Journal of Artificial Intelligence Research 72 (2021) 819-847 Submitted 04/2021; published 11/2021

Task-Aware Verifiable RNN-Based Policies for Partially Observable Markov Decision Processes

Steven Carr STEVENCARR@UTEXAS.EDU
The University of Texas at Austin, 2617 Wichita Street, C0600, Austin, Texas 78712-1221, USA

Nils Jansen N.JANSEN@SCIENCE.RU.NL
Faculty of Science, University of Nijmegen, Postbus 9010, Nijmegen 6500GL, The Netherlands

Ufuk Topcu UTOPCU@UTEXAS.EDU
The University of Texas at Austin, 2617 Wichita Street, C0600, Austin, Texas 78712-1221, USA

Abstract

Partially observable Markov decision processes (POMDPs) are models for sequential decision-making under uncertainty and incomplete information. Machine learning methods typically train recurrent neural networks (RNN) as effective representations of POMDP policies that can efficiently process sequential data. However, it is hard to verify whether the POMDP driven by such RNN-based policies satisfies safety constraints, for instance, given by temporal logic specifications. We propose a novel method that combines techniques from machine learning with the field of formal methods: training an RNN-based policy and then automatically extracting a so-called finite-state controller (FSC) from the RNN. Such FSCs offer a convenient way to verify temporal logic constraints. Implemented on a POMDP, they induce a Markov chain, and probabilistic verification methods can efficiently check whether this induced Markov chain satisfies a temporal logic specification. Using such methods, if the Markov chain does not satisfy the specification, a by-product of verification is diagnostic information about the states in the POMDP that are critical for the specification. The method exploits this diagnostic information to either adjust the complexity of the extracted FSC or improve the policy by performing focused retraining of the RNN. The method synthesizes policies that satisfy temporal logic specifications for POMDPs with up to millions of states, which are three orders of magnitude larger than comparable approaches.

1. Introduction

Partially observable Markov decision processes (POMDPs) are models for sequential decision-making under uncertainty and incomplete information. They model many applications, including control (Bai et al., 2015), planning (Kaelbling et al., 1998), scheduling (Norman et al., 2017) and reinforcement learning (Jaakkola et al., 1995).

Due to their ability to process sequential data efficiently, recurrent neural networks (RNNs) offer an effective policy representation for POMDPs. RNNs use internal memory states, such as those in long short-term memory (LSTM) architectures (Hochreiter & Schmidhuber, 1997), to infer temporal behavior from sequences of data (Pascanu et al., 2014). Reinforcement learning research has shown that RNNs used in environments modeled by POMDPs perform well as black-box functions for either state or value estimators (Wierstra et al., 2007; Bakker, 2001) or as control policies (Hausknecht & Stone, 2015; Heess, Wayne, et al., 2015).

In POMDPs that model agents in safety-critical environments, policies that are guaranteed to prevent unsafe behavior are necessary. The agent's behavior may then have to obey more complicated specifications than maximizing an expected reward, such as reachability, liveness or, more generally, specifications expressed in temporal logic, e. g. linear temporal logic (Pnueli, 1977). Such specifications indeed describe tasks that cannot be expressed using the traditional reward shaping techniques employed in machine learning (Littman et al., 2017; Hadfield-Menell et al., 2017).

Verifying whether an agent following an RNN-based policy in a POMDP satisfies temporal logic specifications is, in general, hard. RNNs are complex structures that capture non-linear input-output relations (Mulder et al., 2015). To formally analyze how RNNs interpret sequences of data, one must fix a defined sequence length for analysis and perform an *unrolling* procedure (Sherstinsky, 2020), which converts the RNN to a feedforward neural network with the same number of layers as that defined length (Goodfellow et al., 2016). Checking whether the agent's behavior satisfies the specification for the set of all possible sequences of data with a given length in the POMDP is intractable (Meuleau et al., 1999). Existing works on verifying policies encoded as feedforward neural networks employ satisfiability-modulo-theories (Q. Wang et al., 2018) or mixed-integer linear programs (Akintunde et al., 2019). However, such methods not only scale exponentially in the number of nodes in the neural network but also rely on rectified linear units, which do not allow for internal memory states such as LSTM.

We combine the effectiveness of RNN-based representations from machine learning with the provable guarantees that are at the heart of formal verification. The latter can efficiently verify whether an agent following a given policy, typically in the form of a finite-state controller (FSC) (Poupart & Boutilier, 2003; Junges et al., 2018), adheres to a temporal logic specification (Baier & Katoen, 2008). However, despite advances in optimization-based approaches (Meuleau et al., 1999; Sharan & Burdick, 2014; Junges et al., 2018), the problem of directly computing such an FSC is EXPTIME-complete (Chatterjee et al., 2015). Such a problem requires—in general—memory of exponential size in the number of POMDP states (Baier et al., 2012). Machine learning, on the other hand, provides an efficient approach, in the form of training RNN-based policy representations from sequences of data, to find candidate policies that might ensure an agent in a POMDP satisfies a temporal logic specification (Heess, Hunt, et al., 2015).

There remains a central gap: How to close the loop between training an RNN-based policy and efficiently verifying for a candidate policy? Our method closes this gap by tightly integrating formal verification and machine learning towards three key steps: (1) extracting an FSC from an RNN-based policy, (2) verifying this candidate FSC for the POMDP against a temporal logic specification, and (3) if needed, either refining the FSC or generating more training data for the RNN, see Figure 1.

To extract an FSC, we employ a technique called *quantized bottleneck insertion* (Koul et al., 2019). An autoencoder (Goodfellow et al., 2016) discretizes the activation function that is associated with the recurrent hidden node of the RNN. Basically, this discretization facilitates a mapping of the continuous memory structure in the RNN to a pre-defined number of memory nodes and transitions of an FSC. Implementing an FSC in a POMDP results in a so-called *induced Markov chain*. For this Markov chain, a model significantly less complex than the original POMDP, verification methods certify against temporal logic specifications scaling up to billions of states (Baier & Katoen, 2008). Tool support is available via probabilistic model checkers such as PRISM (Kwiatkowska et al., 2011) or Storm (Dehnert et al., 2017).

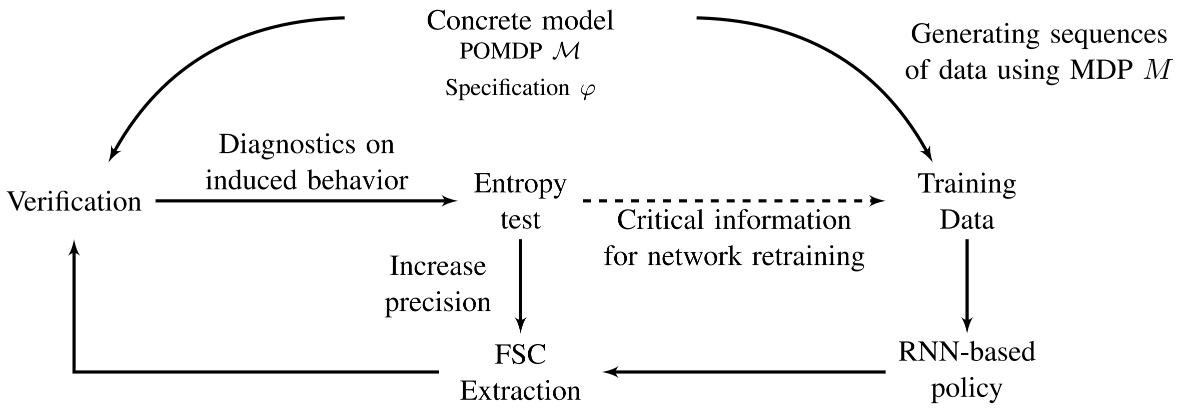


Figure 1: High-level iterative policy improvement process.

Recall that the RNN-based policy, and consequently the extracted FSC, is just a *candidate* policy and may not ensure satisfaction of the specification. If the specification does not hold for the induced Markov chain, the proposed method iteratively improves the extracted policy, as outlined in Figure 1. A by-product of verification is diagnostic information about the states in the POMDP that are critical for the specification in the form of so-called counterexamples (Wimmer et al., 2014; Jansen et al., 2014). The method analyzes whether the FSC can be improved, i.e. by either training a *better* RNN-based policy or by extracting a *better* FSC. This analysis relies on examining whether the decisions made in the resulting counterexamples are considered *arbitrary* by measuring the entropy (Cover & Thomas, 2012) of the action mapping for the FSC. That is, if the entropy is high across these decisions, the method deems the action mapping of the FSC at those decision-points as arbitrary. Therefore, the RNN-based policy needs to reduce the uncertainty in the action mapping at these critical states by training on more sequences of data. If the entropy is low, then increasing the number of memory nodes in the FSC may help it to approximate the decisions of the RNN-based policy more precisely (Koul et al., 2019).

We demonstrate how different approaches to generating sequences of training data for the RNN-based policy impact both the computation time and the probability that the agent satisfies the specification. The proposed method generates sequences of data by assuming full observability and following the policy that maximizes the probability of satisfying the specification in the underlying Markov decision process to create sequences of observation-action pairs (Cassandra, 1998). We compare this approach with one that performs a similar technique but on a task-aware product POMDP, a larger model created by transforming the specification into an automaton and composing it with the POMDP (Bouton et al., 2020).

We also show the existence of a trade-off between the number of memory nodes in an extracted FSC and the probability that the agent following this FSC satisfies the specification. In particular, we empirically demonstrate how increasing the number of memory nodes increases the probability that the induced Markov chain satisfies the temporal logic specification at the cost of longer verification times. However, for each FSC, there is a point of diminishing returns after which increasing the number of memory nodes only marginally increases this probability.

The proposed method computes RNN-based policies and then subsequently extracts candidate policies in the form of an FSC that satisfies temporal logic specifications on a set of POMDP benchmarks with up to millions of states, which is three orders of magnitude larger than comparable approaches. In particular, we benchmark the method against two well-known POMDP solvers from the formal methods (Norman et al., 2017) and planning (Walraven & Spaan, 2017) communities. Computing policies that satisfy temporal logic specifications is undecidable for POMDPs (Madani et al., 1999). Therefore, the proposed method is not *complete*, i.e. it is not guaranteed to find an FSC that ensures an agent in a POMDP satisfies temporal logic specifications. On the other hand, it is *sound*, as in each iteration verification yields provable guarantees on the induced behavior.

Building on preliminary results in Carr et al. (2019, 2020), the current paper makes the following contributions. First, it presents an iterative method that employs state-of-the art tools from machine learning and formal verification to find policies that ensure that an agent in a POMDP satisfies any given linear temporal logic specification. Second, it introduces a novel method that uses a task-aware product model to generate sequences of data for training RNN-based policies. The FSCs, extracted from these RNN-based policies by the iterative method using this product model, have higher probabilities of satisfying the specification than those that are computed using just the POMDP. Finally, it demonstrates empirically that the proposed method can trade policy complexity with increasing the probability of satisfying the specification by restricting the number of memory nodes in the extracted FSC. This complexity trade-off has implications both when attempting to control the available memory for computation and the amount of time spent performing verification.

Structure of the paper: The rest of the paper is structured as follows. After formal foundations on POMDPs in Section 3, Section 4 describes the synthesis procedure. In Section 4.1, we detail the sampling procedure to obtain the RNN training data. We demonstrate the applicability of the proposed approach using a selection of temporal logic examples as well as comparing to well-known benchmarks (Smith & Simmons, 2004) in Section 5.

