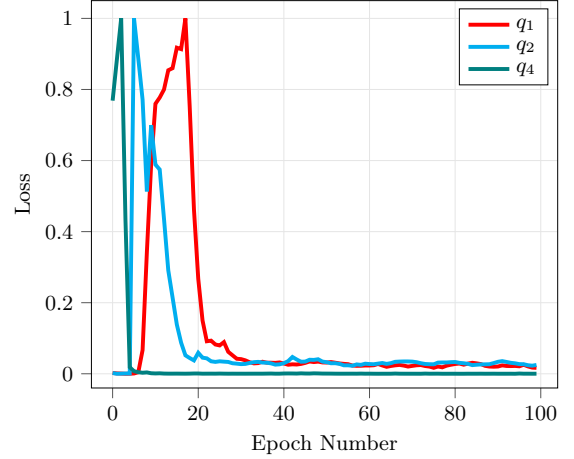


Figure 11: Training progress for Task 1 with three active hybrid deep nets coupled together

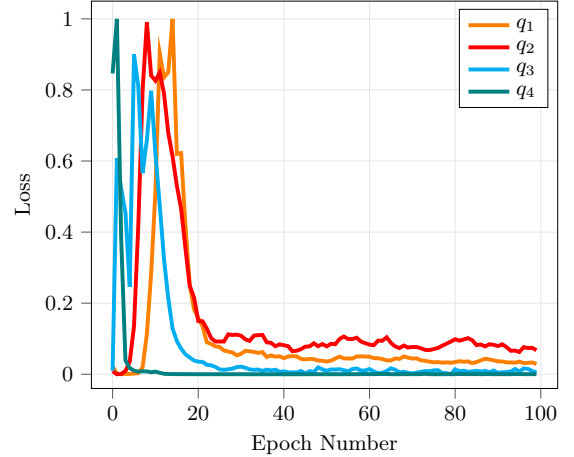


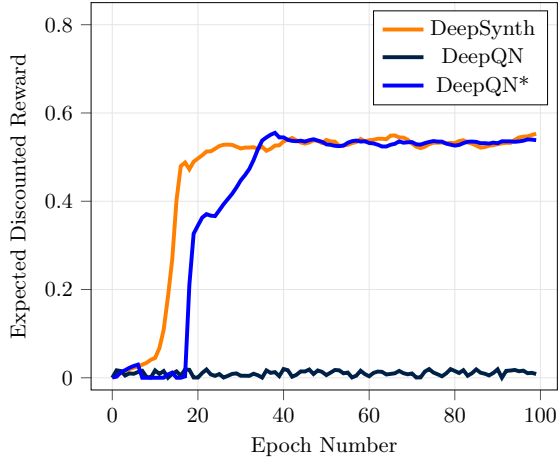
Figure 12: Training progress for Task 3 with four hybrid deep nets coupled together

the second and so on. Hence, the deep net associated to  $q_1$  converges at last. Other networks associated with automaton states within non-optimal paths remained unstable and are not shown.

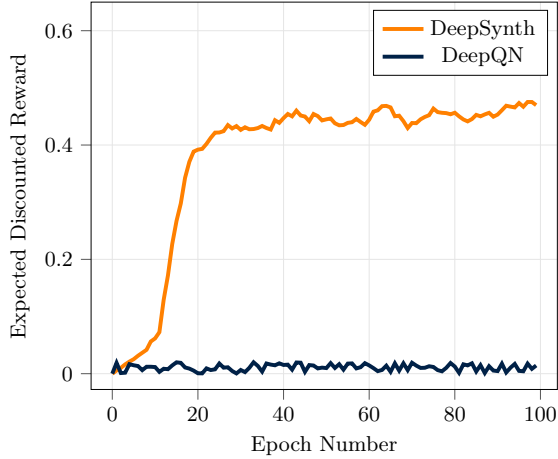
The training process for Task 3 is illustrated in Fig. 12 where the orange line shows the very first deep net associated to  $q_1$ , the red and blue ones are of the intermediate states  $q_2$  and  $q_3$  in the DFA and the green line is associated to  $q_4$ . Fig. 12 shows back-propagation from  $q_4$  to  $q_1$ , namely once the last deep net converges the expected reward is back-propagated to the second and so on. Hence, the deep net associated to  $q_1$  converges eventually.

