DeepSynth: Program 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 efficient training of deep Reinforcement Learning (RL) agents when the reward is highly sparse and non-Markovian, but at the same time admits a high-level yet unknown sequential structure, as seen in a number of video games. This high-level sequential structure can be expressed as a computer program, which our method infers automatically as the RL agent explores the environment. Through this process, a high-level sequential task that occurs only rarely may nonetheless be encoded within the inferred program. A hybrid architecture for deep neural fitted $Q$-iteration is then employed to fill in low-level details and generate an optimal control policy that follows the structure of the program. Our experiments show that the agent is able to synthesise a complex program to guide the RL exploitation phase, which is otherwise difficult to achieve with state-of-the-art RL techniques.

1 INTRODUCTION
Reinforcement Learning (RL) is the key enabling technique for a very broad variety of applications of artificial intelligence, including robotics [27], resource management [20], traffic management [24], flight control [1], chemistry [34], and playing video games [21]. 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 determine the next action of the agent. A deep RL algorithm, AlphaGo [25], 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 [29], which was able to defeat 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 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 [21] fail to score even once. Interestingly, Montezuma’s Revenge and many other hard-to-solve systems encountered in potential applications exhibit a hierarchical or a temporal structure. These systems are composed of interrelated sub-systems, that in turn might have their own sub-systems. This insight can be a lever that enables us to obtain a manageable yet predictive model of their behaviour and their dynamics.

These hierarchical interrelations, which are often called options [26] in RL, can be embedded into general learning algorithms to address such problems. But current approaches in hierarchical RL very much depend on state representations and whether they are structured enough for a suitable reward signal to be effectively engineered by hand. Hierarchical RL therefore often requires detailed supervision in the form of explicitly specified high-level actions or intermediate supervisory signals [11, 16, 18, 22, 28].

In this paper we propose a new framework that infers high-level hierarchies automatically and exploits their structural sequentiality to guide an RL agent when the reward signal is history-dependant and significantly delayed. The temporal sequentiality is the key in breaking down a complex task into a sequence of many Markovian ones. We use a computer program, modelled as an automaton, to orchestrate the sequencing of the small steps and employ a counterexample-guided inductive synthesis algorithm to infer the program automatically.

An unknown environment may require solving an unknown number of high-level tasks. So when dealing with an unknown MDP, we may encounter numerous high-level tasks that are to be accomplished. The key contribution of this work is that no matter how rare the occurrence of a high-level task is, we can still encode it as a program for use in a deep RL algorithm.

Towards this goal, we developed a deep RL scheme that can be synchronised with a computer program and outputs a policy that follows the high-level structure of the program. We emphasise that the segmented task can be used in learning
transfer scenarios, since the inferred program is a formal, un-grounded and symbolic representation of the task and its components. So any segment of the proposed deep RL can be used as a trained module in other related environments.

The closest line of work to ours are the model-based [13, 24] or model-free [15] approaches in RL that constrain the agent with a temporal logic property. But these approaches are limited to finite-state MDPs, and also require a temporal property that must be known a priori. Further related work is policy sketching [3], 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.

2 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 2.1 (General MDP). The tuple \( \mathcal{M} = (S, A, s_0, P, \Sigma, L) \) is a general MDP over a set of continuous states \( S = \mathbb{R}^n \), where \( A \) is a set of the finite actions, and \( s_0 \in S \) is the initial state. \( P : \mathcal{B}(\mathbb{R}^n) \times S \times A \to [0, 1] \) is a Borel-measurable conditional transition kernel that assigns to any pair of state \( s \in S \) and action \( a \in A \) a probability measure \( P(\cdot | s, a) \) on the Borel Space \( (\mathbb{R}^n, \mathcal{B}(\mathbb{R}^n)) \). \( \Sigma \) is called the vocabulary set in this work and is essentially a finite set of atomic propositions for which there exists a labelling function \( L : S \to \Sigma \) that assigns to each state \( s \in S \) an atomic proposition \( L(s) \in \Sigma \).

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. Note that states in the MDP \( \mathcal{M} \) may be associated with none of the elements in \( \Sigma \), i.e. for some states \( s \) we may have \( L(s) = \emptyset \).