2. Related Work

RNNs pose suitable policy representations for deep reinforcement learning problems that need to account for sequences of data. For instance, Wierstra et al. (2007) integrate policy gradient methods with RNN-based policies to perform reinforcement learning on POMDPs, which combines REINFORCE (Williams, 1992) with an LSTM architecture (Hochreiter & Schmidhuber, 1997). Recent progress in deep learning has enabled neural networks to compute policies for very large and complex POMDPs (Heess, Wayne, et al., 2015). For example, neural network-based Q-learning algorithms play video games straight from video frames, under partial observability (Mnih et al., 2015). Instead of using recurrent architectures, they solve the memory problem by replaying a series of frames at every step. This work was extended to RNNs by adding an LSTM cell to enhance the algorithm's capacity to incorporate memory-based decision making (Hausknecht & Stone, 2015).

To analyze neural networks, there are two lines of related research. The first one concerns the formal verification of neural network-based control policies. Two prominent approaches for the class of feed-forward deep neural networks rely on encoding neural networks as SMT problems through adversarial examples (Huang et al., 2017) or ReLUs architectures (Katz et al., 2017; Amir et al., 2021). Akintunde et al. (2019) directly verify RNNs constructed with rectified linear unit activation functions using satisfiability-modulo-theories or mixed-integer linear programs. However, these solver-based methods are known to have scalability issues for larger neural networks. The proposed method, in contrast, restricts the neural network to a specific POMDP model to achieve a tractable setting. Khmelnitsky et al. (2020) use active automata learning to create a surrogate finite automaton, whose properties are efficiently verifiable. However, unlike the proposed method, these works do not verify the behavior induced on POMDPs.

The second direction relevant to verifiable properties of RNNs concerns the extraction of FSCs from neural networks (Zeng et al., 1993; Weiss et al., 2018; Michalenko et al., 2019). Another approach takes a (different) perspective in employing RNNs towards regular property model checking (Ghosh & Neider, 2020; Khmelnitsky et al., 2020). All of these approaches do not integrate the extracted FSCs with verification to provide formal guarantees for the agent's behavior in a POMDP. Recent work in the field of graph neural networks gives designers flexibility to incorporate problem specific structure into the network architecture (Wu et al., 2021). However, efficiently verifying the behavior induced by such networks is both problem and architecture specific.

Research in the fields of planning and formal methods focus on the synthesis of FSCs for POMDPs without neural networks. Yet, the underlying problem is EXPTIME-complete (Chatterjee et al., 2015) and intractable for even small instances (Meuleau et al., 1999). Optimization-based approaches use branch-and-bound (Meuleau et al., 1999) or segmentation into reachable sets (Sharan & Burdick, 2014) to limit the searchable space. Junges et al. (2018) construct an FSC using parameter synthesis for Markov chains, which is known to be ETR-complete (Junges, Katoen, et al., 2021), whereas NP $\subseteq$ ETR $\subseteq$ PSPACE. Carr et al. (2018) render common POMDP scenarios as arcade games to capture human preferences that are formally cast into FSCs and subsequently verified. Ahmadi et al. (2020) use control barrier functions to compute safe reachable sets in the belief space of POMDPs. Extensions to epistemic or uncertain POMDPs compute FSCs using convex optimization (Cubuktepe et al., 2021; Suilen et al., 2020).

Chatterjee, Chmelik, and Davies (2016) use a SAT-based approach to compute FSCs for qualitative properties, extended towards the safe exploration of POMDPs by Junges, Jansen, and Seshia (2021). Y. Wang et al. (2018) both constrain the searchable space and bound the horizon for use in an incremental SMT solver. In the planning community, there are many solvers that attempt to compute the policy that maximizes an expected reward without temporal logic specifications (Walraven & Spaan, 2017; Kurniawati et al., 2008; Spaan & Vlassis, 2005). Most of these solvers rely on point-based value iteration with different heuristics on how to segment the searchable space and tend to have limitations associated with model complexity (Zhang et al., 2014). Recent work has expanded these point-based methods to task-aware models, such as those with temporal logic specifications (Bouton et al., 2020). POMCP uses Monte-Carlo tree search to find high performing policies in very large POMDP environments (Silver & Veness, 2010).

3. Preliminaries

A *probability distribution* over a set X is a function $\mu\colon X \to [0, 1] \subseteq \mathbb{R}$ with $\sum_{x\in X} \mu(x) = \mu(X) = 1$. The set of all distributions on X is given by $Distr(X)$, and the support of a distribution μ is $\text{supp}(\mu) = \{x \in X \mid \mu(x) > 0\}$. The *entropy* (Cover & Thomas, 2012) of a distribution μ is defined as $\mathcal{H}(\mu) := -\sum_{x\in X} \mu(x) \log_{|X|} \mu(x)$.

3.1 Partially Observable Markov Decision Processes

A *Markov decision process (MDP)* M is a tuple $M = (S, A, T)$ with a finite (or countably infinite) set S of *states*, a finite set A of *actions*, and a *transition probability function* $T\colon S \times A \to Distr(S)$. We call a pair $(s, a) \in S \times A$ of states and actions a *transition*. The reward function for *transitions* is given by $r\colon S \times A \to \mathbb{R}$.

A finite *path* $e = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots s_n$ of an MDP M is a sequence of states and actions; $\mathrm{last}(e) = s_n$ is the last state of e. The set of all finite paths is $\mathsf{Paths}^{M}_{\mathit{fin}}$.

Definition 1 (POMDP). *A* POMDP *is a tuple $\mathcal{M} = (M, Z, O)$, with M the* underlying MDP *of $\mathcal{M}$, Z a finite set of observations, and $O\colon S \to Z$ the* observation function.

For brevity in our presentation, we use so-called deterministic observations functions which may be derived from the more standard stochastic observation functions $O\colon S \to \mathit{Distr}(Z)$ via a (polynomial) reduction (Chatterjee, Chmelík, et al., 2016). In our experiments we use stochastic functions.

For POMDPs, observation-action sequences are based on a finite path $e \in \mathsf{Paths}^{M}_{\mathit{fin}}$ of the underlying MDP M and have the form: $e_o = O(e) = O(s_0) \xrightarrow{a_0} O(s_1) \xrightarrow{a_1} \cdots O(s_n)$. The set of all finite observation-action sequences for a POMDP $\mathcal{M}$ is $\mathsf{ObsSeq}^{\mathcal{M}}_{\mathit{fin}}$.

While the agent acts within the environment, it encounters certain observations, according to which it can infer the probability of the system being in a certain state. Technically, this *belief* b is a distribution $b \in \mathit{Distr}(S)$, such that $b(s)$ describes the probability to be in state $s \in S$.

Definition 2 (POMDP Policy). *A policy $\pi\colon \mathsf{Paths}^{M}_{\mathit{fin}} \to \mathit{Distr}(A)$ maps a finite path e to a distribution over actions. A policy is* observation-based*, if for each two paths e, e' it holds that $O(e) = O(e') \Rightarrow \pi(e) = \pi(e')$. A policy is* memoryless*, if for each e, e' it holds that* $\mathrm{last}(e) = \mathrm{last}(e') \Rightarrow \pi(e) = \pi(e')$. *A policy is* deterministic *(or pure), if for each e, $\pi(e)$ is a Dirac distribution, i.e., if* $|\operatorname{supp}(\pi(e))| = 1$. $\Pi^{\mathcal{M}}_{z}$ *is the set of all observation-based policies for $\mathcal{M}$.*

A policy resolves the nondeterministic choices in a POMDP, potentially based on the history of previous observations, by assigning distributions over actions. In particular, a policy π applied to a POMDP $\mathcal{M}$ (randomly) *generates observation-action sequences* that we will use as data later on. Technically, such sequences are derived from the *induced discrete-time Markov chain (DTMC)* $\mathcal{M}^{\pi}$, which does not contain any nondeterminism or partial observability.

Note that our definition restricts POMDP policies to *finite memory*. We represent such *finite-memory policies* as *finite-state controllers (FSCs)*.

Definition 3 (FSC). *A k-FSC for a POMDP $\mathcal{M} = (M, Z, O)$ is a tuple $\mathcal{A} = (N, n_I, \alpha, \delta)$ where N is a* finite set *of k memory nodes, $n_I \in N$ is the initial memory node, α is the action-mapping $\alpha\colon N \times Z \to \mathit{Distr}(A)$ and δ is the memory update $\delta\colon N \times Z \times A \to \mathit{Distr}(N)$.*

An FSC has the POMDP's observations Z as input and the actions A as output. Upon an observation, depending on the current memory node the FSC is in, the action-mapping α returns a distribution over A followed by a (probabilistic) change of memory nodes according to δ. Note that depending on the application, the memory update may also be deterministic, that is of the form $\delta\colon N \times Z \times A \to N$.

FSCs are typical *finite-state machines* with inputs and outputs. In particular, FSCs are an extension of so-called *Moore machines*, where the action-mapping is deterministic, that is, $\alpha\colon N \times Z \to A$, and the memory update $\delta\colon N \times Z \to \mathit{Distr}(N)$ does not depend on the choice of action.

3.2 Specifications

Undiscounted expected reward properties $\varphi = \mathbb{E}_{\sim\lambda}(\lozenge\, \mathit{ap})$ require that the expected accumulated cost until reaching a state satisfying ap respects $\lambda \in \mathbb{R}_{\geq 0}$. As a more general notion to reason about safety, we consider *linear temporal logic (LTL)* properties (Pnueli, 1977).

Definition 4 (LTL Properties)**.** *For a set of atomic propositions AP, which are either satisfied or violated by a state, and $ap \in AP$, the set of all LTL formulas is defined by the following grammar:*

$$\Psi ::= ap \mid (\Psi \wedge \Psi) \mid \neg\Psi \mid \bigcirc \Psi \mid \Box\, \Psi \mid (\Psi \,\mathsf{U}\, \Psi)\,.$$

Intuitively, arbitrarily complex LTL formulas may be generated by this grammar as it allows to (recursively) replace Ψ by atomic propositions or (nested) formulas that may use all necessary unary or binary operators. In particular, a path e satisfies the proposition a if its first state does; $(\psi_1 \wedge \psi_2)$ is satisfied, if e satisfies both ψ_1 and ψ_2; $\neg\psi$ is true on e if ψ is not satisfied. The formula $\bigcirc\,\psi$ holds on e if the subpath starting at the second state of e satisfies ψ; e satisfies $\Box\,\psi$ if all suffixes of e satisfy ψ. Finally, e satisfies $(\psi_1 \,\mathsf{U}\, \psi_2)$ if there is a suffix of e that satisfies ψ_2 and all longer suffixes satisfy ψ_1. $\Diamond\,\psi$ abbreviates $(\text{true}\,\mathsf{U}\,\psi)$. To add such semantics to (PO)MDP states, one can use a *labeling function* $L\colon S \to 2^{AP}$ to add sets of atomic propositions (labels) to the states. A path $e \in \mathsf{Paths}^{M}_{\mathit{fin}}$ of an MDP then generates a so-called *trace* of atomic propositions. For more details, we refer to literature from the formal methods community such as that by Baier and Katoen (2008) and Clarke et al. (2018).

We form LTL specifications for POMDPs by requiring that the probability of satisfying an LTL-property respects a given bound, denoted $\varphi = \mathbb{P}_{\sim\lambda}(\psi)$ for $\sim\, \in \{<, \leq, \geq, >\}$ and $\lambda \in [0,1]$. For instance, if we want to constrain the probability to reach a critical state s, labelled $L(s) = \{\mathit{crit}\}$, to be at most 10%, we write $\varphi = \mathbb{P}_{\leq 0.1}(\Diamond\, \mathit{crit})$. In particular, the aim is to synthesize a policy that verifiably satisfies the specification φ.

Verification of LTL properties. We first introduce the notion of deterministic Rabin automata.