Each  $B_{q_i}$  in this work includes 2 hidden layers and 128 ReLu units in each layer, and the training is done using the Adam optimizer with a discount factor of 0.95.





**Figure 13: Training progress for Task 1 with DeepSynth and DeepQN on the same training set  $\mathcal{E}$ . DeepQN\* is given a larger training set  $\mathcal{E}'$  to enable it to converge. The expected return is over state  $[4, 4]$  with origin being the bottom left corner.**



**Figure 14: Training progress for Task 3 with DeepSynth and DeepQN on the same training set  $\mathcal{E}$ . The expected return is over state  $[4, 4]$  with origin being the bottom left corner.**

After the training, by starting from any initial point in the crafting environment (Fig. 1), the agent is able to accomplish Task 1 and Task 3 with 100% success rate. Further, each trained neural nets can be individually employed to accomplish any arbitrary event such as grass, wood, etc. in transfer learning scenarios.

### 6.3 DeepSynth vs. DeepQN

This section compares the performance of DeepSynth with DeepQN [31] on Task 1 and Task 3. The crafting environment outputs a reward for Task 1 when the trace the agent brings

“wood” to the “craft table”. Task 3 has a more complicated sequential structure, as given in Table 1. Fig. 13 illustrates the results of training for Task 1 in DeepSynth and DeepQN. Note that with the very same training set  $\mathcal{E}$  with 4500 training samples DeepSynth is able to converge while DeepQN fails. However, we allowed DeepQN to explore more and gather enough training samples to converge. The larger training set required for this is denoted by  $\mathcal{E}'$  and contains 5500 training samples. The algorithm that employs this larger set is called DeepQN\* in Fig. 13.

Fig. 14 gives the result of training for Task 3 using DeepSynth and DeepQN where the training set  $\mathcal{E}$  has 6000 training samples. However, for Task 3 DeepQN failed to converge even after we increased the training set by an order of magnitude to 60000.

## 7 CONCLUSIONS

We have proposed a fully-unsupervised approach for training of deep RL agents when the reward is extremely sparse and non-Markovian but it features a high-level and unknown sequential structure. We *automatically* infer this high-level structure from observed exploration traces by employing techniques from automata synthesis. This allows us to recall how to achieve any high-level task, even when it has been observed only once. This high-level structure is then synchronised with a hybrid deep neural fitted  $Q$ -iteration to convert the reward into a Markovian reward and also fill in low-level policy generation. We showed that we are able to learn optimal policies that achieve complex high-level objectives using fewer training samples as compared to the DeepQN algorithm. Finally, we would like to emphasise that DeepSynth is the first deep RL architecture to synthesise policies for unknown non-Markovian sequential objectives over continuous-state MDPs. However, note that the actual satisfaction probability can not be computed when the MDP has uncountably infinite states.

## 8 ACKNOWLEDGEMENTS

This research was supported in part by a grant from the Semiconductor Research Corporation, Task 2707.001 and the Jason Hu scholarship.