Definition 2.2 (Path). In a general MDP \( \mathcal{M} \), an infinite path \( \rho \) starting at \( s_0 \) is a sequence of states \( \rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \) such that every transition \( s_i \xrightarrow{a} s_{i+1} \) is possible in \( \mathcal{M} \), i.e., \( s_{i+1} \) belongs to the smallest Borel set \( B \) such that \( P(B|s_i, a_i) = 1 \). A finite path \( \rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n \) is a prefix of an infinite path.

At each state \( s \in S \), 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., \( \pi : S \to \mathcal{P}(A) \). Further, a Markovian reward function \( R : S \times A \to \mathbb{R} \) is defined to denote the immediate scalar bounded reward received by the agent from the environment after performing action \( a \in A \) in state \( s \in S \).

Definition 2.3 (Expected Discounted Reward). For a policy \( \pi \) on an MDP \( \mathcal{M} \), the expected discounted reward is defined as [26):

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

where \( \mathbb{E}^\pi[\cdot] \) denotes the expected value given that the agent follows policy \( \pi \), \( \gamma \in [0, 1] \) is a discount factor, \( R : S \times A \to \mathbb{R} \) is the reward, and \( s_0, \ldots, s_n \) 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 \( Q : S \times A \to \mathbb{R} \) as follows:

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

Q-Learning (QL) [33] 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

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

where \( 0 < \mu \leq 1 \) is the learning rate, \( \gamma \) is the discount factor, and \( s' \) is the state reached after performing action \( a \). Under mild assumptions, QL converges to a unique limit \( Q^* \), as long as every state action pair is visited infinitely many times [33]. Once QL converges, the greedy policy can be obtained as follows

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

where \( \pi^* \) is the same optimal policy that can be alternatively generated with Bellman iterations [5] 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 this work since the MDP has a continuous state space, the recursion in (3) has to be approximated by parameterising \( Q \) using \( \theta^Q \) and by minimizing the following loss function:

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

where \( \rho^\beta \) is the probability distribution of state visit over \( S \) under any arbitrary stochastic policy \( \beta \), and \( y = R(s, a) + \gamma \max_{a' \in A} Q(s', a'|\theta^Q) \). In this work \( Q \) is approximated via a deep neural network architecture and the parameter set \( \theta^Q \) represent the weights of such a neural network.

3 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, and craft intermediate tools to alter the environment later (Fig. 1). In this setup, the agent location is the MDP state \( s \in S \). At each state \( s \in S \) the agent has a set of actions \( A = \{\text{left, right, up, down}\} \) by which it is able to move to a neighbour state \( s' \in S \) unless stopped by the boundary or an obstacle. Recall that we assumed the elements of the vocabulary set \( \Sigma \) are known but they are ungrounded, namely their mapping \( L \) to the states is initially unknown to the agent.
DeepSynth: Program Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning

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

We emphasise that our algorithm can handle a stochastic environment with continuous state spaces, and the Minecraft deterministic example is chosen for the sake of exposition.

The reward in this game is sparse and non-Markovian: the agent will receive a corresponding positive reward only when a correct sequence is performed in each (high-level) task. Namely, the reward $\bar{R} : (S \times A)^* \rightarrow \mathbb{R}$ 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 [3, 15] requires the underlying sequential structure to be known in advance. DeepSynth is a formal and intuitive framework to tackle such complex yet practical problems. A schematic of the DeepSynth framework is provided in Fig. 2.

### 3.1 Program ‘Synth’esis

The automatic inference of the high-level sequence is done using program synthesis from examples [14]. The program synthesis framework begins with a predefined vocabulary of events, $\Sigma$, that we record as the agent randomly explores the environment. All the transitions with their corresponding events are stored as tuples $(s, a, s', R(s, a), v)$. 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 $v \in \Sigma$ is an event that occurred at state $s'$. The set of past experiences is called the experience replay buffer $E$.

This random exploration process generates a set of traces:

**Definition 3.1 (Trace).** In a general MDP $\mathcal{M}$, and over a finite path $\rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$, a trace $\sigma$ is defined as a sequence of events $\sigma = \{v_i\}_{i=0}^n$ from the vocabulary $\Sigma$, where $v_i = L(s_i)$.

Our program synthesis algorithm uses the generated traces to construct an automaton that represents the behaviour exemplified by them. An automaton can be expressed as a

Figure 2: DeepSynth framework

```plaintext
switch(state) {
    case q1:
        if(letter == 'a')
            next_state = q2;
        else
            next_state = dead;
        break;
    case q2:
        if(letter == 'b')
            next_state = q1;
        else
            next_state = dead;
        break;
}
```

Figure 3: Computer program for a simple automaton with three states over the alphabet $\{a, b\}$. We usually omit the transitions into the dead state.
computer program, as illustrated in Fig. 3, and in what follows we use the terms “program” and “automaton” interchangeably.