Definition 5 (DRA)**.** *A* deterministic Rabin automaton (DRA) *is a tuple $\mathcal{R} = (Q, AP, \Delta, q_0, F)$ with Q the set of states, AP the set of atomic propositions, $\Delta\colon Q \times 2^{AP} \to Q$ the transition function, q_0 the initial state, and F the acceptance condition: $F = (L_1, K_1), \ldots, (L_k, K_k)$ where L_i and K_i are sets of states for all $1 \leq i \leq k$ with $k \in \mathbb{N}$.*

Intuitively, a Rabin automaton accepts an (infinite) sequence of states, if there is a pair (L_i, K_i) such that (some) states from L_i are visited infinitely often, while K_i is at some point not visited anymore. An LTL formula φ can be transformed into a DRA $\mathcal{R}_\varphi$, which yields the means to automatically decide (via the automaton) if a sequence of atomic propositions from AP satisfies φ. The number of states in this automaton is doubly exponential in the size of AP. However, efficient implementations and heuristics effectively reduce the number of states (Kretínský et al., 2018).

To verify LTL properties for POMDPs, we compute the product $\mathcal{M} \times \mathcal{R}_\varphi$ of a POMDP $\mathcal{M}$ and a DRA $\mathcal{R}_\varphi$. The details of this construction are beyond the scope of this paper and we refer to Bouton et al. (2020). Essentially, this product yields a larger POMDP in which one computes the probability to reach so-called *maximal end components*. The probability of reaching these components can be related to the probability of satisfying LTL specifications for the original POMDP.

Finally, we state the following relation. A *specification* φ is *satisfied* for POMDP $\mathcal{M}$ and π if and only if it is satisfied in the induced DTMC $\mathcal{M}^\pi$ ($\mathcal{M}^\pi \models \varphi$). Intuitively, if a *candidate* FSC is available, we are able to use efficient verification tools and algorithms for DTMCs to verify the correctness of the specification.

4. Method

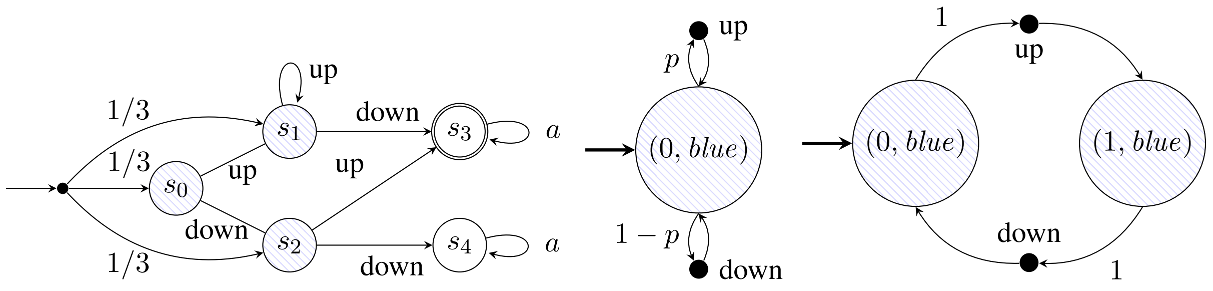
We first state the formal problem that we address in this paper.

Formal problem statement. For a POMDP $\mathcal{M}$ and a specification φ, where either $\varphi = \mathbb{P}_{\sim\lambda}(\psi)$ with ψ an LTL specification, or $\varphi = \mathbb{E}_{\sim\lambda}(\lozenge\, a)$, the problem is to determine a (finite-memory) policy $\pi \in \Pi_z^{\mathcal{M}}$ such that $\mathcal{M}^\pi \models \varphi$.

If such a policy does not exist, the problem is infeasible. Note that, in general, this problem is undecidable and each method is necessarily incomplete.

Outline. Recall Figure 1 that outlines our overall approach. We train and maintain an RNN that serves as efficient representation of a POMDP policy. As safety-critical scenarios necessitate a *sound* notion of correctness, we evaluate such an RNN-based policy using formal verification against LTL specifications, see Section 3.2. We provide the means to use the verification result to effectively re-train the RNN in case the specification does not hold. There are four main building blocks towards that approach, and we structure this section accordingly: (1) *Training* the RNN (Section 4.1), (2) *extracting* FSCs as tractable representation of the RNN-based policy (Section 4.2), (3) *evaluating* the policy (Section 4.3), and (4) *improving* the policy (Section 4.4).

We start with a simple 5-state reachability example to highlight the utility of FSCs as finite-memory POMDP policies.



(a) 5-State POMDP (b) 1-FSC (c) 2-FSC

Figure 2: (a) POMDP for Example 1 with three observations $\{blue, s_3, s_4\}$ with (b) 1-FSC and (c) 2-FSC. Both FSCs are defined for observing "blue" and subsequent action choices that may result in a change of memory node for the 2-FSC.

Example 1. *Consider the POMDP in Figure 2. The POMDP has three observations (*blue*, s_3 and s_4), where observation* blue *is received upon visiting s_0, s_1, and s_2. That is, the agent is unable to distinguish between these states. The specification is $\varphi = \text{Pr}_{\geq 0.9}(\lozenge\, s_3)$, so the agent reaches state s_3 with at least probability 0.9. We define the following 1-FSC $\mathcal{A}_1$ with one memory node 0:*

$$\alpha(0, blue) = \begin{cases} up & \textit{with probability} \quad p, \\ down & \textit{with probability} \quad 1-p, \end{cases}$$
$$\delta(0, z, a) = 0 \quad \forall z \in Z, a \in A.$$

A 2-FSC with two memory nodes (0 and 1), see Figure 2c, allows for greater expressivity, i.e. the policy can base its decision on larger observation sequences.

With this memory structure, we can create an FSC $\mathcal{A}_2$ that ensures the satisfaction of φ:

$$\alpha(0, blue) = \begin{cases} up & \textit{with probability} \quad 1, \\ down & \textit{with probability} \quad 0, \end{cases}$$

$$\alpha(1, blue) = \begin{cases} up & \textit{with probability} \quad 0, \\ down & \textit{with probability} \quad 1, \end{cases}$$

$$\delta(0, blue, up) = 1,$$

$$\delta(1, blue, down) = 0.$$

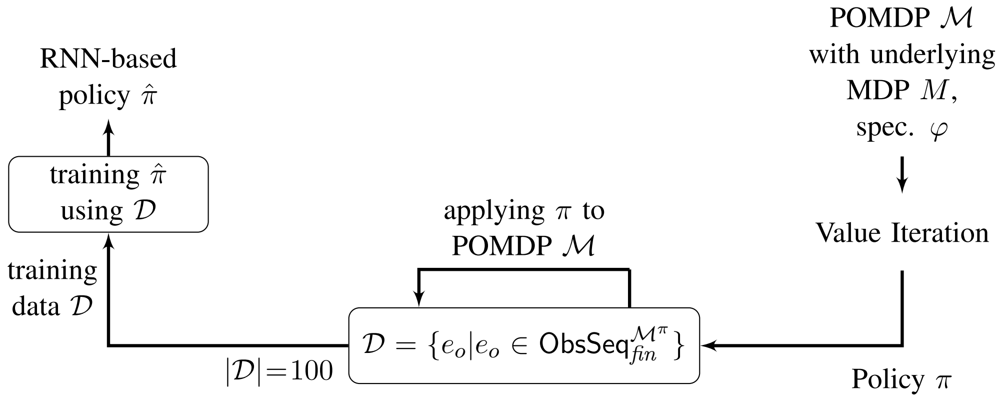

RNN-based policy $\hat{\pi}$
training $\hat{\pi}$ using $\mathcal{D}$
training data $\mathcal{D}$
$|\mathcal{D}|=100$
applying π to POMDP $\mathcal{M}$
$\mathcal{D} = \{e_o | e_o \in \mathsf{ObsSeq}_{fin}^{\mathcal{M}^\pi}\}$
POMDP $\mathcal{M}$ with underlying MDP M, spec. φ
Value Iteration
Policy π

Figure 3: Process for generating 100 sequences of data and training an RNN-based policy.

4.1 Training an RNN-Based Policy

Figure 3 gives the high-level process for training policies from observation sequences. We first define *recurrent neural networks (RNNs)* as efficient representation of POMDP policies.

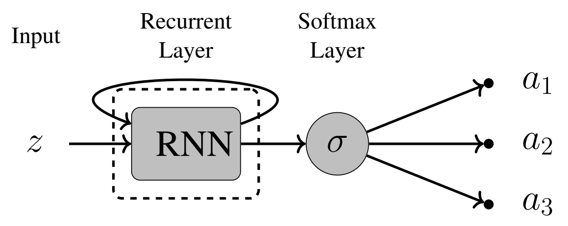
Definition 6 (RNN-based policy)**.** *An RNN-based policy for a POMDP is a function $\hat{\pi}\colon \mathsf{ObsSeq}_{fin}^{\mathcal{M}} \to Distr(A)$. The RNN, which receives sequential input in the form of (finite) observation-action sequences from $\mathsf{ObsSeq}_{fin}^{\mathcal{M}}$, the output is a distribution over actions. To be more precise, we identify the main components of such a network. An RNN-based policy $\hat{\pi}$ is sufficiently described by a* hidden-state update function $\hat{\delta}\colon \mathbb{R} \times Z \times A \to \mathbb{R}$ *and an* action-mapping $\pi_h\colon \mathbb{R} \to Distr(A)$.

Consider the following observation-action sequence:

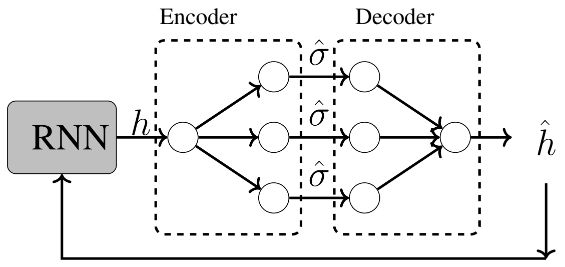
$$O(e) = O(s_0) \xrightarrow{a_0} O(s_1) \xrightarrow{a_1} \cdots O(s_i) \tag{1}$$

The RNN-based policy receives an observation-action sequence and returns a distribution over the action choices. Throughout the execution of the sequence, the RNN holds a continuous hidden state $h \in \mathbb{R}$, typically described as an internal memory state, which captures previous information. On each transition, this hidden state is updated to include the information of the current state and the last action taken under the hidden-state update function $\hat{\delta}$. From the prior observation sequence in (1), the corresponding hidden state sequence would be defined as:

$$\hat{\delta}(e) = h_0 \xrightarrow{a_0,\, O(s_1)} h_1 \xrightarrow{a_1,\, O(s_2)} \cdots h_i$$



(a) RNN-based Policy $\hat{\pi}$.



(b) RNN block and associated QBN of $B_h = 3$ with quantized activation $\hat{\sigma}\colon \mathbb{R} \to \{-1, 0, 1\}$.

Figure 4: RNN-based policy structure (a) without and (b) with a QBN.

Additionally, the output of the RNN-based policy is expressed by the action-distribution function π_h, which maps the value of hidden state to an action mapping π_h. At an internal memory state h_i, we have $\hat{\delta}(h_i, O(s_i), a_i) = h_{i+1}$ and $\pi_h(h_{i+1}) = \mu(A)$ for state s_i on path e.

RNN-based policy architecture. We construct an RNN-based policy using a three-layer network, shown in Figure 4a. We use an LSTM architecture (Hochreiter & Schmidhuber, 1997) for the recurrent layer $\hat{\delta}$ and then a softmax layer for the output action mapping π_h (see Definition 6). Both input and output sets were processed using One-hot encoding (Goodfellow et al., 2016). To fit the RNN model to the sequences of training data, we use the Adam optimizer (Kingma & Ba, 2015) with a categorical cross-entropy error function (Goodfellow et al., 2016).