## REFERENCES

- [1] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y Ng. 2007. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in Neural Information Processing Systems*. MIT Press, 1–8.
- [2] Jacob Andreas, Dan Klein, and Sergey Levine. 2017. Modular multitask reinforcement learning with policy sketches. In *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70. JMLR.org, 166–175.
- [3] Jacob Andreas, Dan Klein, and Sergey Levine. 2017. Modular Multitask Reinforcement Learning with Policy Sketches. In *ICML*, Vol. 70. 166–175.
- [4] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (Nov. 1987), 87–106. [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6)
- [5] Dimitri P Bertsekas and Steven Shreve. 2004. *Stochastic optimal control: the discrete-time case*. Athena Scientific.
- [6] Dimitri P Bertsekas and John N Tsitsiklis. 1996. *Neuro-dynamic Programming*. Vol. 1. Athena Scientific.
- [7] A. W. Biermann and J. A. Feldman. 1972. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Comput.* 21, 6 (June 1972), 592–597. <https://doi.org/10.1109/TC.1972.5009015>
- [8] Ronen I Brafman, Giuseppe De Giacomo, and Fabio Patrizi. 2018. LTLf/LDLf Non-Markovian Rewards. In *Thirty-Second AAAI Conference on Artificial Intelligence*.
- [9] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *arXiv preprint arXiv:1606.01540* (2016).
- [10] I. Buzhinsky and V. Vyatkin. 2017. Automatic Inference of Finite-State Plant Models From Traces and Temporal Properties. *IEEE Trans. Ind. Informat.* 13, 4 (Aug 2017), 1521–1530. <https://doi.org/10.1109/TII.2017.2670146>
- [11] Igor Buzhinsky and Valeriy Vyatkin. 2017. Modular plant model synthesis from behavior traces and temporal properties. *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (2017), 1–7.
- [12] Edmund M. Clarke, Jr., Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmuth Veith. 2018. *Model Checking* (2nd ed.). MIT Press.
- [13] Christian Daniel, Gerhard Neumann, and Jan Peters. 2012. Hierarchical relative entropy policy search. In *Artificial Intelligence and Statistics*. 273–281.
- [14] Giuseppe De Giacomo, Luca Iocchi, Marco Favorito, and Fabio Patrizi. 2019. Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 29. 128–136.
- [15] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. 2019. Go-Explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995* (2019).
- [16] Jie Fu and Ufuk Topcu. 2014. Probably Approximately Correct MDP Learning and Control With Temporal Logic Constraints. In *Robotics: Science and Systems X*.
- [17] E. Mark Gold. 1978. Complexity of Automaton Identification from Given Data. *Information and Control* 37 (1978), 302–320.
- [18] Sumit Gulwani. 2012. Synthesis from Examples. In *3rd Workshop on Advances in Model-Based Software Engineering (WAMBSE)*.
- [19] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. 2019. Omega-Regular Objectives in Model-Free Reinforcement Learning. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 395–412.
- [20] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. 2018. Logically-Constrained Reinforcement Learning. *arXiv preprint arXiv:1801.08099* (2018).
- [21] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. 2019. Logically-Constrained Neural Fitted Q-Iteration. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. International Foundation for Autonomous Agents and Multiagent Systems, 2012–2014.
- [22] Marijn J. H. Heule and Sicco Verwer. 2013. Software Model Synthesis Using Satisfiability Solvers. *Empirical Software Engineering* 18, 4 (01 Aug 2013), 825–856. <https://doi.org/10.1007/s10664-012-9222-z>
- [23] Rodrigo Toro Icarte, Ethan Waldie, Toryn Klassen, Rick Valenzano, Margarita Castro, and Sheila McIlraith. 2019. Learning Reward Machines for Partially Observable Reinforcement Learning. In *Advances in Neural Information Processing Systems*. 15497–15508.
- [24] Natasha Yogananda Jeppu, Tom Melham, Daniel Kroening, and John O’Leary. 2020. Learning Concise Models from Long Execution Traces. *arXiv preprint arXiv:2001.05230* (2020).
- [25] Michael Kearns and Satinder Singh. 2002. Near-optimal reinforcement learning in polynomial time. *Machine learning* 49, 2-3 (2002), 209–232.
- [26] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. *ICLR* (2015).
- [27] Sriram C. Krishnan, Anuj Puri, Robert K. Brayton, and Pravin P. Varaiya. 1995. The Rabin Index and Chain Automata, with Applications to Automata and Games. In *Computer Aided Verification*, Pierre Wolper (Ed.). Springer, 253–266.
- [28] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. 2016. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in neural information processing systems*. 3675–3683.
- [29] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. 1998. Results of the Abbadingo One DFA Learning Competition and a new Evidence-driven State Merging Algorithm. In *Grammatical Inference*, Vasant Honavar and Giora Slutzki (Eds.). Springer, 1–12.
- [30] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *ACM Workshop on Networks*. ACM, 50–56.
- [31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level Control Through Deep Reinforcement Learning. *Nature* 518, 7540 (2015), 529–533.
- [32] Doina Precup. 2001. *Temporal abstraction in reinforcement learning*. Ph.D. Dissertation. University of Massachusetts Amherst.
- [33] Martin Riedmiller. 2005. Neural Fitted Q iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method. In *ECML*, Vol. 3720. Springer, 317–328.
- [34] Dorsa Sadigh, Eric S Kim, Samuel Coogan, S Shankar Sastry, and Sanjit A Seshia. 2014. A learning based approach to control synthesis of Markov decision processes for linear temporal logic specifications. In *CDC*. IEEE, 1091–1096.
- [35] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (2016), 484–503. <http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html>
- [36] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.
- [37] Richard S Sutton, Doina Precup, and Satinder Singh. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence* 112, 1-2 (1999), 181–211.
- [38] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. 2018. Teaching Multiple Tasks to an RL Agent using LTL. In *AAMAS*. 452–461.
- [39] Vladimir Ulyantsev, Igor Buzhinsky, and Anatoly Shalyto. 2016. Exact Finite-State Machine Identification from Scenarios and Temporal Properties. *CoRR* abs/1601.06945 (2016). [arXiv:1601.06945](https://arxiv.org/abs/1601.06945)
- [40] V. Ulyantsev and F. Tsarev. 2011. Extended Finite-State Machine Induction Using SAT-Solver. In *International Conference on Machine Learning and Applications and Workshops*. 346–349. <https://doi.org/10.1109/ICMLA.2011.166>
- [41] Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John Agapiou, et al. 2016. Strategic attentive writer for learning macro-actions. In *Advances in neural information processing systems*. 3486–3494.
- [42] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* (2019), 1–5.