The program obtained by our synthesis framework is deterministic, because at any given point in time the agent executes a single action or event in the environment. These actions get recorded in the traces and are eventually expressed as transition predicate on the edges of the automaton. The learned program follows the standard definition of a Deterministic Finite Automaton (DFA), where the alphabet Σ is given by our vocabulary:

\[ \text{Definition 3.2 (Deterministic Finite Automaton). A DFA } \mathcal{A} = (\mathcal{Q}, q_0, \Sigma, F, \delta) \text{ is a state machine, where } \mathcal{Q} \text{ is a finite set of states, } q_0 \in \mathcal{Q} \text{ is the initial state, } \Sigma \text{ is the vocabulary, } F \subseteq \mathcal{Q} \text{ is the set of accepting states, and } \delta : \mathcal{Q} \times \Sigma \rightarrow \mathcal{Q} \text{ is a transition function.} \]

Let \( \Sigma^* \) be the set of all finite words over \( \Sigma \). A finite word \( w = \sigma_1 \sigma_2 \ldots \sigma_m \in \Sigma^* \) is accepted by a DFA \( \mathcal{A} \) if there exists a finite run \( \theta \in \mathcal{Q}^* \) starting from \( q_0 \) where \( \delta(\theta_1, \sigma_1) = \delta(\theta_1, \sigma_2) \) for \( i \geq 1 \) and \( \delta_{m} \in F \).

For each task in the Minecraft environment, we construct a DFA from a trace sequence using an approach based on Counterexample Guided Inductive Synthesis (CEGIS) [2]. This is an iterative process whereby candidate programs are generated based on a set of program specifications and validated against an oracle. Any counterexamples generated are used as additional constraints on the program. In our setting we generate programs, modelled as automata, using trace segments as specifications. The generated automaton is verified against an oracle, which is the full trace.

The algorithm for model synthesis from traces is provided in Algorithm 1. An overview of the framework is provided below and described in detail in the sections that follow:

- We challenge a model checker [9] with the hypothesis that there exists no DFA that conforms to a set of input event sequences.
- The model checker in turn will generate an automaton as a counterexample to the hypothesis.
- The automaton is refined by checking it against the trace sequence. This step reduces overgeneralisation by eliminating transition sequences that are allowed by the learned model but do not appear in the trace.

3.1.1 Tracing (Step 1 in Fig. 2): The agent is made to explore the unknown MDP randomly. A sequence of events is recorded during random exploration and fed as trace input to model synthesis. As we will see later, the recorded sequences play a critical role in breaking down the non-Markovian trace-dependant reward \( R \) into Markovian history-independent smaller rewards \( R \). Recall that a trace-dependant reward is associated to the accomplishment of a given task: for example, performing a high-level task wood→iron→craft-table results in a reward \( R_1 \), and for another high-level task such as grass→wood→iron→smith-table the reward is \( R_2 \). However, once a high-level task is done, what the agent records into the replay buffer \( E \) 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 event \( v \). The sequence of events \( v \) acts as a memory for the trace-dependant reward and allows us 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.

Algorithm 1: Model Synthesis

| Input | Output |
|-------|--------|
| Trace \( \sigma = \{v_1, v_2, \ldots, v_m\} \) for a given task | DFA \( \mathcal{A} = \{(q_i, v_i, q'_i)\}_{i=1}^m \) for a given task |
| \( w \leftarrow \) sliding window size | \( j \leftarrow 0 \) |
| 2. Divide \( \sigma \) into segments \( \{\sigma_1, \sigma_2, \ldots, \sigma_{n+1-w}\} \) of size \( w \) | 6. For each \( \sigma \), do |
| 3. \( N \leftarrow \) number of states of \( \mathcal{A} \) | 7. For \( k = 1 \) to \( i + w - 1 \) do |
| 4. Assume \( 0 < q_i, q'_i \leq N \), \( \forall i \) | 8. \( \delta(q_i, v_k) = q'_i \) |
| 5. \( j \leftarrow (j + 1) \) | 9. Assume \( q_{i+1} = q'_j \) |
| 11. End | 12. End |
| 13. \( \text{wrong}_\text{transition} \leftarrow \text{false} \) | 14. If \( \exists i, j \in \{1, \ldots, m\} : (q_i = q_j \land v_i = v_j \land q'_i \neq q'_j) \) then |
| 15. \( \text{wrong}_\text{transition} \leftarrow \text{true} \) | 16. End |
| 17. Run CBMC with assertion (\( \text{wrong}_\text{transition} = \text{true} \)) | 18. If \( SAT \) then |
| 19. \( N \leftarrow \{N + 1\} \) | 20. Go to 4 |
| 21. Else if \( \mathcal{A} \neq \sigma \) then | 22. Add counterexample to constraints |
| 23. Go to 17 | 24. Else |
| 25. Return \( \mathcal{A} \) | 26. End |