RNN-based policies learning temporal logic specifications from sequential data. The hidden state update $\hat{\delta}$ of an RNN-based policy allows it to make inferences about the sequences of data it processes. In a POMDP, this sequential data takes the form of observation sequences $e_o \in \mathsf{ObsSeq}^{\mathcal{M}}_{\mathit{fin}}$. For specifications that concern reachability or expected reward measures, see Section 3.2, the internal memory will indirectly learn to infer the belief by processing the observation sequences (Wierstra et al., 2007; Hausknecht & Stone, 2015). In problems with more complex specifications than simple reachability and maximizing an expected reward, the internal memory will also learn how observation sequences map to correct decisions with respect to the specification.

Example 2. *Consider an LTL specification such as $\varphi = \Box(\Diamond A \wedge (A \rightarrow \Diamond B))$, where A and B are labels for two distinct states. This specification requires the agent to always eventually travel to the state labeled A and then to the state labeled B. Now consider an agent, currently at the state that is labeled A, following the policy π and generating observation sequences e_o on a POMDP $\mathcal{M}$. These observation-action sequences e_o will eventually reach the state labeled B. The RNN-based policy can indirectly learn from these sequences of data, inferring that after reaching the state labeled A the agent should attempt to reach the state labeled B eventually. In this case, the LSTM tracks, using internal memory, which label may have been visited recently.*

Consider now a POMDP $\mathcal{M}$ and a DRA $\mathcal{R}_\varphi = (Q, AP, \Delta, q_0, F)$ generated from the LTL formula φ. We compute the product POMDP $\mathcal{M} \times \mathcal{R}_\varphi$. The state space of this product POMDP is $S \times Q$, thus each state consists of a POMDP state from S and a Rabin automaton state from Q. By extension, the sequences of training data $e'_o \in \mathsf{ObsSeq}^{\mathcal{M}\times\mathcal{R}_\varphi}_{\mathit{fin}}$ in form of observation-action sequences, as explained above, carries the information from the POMDP states and the LTL specification. Thus the hidden state update $\hat{\delta}$ indirectly tracks both the agent's belief on the state as well as its status with respect to the specification. Encoding this information into the continuous hidden state $h_i \in \mathbb{R}$. An RNN-based policy makes no distinction between these two uses of memory: it merely processes and implements actions based on the observation-action sequence e_o and the hidden state h_i. One consequence of this memory coalescence is the additional difficulty ascertaining the best approach for how to improve the policy, full details outlined in Section 4.4.

4.1.1 GENERATING TRAINING DATA

In the field of POMDP reinforcement learning, there are many techniques for generating sequences of data for training the RNN-based policy. Some examples include: Monte Carlo tree search in the form of POMCP (Silver & Veness, 2010), tree-structured finite history Q-learning (Hernandez-Gardiol & Mahadevan, 2000) and policy gradient frameworks (Wierstra et al., 2007).

QMDP. In this work, we need to generate sequences of data $\mathcal{D} = \{e_o^0, \cdots, e_o^m\}$ for training an RNN-based policy $\hat{\pi}$ that represents a good initial candidate policy. We use a heuristic search approach based on the underlying MDP M known as QMDP (Cassandra, 1998). QMDP temporarily ignores the observation function and assumes full observability to find Q-values $Q_{MDP}(s, a)$ for the state-action decisions in the underlying MDP, which can be computed efficiently for millions of states using value iteration (Puterman, 1994). QMDP approximates the Q-value for action a in the POMDP by $Q_a(b) = \sum_{s \in S} b(s) Q_{MDP}(s, a)$, which relies on the belief distribution $b \in Distr(S)$. The agent can track the belief distribution b as it executes a policy, inferring the probability of the system being in a certain state from the observation-action sequence $e_o \in \mathsf{ObsSeq}_{\mathit{fin}}^{\mathcal{M}}$.

The advantage to QMDP initialization is that it only requires one to calculate and query the underlying Q-values $Q_{MDP}(s, a)$. It efficiently approximates a value function for any belief distribution $b \in Distr(s)$ regardless of whether that belief is discrete or continuous.

We chose such an approach for its simplicity (Shani et al., 2013); finding quality approaches to discretizing the belief distribution and performing tree search is an active field in planning but often is intractable (Chatterjee et al., 2015). Additionally, we did not want to compare and contrast many different POMDP reinforcement learning techniques training RNN candidate policies, which we view as outside the scope of this work. However, such a choice is not without drawbacks. QMDP is necessarily suboptimal because it ignores partial observability and the curse of history (Smith & Simmons, 2004). Rather QMDP serves the method's need for a good initial guess. In Section 5, we demonstrate the downside to this trade-off in the *RockSample* problem (Smith & Simmons, 2004).

Implementation. In practice, we first compute a policy $\pi \in \Pi^M$ of the underlying MDP M that satisfies φ using the STORM probabilistic model checker (Dehnert et al., 2017). Then we sample an initial state uniformly over the initial belief support $s_0 \in \mathrm{supp}(b)$ and generate finite observation paths e_o, thereby creating multiple trajectory trees (Kearns et al., 1999). When generating sequences of data, selecting one of the trees and following it to a leaf, which forms either at a pre-defined maximum sequence length or a deadlock, gives a finite path $e \in \mathsf{Paths}_{\mathit{fin}}^{M}$. From this path e, we generate one possible observation-action sequence $e_o \in \mathsf{ObsSeq}_{\mathit{fin}}^{\mathcal{M}}$.

Example 1 (cont.). *Consider the POMDP in Figure 2, a sample set of sequences of data would be:* $\mathcal{D} = \{e_o^0 = (\mathit{blue}, \mathit{up}, \mathit{blue}, \mathit{down}, s_3),\ e_o^1 = (\mathit{blue}, \mathit{down}, s_3),\ e_o^2 = (\mathit{blue}, \mathit{up}, s_3)\}$. *An RNN policy π trained on these sequences would yield a policy for observation-action sequence*

$e_{o,0} = (blue)$ as $\pi(e_{o,0}) = \{0.67 : up, 0.33 : down\}$, which has a categorical cross-entropy loss of approximately 0.585. Similarly, the same RNN policy for a longer observation-action sequence such as $e_{o,1} = (blue, up, blue)$ yields a policy $\pi(e_{o,1}) = \{1.0 : down\}$, for a cross-entropy loss of 0.

Implementation: generating training data with the product POMDP. Recalling that the RNN-based policy learns two elements from the observation sequences: the belief state and the correct decision according the specification at that belief state. One can disentangle these two elements by incorporating the product model $\mathcal{M} \times \mathcal{R}_\varphi$, see Section 3 and Bouton et al. (2020) for details. Instead of processing observation sequences e_o generated by following policy π on the underlying MDP M, we generate new observation sequences $e'_o \in \mathsf{ObsSeq}_{\mathit{fin}}^{\mathcal{M}\times\mathcal{R}_\varphi}$ by following the policy π' that maximizes the probability of reaching the maximal-end components in the underlying product MDP $M \times \mathcal{R}_\varphi$. This segmentation makes explicit that the RNN-based policy need only process the observation sequences to infer the belief $b \in \mathit{Distr}(S \times Q)$. In Sections 4.3 and 5.3, we will detail benefits of such a distinction when evaluating the policies and then identifying the critical states from which we generate new sequences of data.

Sampling large environments. In POMDPs $\mathcal{M}$ with a large state spaces ($|S| > 10^6$), synthesizing the underlying MDP policy $\pi \in \Pi^M$ increases the initial computation overhead. In such cases, we generate the observation sequences using a smaller but similar environment that shares the observation Z and action A spaces with $\mathcal{M}$. For example, consider a gridworld scenario with a moving obstacle that has the same underlying probabilistic movement for different problem sizes; such a framework can provide a similar dataset regardless of the size of the grid.

4.2 Policy Extraction

In this section and Figure 5, we describe how we adapt the method called quantized bottleneck insertion (Koul et al., 2019) to extract an FSC from a given RNN-based policy. Let us first explain the relationship between the main components of an RNN-based policy $\hat{\pi}$ (Definition 6) and an FSC $\mathcal{A}$ (Definition 3). In particular, the hidden-state update function $\hat{\delta}$ takes as input a real-valued hidden state of the policy network, while the FSC's memory update function δ takes a memory node from the finite set N. Figure 4a describes a simplified architecture for the former since its recurrent component acts as the hidden-state update function $\hat{\delta}$. The key for linking the two is therefore a mechanism that encodes the continuous hidden state h into a set N of discrete memory nodes. We outline such a mechanism in the sequel and in Figure 4b, which shows the modified activation function (formed using an encoder and a decoder).

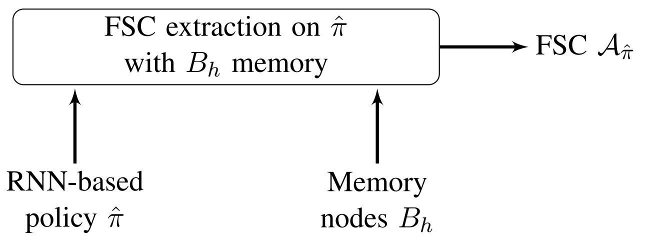

FSC extraction on $\hat{\pi}$ with B_h memory
FSC $\mathcal{A}_{\hat{\pi}}$
RNN-based policy $\hat{\pi}$
Memory nodes B_h

Figure 5: Process for extracting an FSC from an RNN-based policy.

RNN-based policy modification. We leverage an autoencoder (Goodfellow et al., 2016) in the form of a *quantized bottleneck network* (QBN) (Koul et al., 2019). This QBN, consisting of an encoder and a decoder, is inserted into the RNN-based policy directly before the softmax layer, see Figure 4b. In the encoder, the continuous hidden state value $h \in \mathbb{R}$ is mapped to an intermediate real-valued vector $\mathbb{R}^{B_h}$ of pre-allocated size B_h. The decoder then maps this intermediate vector into a discrete vector space defined by $\{-1, 0, 1\}^{B_h}$. This process, illustrated in Figure 4b, provides a mapping of the continuous hidden state h into 3^{B_h} possible discrete values. We denote the discrete state for h by $\hat{h}$ and the set of all such discrete states by $\hat{H}$. Note, that $|\hat{H}| \leq 3^{B_h}$ since not all values of the hidden state may be reached in an observation sequence. Koul et al. (2019) use another QBN for a continuous observation space, however, in this work the method computes policies for POMDPs that have observation functions O, which map to a finite set of observations Z and therefore we can neglect the additional autoencoder.

FSC construction. After the QBN insertion we simulate a series of executions, querying the modified RNN for action choices on the POMDP. We form a dataset of consecutive pairs $(\hat{h}_t, \hat{h}_{t+1})$ of discrete hidden states, the action a_t and the observation z_{t+1} that led to the transition $\{\hat{h}_t, a_t, z_{t+1}, \hat{h}_{t+1}\}$ at each time t during the execution of the RNN-based policy. The number of accessed memory nodes $N \subseteq \hat{H}$ corresponds to the number of different discrete states $\hat{h} \in \hat{H}$ in this dataset. The deterministic memory update rule $\delta(n_t, a_t, z_{t+1}) = n_{t+1}$ is obtained by constructing a $N \times (|Z| \times |A|)$ transaction table, for a detailed description see (Koul et al., 2019). We can additionally construct the action mapping $\alpha \colon N \times Z \to \mathit{Distr}(A)$ with $\alpha(n_t, z_t) = \mu \in \mathit{Distr}(A)$ by querying the softmax-output layer (see Figure 4a) for each memory node and observation. Note that it is possible to build a stochastic memory update rule δ by taking the distribution from sampling for each transition multiple times. Such an approach would account for possible degeneracies, where the same hidden state, action and observation may map to different nodes. One can replicate such behavior by adding additional memory nodes, whose values are sufficiently close together. We show empirically in Section 5 that there exist diminishing returns on additional memory and consequently using a stochastic memory rule would not justify the sampling cost required to generate the distributions.