- [43] Neil Walkinshaw. 2018. *MINT framework Github repository*. <https://github.com/neilwalkinshaw/mintframework>
- [44] N. Walkinshaw and K. Bogdanov. 2008. Inferring Finite-State Models with Temporal Constraints. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 248–257. <https://doi.org/10.1109/ASE.2008.35>
- [45] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. 2007. Reverse Engineering State Machines by Interactive Grammar Inference. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE '07)*. IEEE Computer Society, 209–218. <https://doi.org/10.1109/WCRE.2007.45>
- [46] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *Empirical Software Engineering* 21, 3 (01 Jun 2016), 811–853.
- [47] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.
- [48] Zhenpeng Zhou, Xiaocheng Li, and Richard N Zare. 2017. Optimizing chemical reactions with deep reinforcement learning. *ACS Central Science* 3, 12 (2017), 1337–1344.

## APPENDIX

### A THE AUTOMATA SYNTHESIS FRAMEWORK VS. STATE MERGE

In this section we compare the automata synthesis framework we use to algorithms based on state merging. State merge algorithms are the established approaches in 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 [7] algorithm for state merging, several alternatives to determine state equivalence have been proposed over the years [44]. For our experiment we used the MINT (Model INference Technique) [43] tool that implements different variations of the state merge algorithm, including data classifiers [46] 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. We observed that although MINT is faster, the automata generated by the tool are either too big (large number of states) or are over generalised (sometimes having a single state) depending on the tool configurations. As examples the smallest model that best fit Task 5 traces includes 49 states (Fig. 15), and 14 states for Task 6 (Fig. 16). Here, the ‘start’ label signifies the beginning of a new trace obtained from another instance of random exploration. For the proposed DeepSynth framework we require 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 models that fit a given trace, we prefer using the automata synthesis framework described in this work.

### B AUTOMATA SYNTHESISED FOR OTHER BENCHMARKS

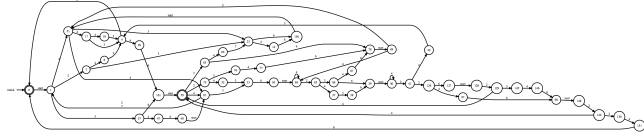


Figure 15: Best fit model for Task 5 generated by the MINT tool.

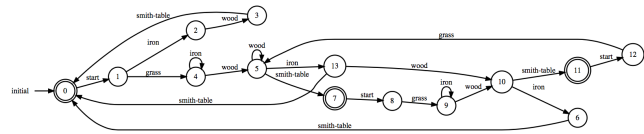


Figure 16: Best fit model for Task 6 generated by the MINT tool.