3.1.2 Model Synthesis (Step 2 in Fig. 2): For each high-level task, our program construction algorithm takes as input a trace sequence of events \( \sigma = \{v_i\}_{i=1}^m \) where \( v_i \in \Sigma \). The DFA to be constructed is represented as a transition array \( \mathcal{A} = \{(q_i, v_i, q'_i)\}_{i=1}^m \) where each transition is a triple comprising the state \( q_i \) from which the transition occurs, a transition event \( v_i \in \Sigma \) and the next state \( q'_i \). The sequence of events \( \sigma \) is divided into segments using a sliding window of size \( w \), a hyperparameter that can be tuned. Unique segments are used for further processing (line 2). For our experiments we incrementally tried different values for \( w \) between \( 1 < w \leq |\sigma| \) and obtained the same automaton in all scenarios. The hyperparameter \( w \) determines the input size, and consequently the algorithm runtime. Choosing \( w = 1 \) will not capture any sequential behaviour but only ensures that all trace events appear in the automaton. For model learning, we would like to choose a value for \( w \) that results in the smallest input size but is not trivial (\( w = 1 \)). The strategy we adopt for our experiments is to fix a segment length \( w = 2 \) to ensure quick results. The result of segmentation is the set of all unique subsequences of \( \sigma \) of length \( w \). The sliding window significantly improves runtime for model synthesis by leveraging
DeepSynth: Program Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning

**Algorithm 2: Deep Temporal NFQ**

```
input : Automaton $A$, a set of transition samples $E$
output : Approximated optimal $Q$-function: $Q^*$
1. initialize all neural nets $B_{q_i}$
2. repeat
   3. for $q_i = [0]$ to 1 do
      4. $P_{q_i} = \{(input_t, target_t), i = 1, ..., |E_{q_i}|\}$
      5. $input_t = (s_i^{\infty}, a_i)$
      6. $target_t = R(s_i^{\infty}, a_i) + \gamma \max_{a' \in A} Q(s_i^{\infty}, a')$
      7. where $(s_i^{\infty}, a_i, s_i^{\infty'}, R(s_i^{\infty}, a_i), q_i) \in E_{q_i}$
      8. $B_{q_i} \leftarrow \text{Adam}(P_{q_i})$
   end
3. until end of trial
```

the presence of repeating patterns in the trace. These segments are used to constrain the automaton to always include the corresponding transition events in the model generated (lines 6–12).

To generate the model, we search systematically for an $N$-state DFA whose behaviours include all the unique segments identified above. We set the number of states for the DFA to be generated by restricting the state variables of $A$ to take values between 1 and $N$ (line 4). To ensure determinism of the automaton, we add an additional constraint: given a state $q$, no two transitions from $q$ are labelled with the same event. A wrong_transition flag is set to true when any trace constraint is violated.

The program, together with the assertion wrong_transition = true, is then fed to the model checker CBMC [8]. The assertion (line 17) is a means of querying the model checker to check if the aforementioned assertion always holds. If SAT (line 18) indicates that for all assignment of state values to state variables $q_i$ and $q'_i$ of $A$, wrong_transition is always true and hence the assertion is satisfied. A counterexample to the assertion is an assignment of state values to state variables of $A$ that form the required $N$-state automaton that satisfies the non-determinism constraint. If no counterexample exists, there is no $N$-state automaton that satisfies our constraints; in this case we increment $N$ and repeat the search. We begin model synthesis with $N = 2$ and increase the number of states by 1 if such an automaton cannot be synthesised. This ensures that we synthesise the smallest automaton that contains all event subsequences of $\sigma$ of length $w$.