4.3 Policy Evaluation

We assume that for POMDP $\mathcal{M} = (M, Z, O)$ and specification φ, we have an extracted FSC $\mathcal{A}_{\hat{\pi}} \in \Pi_z^{\mathcal{M}}$ as in Definition 3. We use the policy $\mathcal{A}_{\hat{\pi}}$ to obtain the induced DTMC $\mathcal{M}^{\mathcal{A}_{\hat{\pi}}}$. For this DTMC, formal verification through model checking checks whether $\mathcal{M}^{\mathcal{A}_{\hat{\pi}}} \models \varphi$ and thereby provides hard guarantees about the quality of the extracted FSC $\mathcal{A}_{\hat{\pi}}$ regarding φ. In particular, (probabilistic) model checking provides the probability (or the expected reward) to satisfy a specification for *all states* $s \in S$ via solving linear equation systems (Baier & Katoen, 2008). Note this policy is only a prediction extracted from the RNN-based and therefore the guarantees do not directly carry over to the RNN-based policy.

Example 1 (cont.). *Consider the case in the 1-FSC $\mathcal{A}_1$ (Figure 2b) where $p = 1$, the probability of reaching the state s_3 in the induced DTMC is $\Pr(\lozenge\, s_3) = \frac{1}{3}$. Clearly, the behavior induced by this 1-FSC violates the specification and we obtain two counterexamples of critical memory-state pairs for this policy $\mathcal{A}_{\hat{\pi}}$: $(0, s_0)$ and $(0, s_1)$.*

If the specification does not hold, the policy may require refinement. As discussed before, on the one hand we can increase the number of memory nodes B_h to extract a new FSC. At each iteration of the loop in Figure 6, we modify the QBN for the new, increased, level of discretization and obtain a new FSC $\mathcal{A}_{\hat{\pi}}$ using the process outlined in Section 4.2. On the other hand, we may decide via a formal entropy check whether new data need to be generated to actually improve the policy.

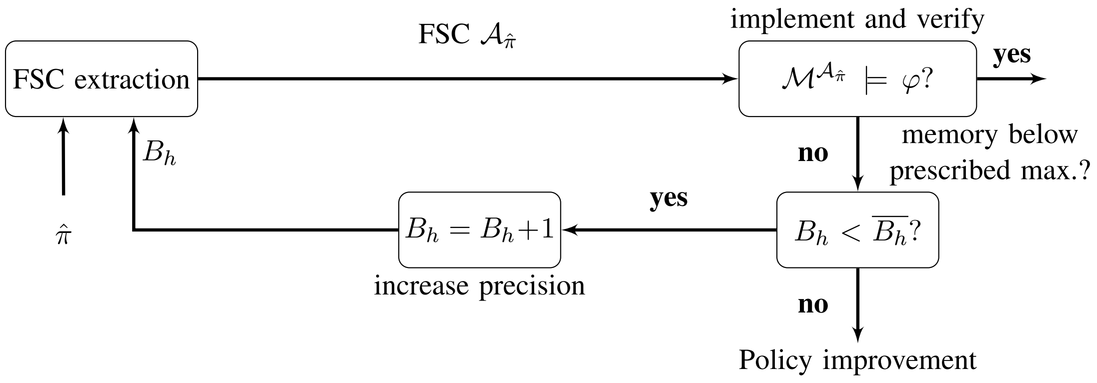


Figure 6: Process for FSC memory refinement.

Product model. Consider the FSC extracted $\mathcal{A}'_{\hat{\pi}}$ from the RNN-based policy trained on observation sequences e'_o drawn from the product model $\mathcal{M}\times\mathcal{R}_\varphi$. We can use this FSC to obtain the induced DTMC $(\mathcal{M}\times\mathcal{R}_\varphi)^{\mathcal{A}'_{\hat{\pi}}}$. In this instance, verification checks whether $(\mathcal{M}\times\mathcal{R}_\varphi)^{\mathcal{A}'_{\hat{\pi}}} \models \mathbb{E}_{\geq\lambda}[\lozenge\, Reach]$, which gives the same guarantees as the policy evaluation described above. For the complete definition of $\lozenge\, Reach$ see Bouton et al. (2020). Model checking provides the probability of reaching the *maximal end components* for all states $(s, q) \in S \times Q$ in the product model $\mathcal{M} \times \mathcal{R}_\varphi$.

4.4 Policy Improvement

Our goal is to determine whether an RNN-based policy requires more training data $\mathcal{D}$ or not. Existing approaches in supervised learning methods leverage benchmark comparisons between a train-test set using a loss function (Baum & Wilczek, 1987). Loss visualization, proposed by Goodfellow and Vinyals (2015), provides a set of analytical tools to show model convergence. However, such approaches aim at continuous functions instead of the discrete representations as in the FSC. More importantly, we leverage the information gained from a model-based approach.

Counterexamples. We first determine a set of states that are critical for satisfaction of the specification under the current policy. Consider a sequence of memory nodes and observations $(n_0, z_0) \xrightarrow{a_0} \cdots \xrightarrow{a_{t-1}} (n_t, z_t)$ from the POMDP $\mathcal{M}$ under the FSC $\mathcal{A}_{\hat{\pi}}$. For each of these sequences, we collect the states $s \in S$ underlying the observations, e.g., $O(s) = z_i$ for $0 \leq i \leq t$. As we know the probability or expected reward for these states to satisfy the specification from previous model checking, we can now directly assess their criticality regarding the specification. We collect all pairs of memory nodes and states from $N \times S$ that contain critical states and build the set $Crit^{\mathcal{M}}_{\mathcal{A}_{\hat{\pi}}} \subseteq N \times S$ that serves us as a counterexample. These pairs carry the joint information of critical states and memory nodes from the policy applied to the DTMC $\mathcal{M}^{\mathcal{A}_{\hat{\pi}}}$.

Counterexamples with the product model. The set of counterexamples $Crit^{\mathcal{M}}_{\mathcal{A}_{\hat{\pi}}} \subseteq N \times S$ are taken from pairs of memory nodes and states $N \times S$. The product POMDP $\mathcal{M} \times \mathcal{R}_\varphi$ of $\mathcal{M}$ and the DRA for the LTL specification φ has the state space $S \times Q$. Thus, there is an additional dimension

Q in the form of the DRA states. The counterexamples drawn from this product model form the set $\mathit{Crit}_{\mathcal{A}_{\hat{\pi}'}}^{\mathcal{M}\times\mathcal{R}_\varphi} \subseteq N \times S \times Q$. As a consequence, this set of counterexamples $\mathit{Crit}_{\mathcal{A}_{\hat{\pi}'}}^{\mathcal{M}\times\mathcal{R}_\varphi}$ contains more detail in the diagnostic information than the set $\mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}}$ derived from $\mathcal{M}$ since it encodes which state $q \in Q$ in the DRA $\mathcal{R}_\varphi$ may lead to a violation of the specification φ.

Entropy measure. The average entropy across the distributions over actions at the choices induced by the counterexample set $\mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}}$ is our measure of choice to determine the level of training for the RNN-based policy. Specifically, for each pair $(n, s) \in \mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}}$, we collect the distribution $\mu \in \mathit{Distr}(A)$ over actions that $\mathcal{A}_{\hat{\pi}}$ returns for the observation $O(s)$ when it is in memory node n. Then, we define the *evaluation function* H using the entropy $\mathcal{H}(\mu)$ of the distribution μ:

$$H\colon \mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}} \to [0,1] \text{ with } H(n,s) = \mathcal{H}(\mu)$$

For high values of H, the distribution is uniform across all actions and the associated RNN-based policy is likely extrapolating from unseen inputs.

We observe that when there are fewer samples and higher memory nodes, the extracted FSC tends to perform arbitrarily, see Section 5.3 and Figure 13 for a detailed empirical analysis. We lift the function H to the full set $\mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}}$:

$$H(\mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}}) = \frac{1}{|\mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}}|} \sum_{(n,s)\in \mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}}} H(n,s) \tag{2}$$

We compare the average entropy over all decision-points of the counterexample against a constant threshold $\eta \in [0,1]$, that is, if $H(\mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}}) > \eta$, we will provide more training data. Vice versa, if $H(\mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}}) \leq \eta$, we increase the upper bound on the number of memory nodes in the FSC, see Figure 6 for the process for increasing precision.

Example 1 (cont.)**.** *Under the working example, the policy $\mathcal{A}_1$ was the 1-FSC with $p = 1$ (Figure 2b), which produces two counterexample memory and state pairs: $\mathit{Crit}_{\mathcal{A}_1}^{\mathcal{M}} = \{(0, s_0), (0, s_1)\}$. The procedure would then examine the policy's average entropy at these critical components $(n, s) \in \mathit{Crit}_{\mathcal{A}_1}^{\mathcal{M}}$, which in this trivial example is given by $H(\mathit{Crit}_{\mathcal{A}_1}^{\mathcal{M}}) = -p\log_2(p) - (1-p)\log_2(1-p) = 0$ from (2). The average entropy is below a prescribed threshold, $\eta = 0.5$, and thus we increase the number of memory nodes, which results in the satisfying FSC $\mathcal{A}_2$ in Figure 2c.*

Generating new sequences of training data from counterexamples. There are several methods to collect data for training the policy network. In this work, we generate new training sequences for the existing policy network by using the QMDP approach outlined in Section 4.1.1 and Figure 7. The key idea is to modify the policy $\pi \in \Pi^M$ for the underlying MDP that generates the observation sequences $e_o^* \in \mathsf{ObsSeq}_{\mathit{fin}}^{\mathcal{M}}$. We redistribute the probabilities of action choices such that the policy favors non-critical states that are not within $\mathit{Crit}_{\mathcal{A}_{\hat{\pi}}}^{\mathcal{M}}$ or $\mathit{Crit}_{\mathcal{A}_{\hat{\pi}'}}^{\mathcal{M}\times\mathcal{R}_\varphi}$. This "local" redistribution of probabilities does not solve the overall problem but is likely to generate better training data. We realize this idea using the following linear program:

$$\max_{\pi(z)(a), a\in A} \min_{s\in S} p_s \tag{3}$$

subject to

$$\forall s \in O^{-1}(z). \quad p_s = \sum_{a\in A} \pi(z)(a) \cdot \sum_{s'\in S} T(s,a,s') \cdot p^*(s')$$

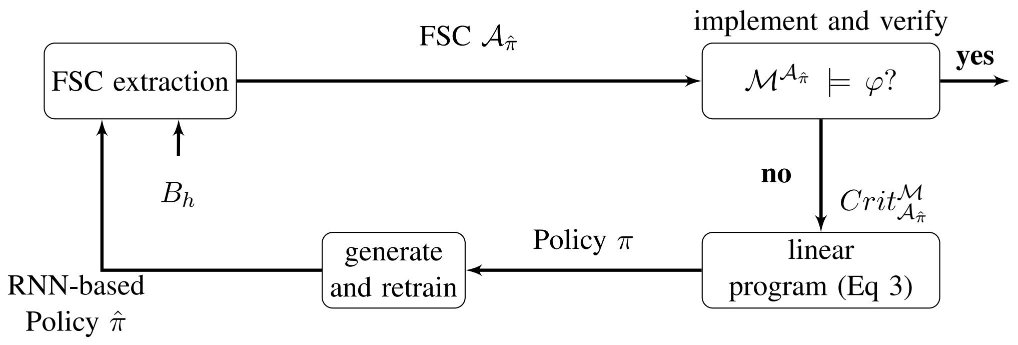

FSC extraction
FSC $\mathcal{A}_{\hat{\pi}}$
implement and verify
$\mathcal{M}^{\mathcal{A}_{\hat{\pi}}} \models \varphi$?
yes
no
$Crit^{\mathcal{M}}_{\mathcal{A}_{\hat{\pi}}}$
B_h
linear program (Eq 3)
Policy π
generate and retrain
RNN-based Policy $\hat{\pi}$

Figure 7: Process for retraining using data from counterexamples.