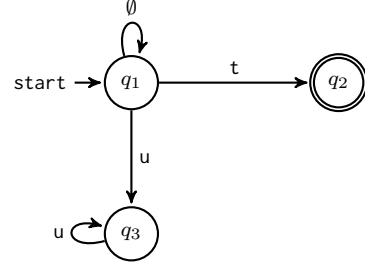


Figure 17: Automata synthesised for mars-rover-1 benchmark.

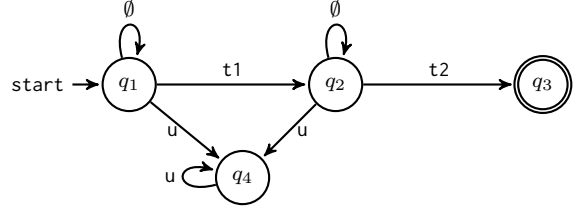


Figure 18: Automata synthesised for mars-rover-2 benchmark.

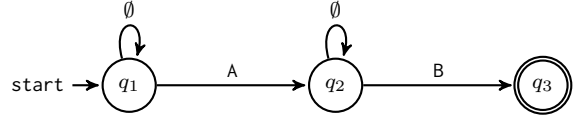


Figure 19: Automata synthesised for robot-survey benchmark.

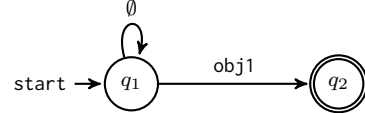


Figure 20: Automata synthesised for slp-easy benchmarks.

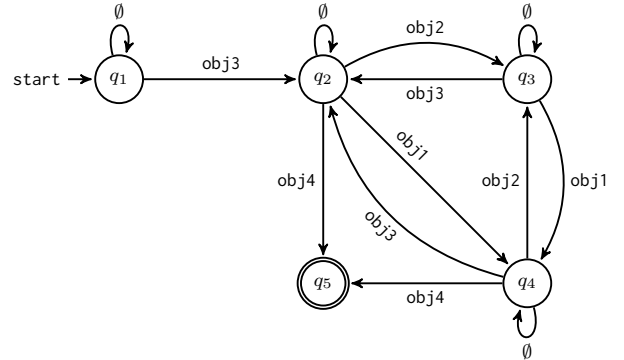


Figure 21: Automata synthesised for slp-hard benchmarks.

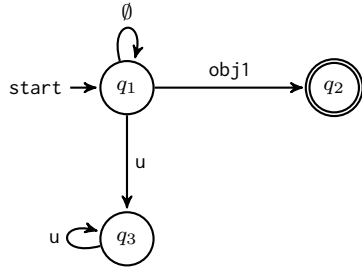


Figure 22: Automata synthesised for frozen-lake-1,2,3 benchmarks.

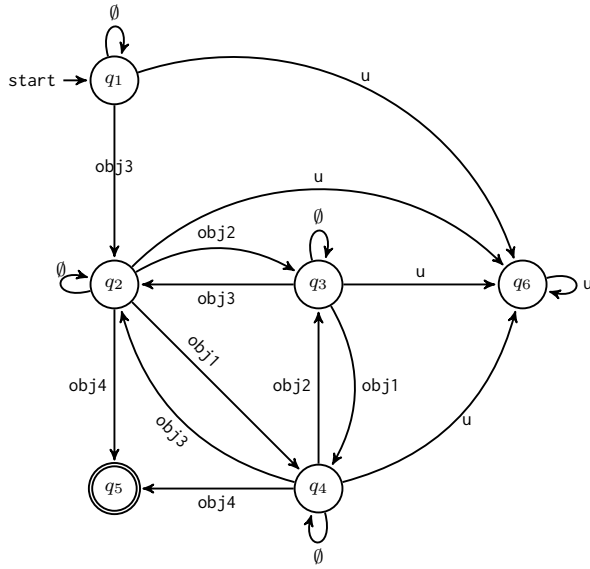


Figure 23: Automata synthesised for frozen-lake-4,5,6 benchmarks.