Once CBMC has generated a candidate model, we check whether the model allows any event sequences that are infeasible according to the trace (line 21). An event sequence is said to be infeasible if it is not a subsequence of $\sigma$. We check if all transition sequences in the model of a given length are subsequences of the trace. We have used a length of 2 so that it is not too complex for the model checker to solve and we still get a model that is not over-generalised to fit the trace. We encode infeasible sequences as additional constraints on the transition system and repeat the search for an automaton as described above. This counterexample refinement loop acts to glean further information from the trace.

The automaton obtained gives insight into the behaviour of the agent as it explores the environment to obtain reward for a given task. In order to get the most succinct sequence of events that guide the agent towards the reward, we extract the shortest path between the start state of the automaton and its accepting state. We use Dijkstra’s shortest path algorithm [10] to achieve this. The output of this stage is a DFA, chosen based on the task we wish to obtain a policy for, from the set of succinct DFAs obtained earlier. The model synthesis phase is represented by the “Synth” box in Fig. 2.

### 3.2 ‘Deep’ Temporal Neural Fitted Q-iteration

In order to exploit the structure of the chosen DFA, we propose a deep RL scheme inspired by Neural Fitted Q-iteration (NFQ) [23] that is able to synthesize a policy whose traces are accepted by a DFA (Step 3 in Fig. 2). In order to explain the core ideas of 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 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 to create a new structure that is both compatible with RL and embraces the DFA temporal structure.

We have observed that there are a number of ways to accomplish a given task over a single DFA; consider Fig. 5 and Fig. 8 as examples. However, we only care about the shortest paths possible, i.e. lowest number of labels to be read. This rules out a number of cases in the resulting DFAs but since the reward is sparse in other cases we might end up having two or more options as the shortest path with some redundant intermediate events (Fig. 4 or Fig. 8). We eliminate all transitions in the DFA that are not in the shortest path and in the rest of the paper by DFA we refer to the pruned automaton.

**Definition 3.3 (Product MDP).** Given an MDP $M = (S, A, s_0, P, \Sigma)$ and a DFA $A = (Q, q_0, \Sigma, F, \Delta)$, the product MDP is defined as $(M \ast A) = M_{\ast A} = (S^\infty, A, s_0^\infty, P^\ast, \Sigma^\ast, F^\ast)$, where $S^\infty = S \times Q$, $s_0^\infty = (s_0, q_0)$, $\Sigma^\ast = \Sigma$, and $F^\ast = S \times F$. The transition kernel $P^\ast$ is such that given the current state $(s_i, q_i)$ and action $a$, the new state is $(s_j, q_j)$, where $s_j \sim P(\cdot|s_i, a)$ and $q_j = \Delta(q_i, v_j)$.

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 [7], that for any MDP $M$ with non-Markovian reward (e.g. Minecraft trace-dependant reward MDP), there exists a Markov reward MDP $M'$ that is equivalent to $M$ such that the states of $M'$ can be mapped into those of $M$ where the corresponding states yield the same transition probabilities, and also corresponding traces have same rewards. Based on this result, [12] showed that the product MDP $M_{\ast A}$ is indeed $M'$ 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 events that led to that reward. This allows us
to run RL over the product MDP and find the optimal policy that maximises the now-Markovian reward.

Note that the DFA transitions can be executed just by observing the events 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 $A = (Q, q_0, \Sigma, F, \Delta)$ is generated, we propose a hybrid architecture of $n = |Q|$ separate deep neural network. 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 acts 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 space. 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 $E$ before the synthesis phase.

For each automaton state $q_i \in Q$ the associated deep net is called $B_{q_i} : S^\otimes \times A \rightarrow R$. 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 $Q : S^\otimes \times A \rightarrow R$. 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 event $v_i$ has happened 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 $B_{q_i}(s^\otimes, a)$.
DeepSynth: Program Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning

| task | sequence |
|------|----------|
| Task1 | Σ* get wood Σ* use craft table |
| Task2 | Σ* get grass Σ* use craft table |
| Task3 | Σ* get wood Σ* get grass |
| Task4 | Σ* get wood Σ* use smith table |
| Task5 | Σ* get grass Σ* use smith table |
| Task6 | Σ* get iron Σ* get wood Σ* use smith table |

Table 1: Correct high-level sequence for each task

Let \( q_i \in \mathbb{Q} \) be a state in the DFA \( \mathcal{A} \). 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 \( \mathcal{E}_{q_i} \):

\[
\mathcal{P}_{q_i} = \{(input_l, target_l), l = 1,\ldots,|\mathcal{E}_{q_i}|\},
\]

where

\[
input_l = (s_i^\otimes, a_i),
\]

and

\[
target_l = R(s_i^\otimes, a_i) + \gamma \max_{a' \in A} Q(s_i^\otimes, a'),
\]

such that \( (s_i^\otimes, a_i, s_i^\otimes', R(s_i^\otimes, a_i)) \in \mathcal{E}_{q_i} \). This pattern set is then used to train the neural net \( B_{q_i} \) as in Algorithm 2.