Basically, we modify the sampling policy π to maximize over the minimal possible worst case probability for critical states, using the original probability p^*. By gathering more sequences of data from these critical decision-points, we locally improve the quality of the policies at those locations and gradually introduce observation-dependencies, see Section 5.3.

From the resulting improved policy π, we generate a new set of observation-action sequences $e_o^* \in \mathsf{ObsSeq}^{\mathcal{M}}_{\mathit{fin}}$, favoring sequences that contain critical states. Recall in Section 4.1.1 that the sequences of data $e_o \in \mathsf{ObsSeq}^{\mathcal{M}}_{\mathit{fin}}$ with initial state s_0 drawn from the initial belief support $s_0 \in \mathrm{supp}(b)$. In contrast, when generating sequences of data using the counterexamples, we draw from the critical set of state-memory node pairs $(n_0, s_0) \in Crit^{\mathcal{M}}_{\mathcal{A}_{\hat{\pi}}}$ for the initial state s_0. In addition, we use the improved sampling policy π, to generate a new set of paths $e^* \in \mathsf{Paths}^{M}_{\mathit{fin}}$. We then convert these paths e^* into observation sequences $e_o^* \in \mathsf{ObsSeq}^{\mathcal{M}}_{\mathit{fin}}$ and retrain the RNN-based policy $\hat{\pi}$ on the new sequences of data.

While this method of sequence generation forms the basis of the proposed supervised learning scheme, it is adaptable to deep reinforcement learning approaches. In particular, one may adapt such a procedure by uniformly sampling from the set of critical states and using them to initialize a given episode in the reinforcement learning procedure. By sampling in this manner, the learned policy will have more data on critical states than a policy learned using just the distribution over the initial states.

Generating new sequences of data from product model counterexamples. In the improvement method outlined above, sequences of data are generated starting at the critical state-memory node pairs (n, s). In contrast, the sequences of data using the product model are generated by initializing using the states s_0 and q_0 from $(n, s_0, q_0) \in Crit^{\mathcal{M}\times\mathcal{R}_\varphi}_{\mathcal{A}_{\hat{\pi}'}}$. Using these starting points along with the improved policy $\pi' \in \Pi^{M\times\mathcal{R}_\varphi}$, the method generates sequences of data $e_o' \in \mathsf{ObsSeq}^{\mathcal{M}\times\mathcal{R}_\varphi}_{\mathit{fin}}$. These sequences of data have the LTL specification encoded and as a result the RNN-based policies trained on them have higher probabilities of satisfying the specification, as shown in Section 5.4.

4.5 The Overall Procedure

Figure 8 shows the overall flow of the proposed method. We start with a POMDP $\mathcal{M}$, a specification φ, an initial RNN-based policy $\hat{\pi}$, a constant η for the entropy-based decision on retraining, and an

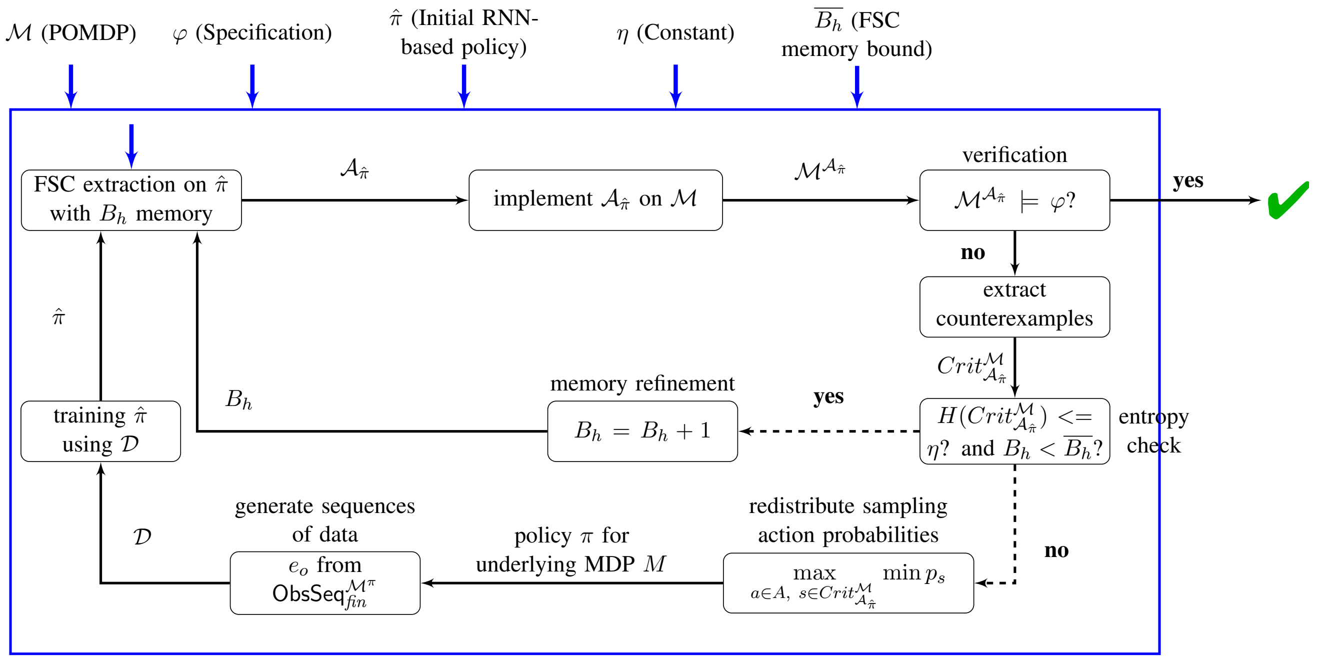

$\mathcal{M}$ (POMDP)
φ (Specification)
$\hat{\pi}$ (Initial RNN-based policy)
η (Constant)
$\overline{B_h}$ (FSC memory bound)
FSC extraction on $\hat{\pi}$ with B_h memory
$\mathcal{A}_{\hat{\pi}}$
implement $\mathcal{A}_{\hat{\pi}}$ on $\mathcal{M}$
$\mathcal{M}^{\mathcal{A}_{\hat{\pi}}}$
verification
$\mathcal{M}^{\mathcal{A}_{\hat{\pi}}} \models \varphi$?
yes
no
extract counterexamples
$Crit^{\mathcal{M}}_{\mathcal{A}_{\hat{\pi}}}$
$H(Crit^{\mathcal{M}}_{\mathcal{A}_{\hat{\pi}}}) <= \eta$? and $B_h < \overline{B_h}$?
entropy check
yes
memory refinement
$B_h = B_h + 1$
B_h
$\hat{\pi}$
training $\hat{\pi}$ using $\mathcal{D}$
no
redistribute sampling action probabilities
$\max_{a \in A,\, s \in Crit^{\mathcal{M}}_{\mathcal{A}_{\hat{\pi}}}} \min p_s$
policy π for underlying MDP M
generate sequences of data
e_o from $\mathsf{ObsSeq}^{\mathcal{M}^{\pi}}_{fin}$
$\mathcal{D}$

Figure 8: Summary flowchart of the RNN-based refinement loop.

upper bound $\overline{B_h}$ on the maximum number of memory nodes for the FSC. At each instance, we extract an FSC $\mathcal{A}_{\hat{\pi}}$ and implement it on the POMDP model $\mathcal{M}$. Verification tools check whether or not the induced behavior of the resulting model satisfies the specification φ. In the case of the latter, we iteratively improve the policy (the FSC) either by adding additional FSC memory states or generating additional sequences of data. Since the underlying problem is undecidable, the procedure is naturally not complete. It is, however, sound as verification yields provable guarantees on the induced behavior. We enforce termination in the experiments by assigning an upper limit on number of times we generate sequences of data.

5. Experimental Results

We evaluate the RNN-based synthesis on benchmark examples that are subject to either LTL specifications or expected reward specifications. For the former, we compare to the tool PRISM-POMDP (Norman et al., 2017), and for the latter to PRISM-POMDP and the point-based solver SolvePOMDP (Walraven & Spaan, 2017). We selected these two solvers from different research communities (formal methods and planning respectively) because they provide the possibility for a straightforward adaption to our benchmark setting. In particular, the tools support undiscounted reward and have a simple and similar input interface. Extended comparisons with Monte-Carlo-based methods, such as POMCP (Silver & Veness, 2010) or reinforcement learning approaches, such as DRQN (Hausknecht & Stone, 2015), are interesting but beyond the scope of this paper.

For a fair comparison, instead of terminating the synthesis procedure once the policy satisfies the specification, we always iterate 10 times, where one iteration encompasses the (re-)training of the RNN-based policy using counterexamples, the FSC extraction, the evaluations, and the policy improvement as detailed in Section 4. For instance, for a specification $\varphi = \mathbb{P}_{\geq\lambda}(\psi)$, we leave the λ

open and seek to compute $\mathbb{P}_{\max}(\psi)$, that is, we compute the minimal probability of satisfying ψ to obtain a policy that satisfies φ. We cannot guarantee to reach that optimum, but we rather improve as far as possible within the predefined 10 iterations. The notions are similar for the expected reward measures $\mathbb{E}_{\geq\lambda}$ and $\mathbb{E}_{\max}$. We will now describe our experimental setup and present detailed results.

5.1 Implementation and Benchmark Set

We created the following Python toolchain to realize the full RNN-based procedure, combining the state-of-the-art tools from deep learning with those from formal verification. First, we use the deep learning library Keras (Ketkar, 2017) to train the RNN-based policy from sequences of data. To evaluate policies, we employ the probabilistic model checkers PRISM (Norman et al., 2017) and STORM (Dehnert et al., 2017) for LTL and undiscounted expected reward respectively. We evaluated on a 2.3 GHz machine with a 12 GB memory limit and a specified maximum computation time of 10^5 seconds. In Tables 3 and 4, TO/MO denote violations of the time/memory limit, respectively and Res. refers to the output value of the induced DTMC.

Temporal logic examples. We examined three problem settings involving motion planning with LTL specifications. For each of the settings, we use a square gridworld of length c with 4 action choices (cardinal directions of movement). The motivation for gridworld examples is that they provide a minimal safety check: a policy that fails to behave safely in such simple environments is also unlikely to behave safely in real-world (Leike et al., 2017). Inside this environment there are a set of static $(\hat{x})$ and moving $(\tilde{x})$ obstacles as well as possible target cells G_1 and G_2, see Figure 9. The agent has a limited visibility region, indicated by the green area, and can infer its state from observations and knowledge of the environment. We define observations as Boolean functions that take as input the positions of the agent and moving obstacles. Intuitively, the functions describe the 8 possible relative positions of the obstacles with respect to the agent inside its viewing range.

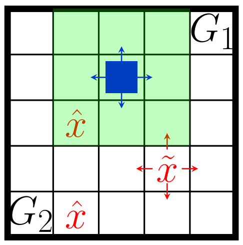


Figure 9: Gridworld environment for LTL examples.

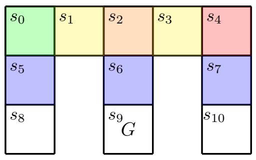
1. **Navigation with moving obstacles** – an agent and a single stochastically moving obstacle. The agent task is to maximize the probability to navigate to a goal state A while not colliding with obstacles (both static and moving): $\varphi_1 = \mathbb{P}_{\max}(\neg X \text{ U } G_1)$ with $x = \hat{x} \cup \tilde{x}$,
2. **Delivery without obstacles** – an agent and static objects (landmarks). The task is to deliver an object from G_1 to G_2 in as few steps as possible: $\varphi_2 = \mathbb{E}_{\max}(\Diamond(G_1 \wedge \Diamond G_2))$.
3. **Slippery delivery with static obstacles** – an agent where the probability of moving perpendicular to the desired direction is 0.1 in each orientation. The task is to maximize the probability to go back and forth from locations G_1 and G_2 without colliding with the static obstacles $\hat{x}$: $\varphi_3 = \mathbb{P}_{\max}(\Box \Diamond G_1 \wedge \Box \Diamond G_2 \wedge \neg \Diamond X)$, with $x = \hat{x}$.