We use the Adam optimizer [17] 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 \( Q \)-value converges to the approximated optimal \( Q^*_\pi \), the policy is synthesised by ascending the \( Q^* \).

4 EXPERIMENTS

We evaluate the performance of our framework in the crafting environment as in Fig. 1. The crafting environment involves various kinds of challenging low-level control tasks, and related joint high-level goals. Note 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). 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.

As discussed in Fig. 2, we let the agent randomly explore the crafting environment to find possible rewards. Each episode of exploration starts with the agent initialised at a random position in the environment. 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 occurrence of task \( i \) completion is enough for our framework to find a policy that accomplishes task \( i \). For each task in Table 1 the generated DFAs are presented in Fig. 4 to Fig. 9.

Interestingly, we found out there are a number of ways to accomplish a single task as in Fig. 5 or Fig. 8. However, we only care about the shortest path possible. This rules out a number of cases in the resulting DFAs but since the reward is sparse in other cases we might end up having two or more options as the shortest path with some redundant intermediate events (Fig. 4 or Fig. 8). 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 wrong option automatically falls out. 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.

For the sake of exposition, we pick Task 1 and we train a hybrid deep architecture with three deep feedforward nets as in Fig. 2 to learn the correct policy. Note that according to DFA in Fig. 4 there are two options to accomplish Task 1:

- grass – wood – craft table
- iron – wood – craft table

The training progress is shown in Fig. 13. The purple line shows the very first deep net associated to \( q_1 \), the blue one is one of the intermediate states in the DFA and the black line is associated to \( q_1 \). This figure shows back-propagation from \( q_1 \) to \( q_4 \), 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 at last. 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. The training took approximately 15 hours to complete.

After the training, by starting from any initial point in the crafting environment (Fig. 1), the agent is able to accomplish Task 1 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.

4.1 DeepSynth vs. DeepQN

This section compares the performance of DeepSynth with DeepQN [21] across two different tasks: 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 encompasses a more complicated sequential structure as in Table 1. Fig. 15 gives the result of training for Task 1 in DeepSynth and DeepQN. Note that with the very same training set \( \mathcal{E} \) with 4500 training samples DeepSynth managed to converge while DeepQN failed. However, we allowed DeepQN to explore more and gather enough training samples to converge. The larger training set is denoted by \( \mathcal{E}' \) with
5500 training samples and the algorithm that employed this larger set is called DeepQN* in Fig. 15.

Fig. 16 gives the result of training for Task 3 in DeepSynth and DeepQN where the training set $E$ has 6000 training samples. However, for Task 3 DeepQN failed to converge even after we increased the training set by almost an order of magnitude.

4.2 Synth vs. State Merge

In this section we compare our synthesis based model generation approach 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 [6] algorithm for state merging, several alternatives to determine state equivalence have been proposed over the years [19, 31, 32]. For our experiment we used the MINT (Model INference Technique) [30] tool that implements different variations of the state merge algorithm, including data classifiers [32] to check state equivalence for merging.

We generated models using MINT for all six tasks 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. Even the most succinct models did not accurately capture the underlying sequential behaviour (Table 1) for the tasks.

As examples the smallest model that best fit Task 5 traces includes 49 states (Fig. 10), and 14 states for Task 6 (Fig. 11). Here, the ‘start’ label signifies the beginning of a new trace obtained from another instance of random exploration. Looking at the shortest path between accepting states, we get the sequence “grass – smith table” to accomplish Task 6, while the task requires “iron – wood – smith table” to complete the task. Since state merge algorithms do not produce the most succinct models that fit a given trace it is difficult, if at all possible, to determine the right sequence of events required for a task as is done in our framework.

5 CONCLUSION

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 and formalize this high-level structure by employing techniques from program synthesis and observing exploration traces. This allows us to store and recover any sparse 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.