Problem	$\|S\|$	$\|A\|$	$\|Z\|$
Navigation (c)	c^4	4	256
Delivery (c)	c^2	4	256
Slippery (c)	c^2	4	256
Maze(c)	$3c+8$	4	7
Grid(c)	c^2	4	2
RockSample$[4,4]$	257	9	2
RockSample$[5,5]$	801	10	2
RockSample$[7,8]$	12545	13	2

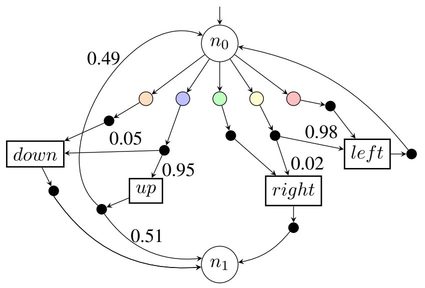
Table 1: Benchmark metrics (LTL examples in gray) .

Expected value benchmarks. For comparison to existing benchmarks, we extend two examples – *Maze*(c) and *Grid*(c) – from PRISM-POMDP for an arbitrary-sized structure. These problems are quite different to the LTL examples, in particular the significantly smaller observation spaces, see Table 1 for details on model sizes. Additionally, we compare against the parametric benchmark *RockSample*$[c, m]$:

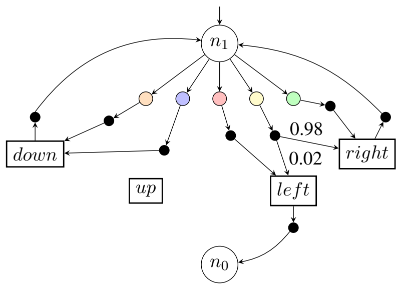
1. ***Maze***(c) with $c + 2$ rows. The agent can only detect its neighboring walls and attempts to reach a goal state G in as few steps as possible, see Figure 10a for Maze(1). Extra rows add uncertainty over the agent's position in the corridors, see the blue observations in Figure 10a.
2. ***Grid***(c), a square grid with length c where the agent attempts to reach a goal state G at the top right of the square. The agent is placed in the grid according to a uniform distribution of the states and can only observe its exact location when it reaches the goal state.
3. ***RockSample***$[c, m]$, a scalable example that models a Mars rover science exploration experiment. The agent attempts to maximize the expected reward based on drilling m rocks in a square grid with length c. Each rock has a binary feature $RockType_i = \{Good, Bad\}$ that indicates whether the rock admits a positive reward. The agent may take an action $Check_i$ that gives a noisy observation about the status of this binary feature. For further details on the action set, the observation and reward functions see (Smith & Simmons, 2004).



(a) Maze(1)



(b) FSC $\mathcal{A}_\pi$ at n_0



(c) FSC $\mathcal{A}_\pi$ at n_1

Figure 10: Maze(1) example with corresponding FSC $\mathcal{A}_\gamma$. The agent's initial state $s_I \in S \backslash \{s_9\}$ is allocated over a uniform distribution and each color represents a different observation. The FSC $\mathcal{A}_\gamma$ has two memory nodes (n_0 and n_1), we prune action mappings and memory updates with low probabilities from 10b and 10c.

Component	Time (s)	%
Total	311.65	100
Training/Retraining RNN	205.77	66.0
Extracting/Implementing FSC	80.21	25.7
Verification of DTMC	2.23	0.7
Counterexamples/entropy check	7.05	2.2

(a) Execution time by component - Navigation (5)

Grid-size	No. of samples (Approx.)	Cross-entropy accuracy
4	1500	0.844
5	4500	0.894
10	10^5	0.852
20*	10^6	0.895
30*	10^6	0.896

(b) Training metrics for RNN policy

Table 2: Metrics of interest for the *Navigation*(c) benchmarks. (a) contains the process execution time for each element in Figure 8 when computing a policy for *Navigation*(5). (b) gives the size of the initial training data and categorical cross-entropy accuracy for training the RNN-based policy trained on this data.

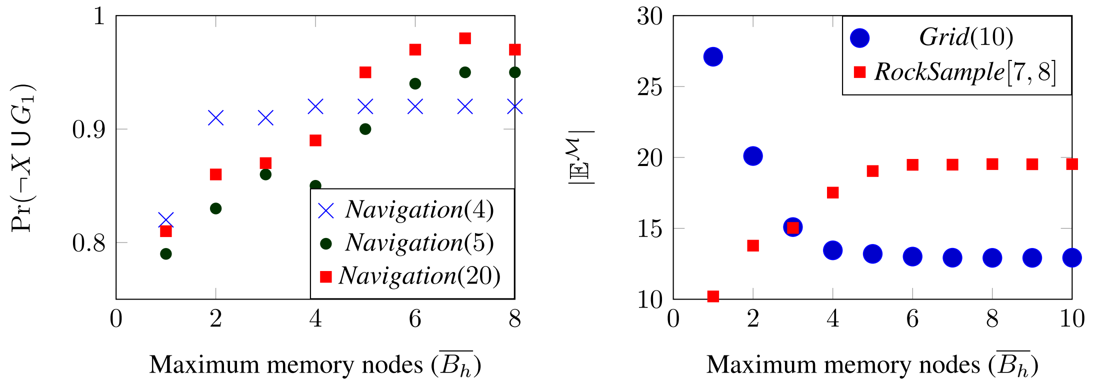
5.2 Training RNNs

Initial policy. In Table 2, we include the size of the training data set used to generate the initial policy, described in 4.1. Additionally, we show how well this initial RNN-based policy matches the data it is trained on using the categorical cross-entropy accuracy. The size of the initial training data set generally scales with the number of states, however, for environments larger than 10^4 states we utilize the samples from smaller environments to train policies for significantly larger state spaces. The predictive accuracy of the RNN-based policy's decisions to those found in the training data is fairly consistent across all problem sizes and does not decrease even when the sequences of training data are generated using similar but smaller environments than the actual POMDP.

Execution breakdown. In Table 2a, we enumerate the proposed method's execution times for the example in *Navigation*(5), see Figure 8 for the location of each row in the process. As the model size increases in the examples, the proportion of the execution time spent extracting/implementing the FSC and verifying the DTMC increases significantly. The primary cause of the timeouts that occur in *Navigation*(40) are due to the significantly longer implementation time required to create and then verify the induced DTMC when the number of states $|S| > 10^6$.

5.3 Policy Improvement

More memory nodes – higher performance. In Figure 11, we show that increasing the number of memory nodes in the FSC produces higher performing policies, both in the form of higher probabilities of satisfying the specification and higher undiscounted expected rewards. In the case of *RockSample*$[7, 8]$ the 1-FSC performs particularly poorly as the agent needs memory to keep track of whether an observed rock is worth drilling. Therefore, the highest performing 1-FSC for *RockSample*$[7, 8]$ involves the agent drilling nothing and driving to the exit area to obtain the reward $R = 10$. Another noticeable characteristic is that for each FSC in Figure 11, there is a point of diminishing returns where the additional memory does not produce higher quality policies. In most cases, this point falls between 6 and 8 memory nodes. As a consequence for the set of benchmarks, unless otherwise specified, we set the upper bound for the number of memory nodes at $\overline{B_h} = 8$. However, if one was to perform task and model analysis prior to synthesis, they could lower this bound and thus reduce the synthesis time. For instance, while the observation functions



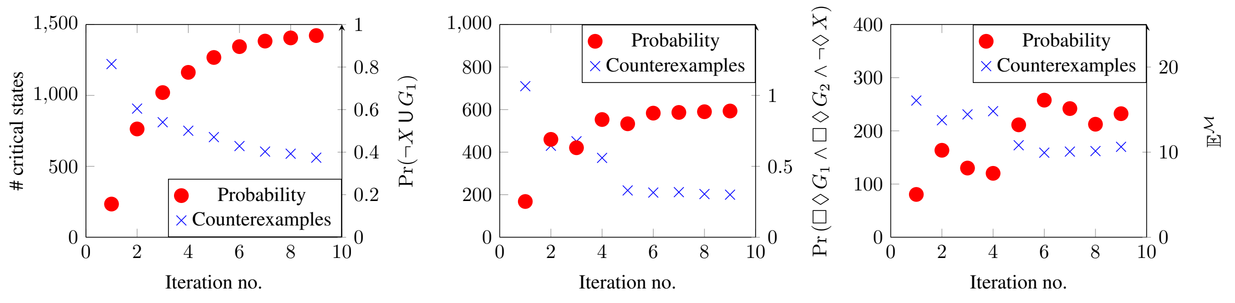
(a) LTL examples

(b) Expected reward examples

Figure 11: Performance of FSCs as the number of memory nodes increases. Each point represents the best performing policy over 10 iterations of the lower loop in Figure 8. Each FSC begins to experience diminishing returns at values of $B_h = 6$ and higher. In (b) the policy attempts to maximize the expected reward in *Grid*(10) and *RockSample*[7, 8]. Note: *Grid*(10) admits negative rewards for each agent move and consequently policies with additional memory nodes accumulate fewer negative rewards.

of *Navigation(*4*)*, *Navigation(*5*)* and *Navigation(*20*)* are the same, the smaller area in the former produces a model with a smaller observation set. Consequently, the performance plateau occurs much earlier, around $\overline{B_h} = 2$, in *Navigation(*4*)* than in the larger models.

Counterexample data – trains better policies. Figure 12 compares the number of critical states in a set of counterexamples in relation to the probability of satisfying an LTL specification in each iteration of re-training for the proposed method. In particular, we depict the size of the set of critical states $Crit^{\mathcal{M}}_{\mathcal{A}_{\hat{\gamma}}} \subset S$ regarding the specification φ. Note that even if the probability to satisfy the LTL specification is nearly one (for the initial set of states in the POMDP), there may still be critical intermediate states. Likewise in Figure 12c, even if the expected reward for the initial set of states is near the maximum value, there may still be intermediate states that are deemed critical as they fall below an expected reward threshold. In Figure 12 as the satisfaction probability and the expected reward increases, the number of the critical states identified by the verification decreases. In particular, the retraining of the RNN-based policy on the sequences of data generated using the local improvement step (Eq. 3, Section 4.4) is effective in improving the policy with each iteration. As outlined earlier, for the experiments we upper bounded the number of calls for training from additional sequences of data at 10 iterations. As Figure 12 shows, the policy generally would not significantly improve beyond 8 iterations. Similarly to the case with the memory bound, one may be able to account for task and model features to select lower values of the data sampling bound. In the case of Figures 12a and 12b, the lack of a moving obstacle greatly reduces the size of the observation set and thus the policy needs less sequences of data to converge.

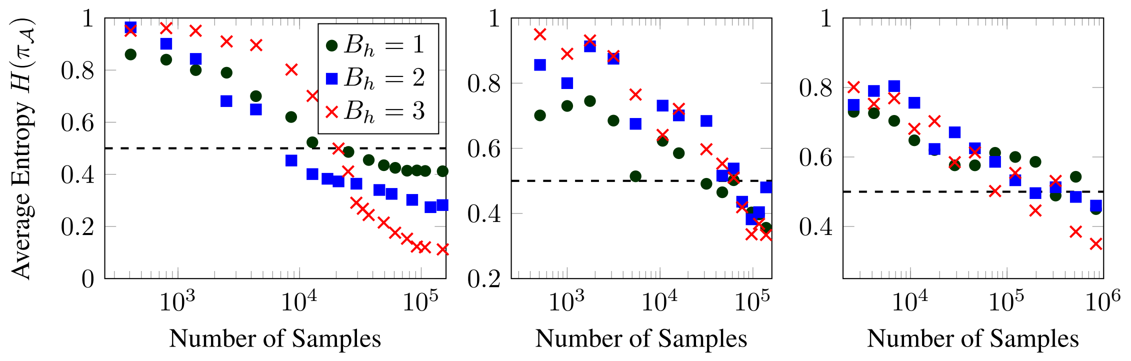


(a) *Navigation*(5) (b) *Slippery*(5) (c) *RockSample*$[4, 4]$

Figure 12: Progression of the number of critical states and the probability of satisfying an LTL specification (a,b) or maximizing an expected reward (c) as a result of retraining on new sequences of data generated using the local improvement steps.

Counterexample data – less arbitrary decisions. In Figure 13, we ignore the decision at the entropy check, fix the memory precision and iteratively add more sequences of data generated using the counterexamples. Each point in Figure 13 in represents one instance of verification in the loop in Figure 8. As the RNN-based policy iteratively trains on additional sequences of data, the subsequent extracted policy makes less arbitrary decisions, shown in Figure 13 by the decrease in entropy of the FSC as the RNN-based policy is trained on larger sets of training sequences.

More memory nodes – less arbitrary decisions. In Figure 13, we also compare how the number of memory nodes in the extracted FSC correlate to the entropy of the decisions at critical states. When trained on a large set of sequences of training data, the FSCs with a higher number of memory nodes have a lower entropy than those without. This behavior is likely due the fact that extracted FSC with more memory nodes can better approximate the RNN-based policy, which itself is making less arbitrary decisions due to the larger training set. Meanwhile, when the extracted FSCs are approximating RNN-based policies trained on smaller sets of training sequences, they generally



(a) *Navigation*(5) (b) *Slippery* (5) (c) *RockSample*$[4, 4]$

Figure 13: Entropy of the extracted FSCs from an RNN as it is trained with more samples. Each point represents an extracted FSC, for color sequence we fix the discretization and add more samples guided by the counterexamples. A sample threshold of $\mu = 0.5$ is also indicated by the dashed line.

			PRISM-POMDP		Handcrafted FSC		Automated Extraction		Automated Extraction with LTL Automaton	
Problem	States	Type, φ	Res.	Time (s)	Res.	Time (s)	Res.	Time (s)	Res.	Time (s)
Navigation (3)	333	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	**0.84**	73.88	0.74	**14.16**	0.80	123.14	0.80	100.20
Navigation (4)	1088	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	**0.93**†	1034.64	0.82	**22.67**	0.92	160.32	0.91	177.91
Navigation (4) [2-FSC]	13373	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	–	–	0.92	47.26	–	–	–	–
Navigation (4) [8-FSC]	53477	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	–	–	**0.92**	85.26	–	–	–	–
Navigation (5)	2725	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	MO	MO	0.91	34.91	**0.95**	311.65	**0.95**	451.22
Navigation (5) [2-FSC]	33357	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	–	–	0.92	159.61	–	–	–	–
Navigation (5) [8-FSC]	133413	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	–	–	**0.92**	253.11	–	–	–	–
Navigation (10)	49060	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	MO	MO	0.79	**822.87**	**0.85**	2561.02	**0.85**	3210.19
Navigation (10) [2-FSC]	475053	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	–	–	0.83	1185.41	–	–	–	–
Navigation (10) [8-FSC]	1900197	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	–	–	0.83	1488.77	–	–	–	–
Navigation (20)	798040	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	MO	MO	0.96	**4712.25***	**0.98***	8173.03*	**0.98**	10574.33
Navigation (30)	4045840	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	MO	MO	0.95	**25191.05***	**0.97***	61350.34*	TO	TO
Navigation (40)	–	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_1$	MO	MO	TO	TO	TO	TO	TO	TO
Delivery (4)	80	$\mathbb{E}^{\mathcal{M}}_{\max}, \varphi_2$	**-6.0**	**28.53**	-6.02	35.35	-6.04	94.32	-6.01	120.12
Delivery (5)	125	$\mathbb{E}^{\mathcal{M}}_{\max}, \varphi_2$	**-8.0**	102.41	-8.11	**78.32**	-8.13	150.44	-8.04	180.31
Delivery (10)	500	$\mathbb{E}^{\mathcal{M}}_{\max}, \varphi_2$	MO	MO	-18.13	**120.34**	-18.13	347.98	**-18.10**	400.14
Slippery (4)	460	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_3$	**0.90**	**5.10**	0.78	67.51	0.80	180.15	0.85	200.67
Slippery (5)	730	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_3$	**0.93**	**83.24**	0.89	84.32	0.89	212.79	0.91	355.00
Slippery (10)	2980	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_3$	MO	MO	**0.98**	**119.14**	**0.98**	280.55	**0.98**	651.47
Slippery (20)	11980	$\mathbb{P}^{\mathcal{M}}_{\max}, \varphi_3$	MO	MO	**0.99**	**1580.42**	**0.99**	2384.56	**0.99**	3312.45

Table 3: Computing policies for examples with LTL specifications.

make arbitrary decisions (see top left of Figure 13a). In these cases, the FSC with more memory nodes tend to make more arbitrary decisions than those with less, which is likely a function of an under-defined hidden state update $\hat{\delta}$ in the RNN-based policy.

5.4 Comparisons

In Tables 3 and 4, we compare the state-of-the-art POMDP solvers to three different approaches to training and extracting policies, which are *Handcrafted FSC*, *Automated Extraction* and *Automated Extraction with LTL Automaton*. The handcrafted FSC describes an approach where the memory updates of the FSC are handcrafted with some intuition about the environment (Carr et al., 2019). The other two both use the automated method of extraction detailed in Section 4.2, one generating sequences of data and verifying against the POMDP $\mathcal{M}$. The other generates sequences of data and performs verification on the larger product model $\mathcal{M} \times \mathcal{R}_{\varphi}$.

Tool comparison – LTL examples. Summarized, the method scales to significantly larger domains than PRISM-POMDP with competitive computation times. Naturally the policies produced by the procedure will not have higher maximum probabilities (or lower minimum expected cost) than those generated by the PRISM-POMDP tool. As mentioned before, there is an inherent level of randomness in extracting a policy. While we always take the first shot result for our experiments, the quality of policies may be improved by sampling several FSC extractions of the RNN-based policy. In the larger environments, $Navigation(20)$ and upwards[*], we employ the data generation technique outlined at the end of Section 4.1.1 on a similar environment $\mathcal{M}_s$ with grid-size $c = 10$.

[*]Sequences of data generated using similar but smaller environment.

		PRISM-POMDP		pomdpSolve		Handcrafted FSC			Automated Extraction		
Problem	Type	Res	Time (s)	Res	Time (s)	States	Res	Time (s)	States	Res	Time (s)
Maze (1)	$\mathbb{E}^{\mathcal{M}}_{\max}$	**-4.30**	**0.09**	-4.30	0.30	68	-4.31	31.70	68	-4.33	80.31
Maze (2)	$\mathbb{E}^{\mathcal{M}}_{\max}$	-5.23	2.176	**-5.23**	**0.67**	83	-5.31	46.65	74	-5.34	114.23
Maze (5)	$\mathbb{E}^{\mathcal{M}}_{\max}$	-13.00[†]	4110.50	**-12.04**	132.12	128	-14.40	**68.09**	92	-13.29	160.12
Maze (10)	$\mathbb{E}^{\mathcal{M}}_{\max}$	MO	MO	MO	MO	203	-100.21	**158.33**	190	**-23.02**	210.01
Grid (3)	$\mathbb{E}^{\mathcal{M}}_{\max}$	-2.88	2.332	**-2.88**	**0.07**	165	-2.90	38.94	186	-2.90	87.31
Grid (4)	$\mathbb{E}^{\mathcal{M}}_{\max}$	-4.13	1032.53	**-4.13**	**0.77**	381	-4.32	79.99	672	-4.20	124.31
Grid (5)	$\mathbb{E}^{\mathcal{M}}_{\max}$	MO	MO	**-5.42**	**1.94**	727	-6.62	91.42	1350	-5.91	250.14
Grid (10)	$\mathbb{E}^{\mathcal{M}}_{\max}$	MO	MO	MO	MO	5457	-13.63	**268.40**	5500	**-12.92**	1031.21
Grid (25)	$\mathbb{E}^{\mathcal{M}}_{\max}$	MO	MO	MO	MO	90000	-531.05	**622.31**	62100	**-35.32**	6514.30
RockSample$[4,4]$	$\mathbb{E}^{\mathcal{M}}_{\max}$	N/A	N/A	**18.04**	**0.43**	2432	17.71	35.35	2427	16.11	113.61
RockSample$[5,5]$	$\mathbb{E}^{\mathcal{M}}_{\max}$	N/A	N/A	**19.23**	621.28	8320	18.40	**43.74**	8400	15.00	250.50
RockSample$[7,8]$	$\mathbb{E}^{\mathcal{M}}_{\max}$	N/A	N/A	**21.46**[†]	20458.41	166656	20.32	**860.53**	152430	19.53	2146.49

Table 4: Comparison for maximing an expected reward for a set of POMDP benchmarks.

Since it is observation-dependent, this policy $\mathcal{A}_{\hat{\gamma}}$ scales to the larger POMDP $\mathcal{M}$ even when trained on observation sequences $e_o \in \mathsf{ObsSeq}^{\mathcal{M}_s}_{\mathit{fin}}$ generated on a smaller state space S_s.

Tool comparison – POMDP benchmarks. The method compares favorably with PRISM-POMDP and pomdpSolve for *Maze*(c) and *Grid*(c) (full set of results in Table 4[†]). However, the proposed method performs poorly in comparison to pomdpSolve for the *RockSample* problems. In *RockSample*, an observation is received after taking an action to $Check_i$ a particular rock. Since QMDP approach to sampling assumes that the problem will eventually be fully observable (Cassandra, 1998), this action never appears when generating sequences of data using the method in Section 4.1. To achieve the results in Table 4, we applied, a far from optimal, method that forces the agent to periodically explore by checking the closest rock with a certain probability. Note, that the primary aim of this work is to provide provable guarantees on RNN-based policies in POMDPs with respect to LTL specifications rather than finding the optimal method of computing RNN-based policies that maximize an expected reward using reinforcement learning.

Handcrafted comparison. In Table 3, the proposed automated method scales to significantly larger examples than both state-of-the-art POMDP solvers which compute near-optimal policies. While the handcrafted approach scales equally well, the automated extraction method produces higher-quality policies - within 2% of the optimum. In Table 4, we observe similar improvement via the automated extraction method. Note that an optimal policy for *Maze*(1) can be expressed using 2 memory states. The FSC structure employed by the handcrafted method uses this structure and consequently, for the small Maze environments, the handcrafted method produces FSCs that have higher expected rewards than the equivalent automatically extracted one. Yet, with larger environments the fixed memory structure in the Handcrafted approach produces poor policies as additional memory nodes are beneficial to account for the past behavior, see Section 5.3.

Product model comparison. In *Navigation*(c), there is an indistinguishable difference between the automated extraction with and without the product model. Noting that DRAs $\mathcal{R}_\varphi$ for specifications involving only a single until (e. g. $\varphi = \neg X \cup G_1$) or eventually (e. g. $\varphi = \lozenge\, G_1$) do not have any effect as these specifications are already a direct reachability problem (Bouton et al., 2020).

[†]Output was a bound; we give the worst-case value from bound.

However, using the product models for *Delivery*(c) $\mathcal{M} \times \mathcal{R}_{\varphi_2}$ and *Slippery*(c) $\mathcal{M} \times \mathcal{R}_{\varphi_3}$ produces extracted policies that have higher probabilities of satisfying the specifications, especially in the smaller environments. As discussed in Section 4.4, this improvement is a function of the higher quality counterexamples, which describe both the states in the POMDP $s \in S$ and in the automaton $q \in Q$ where that lead to the conditions where specification does not hold. However, this increase in probability comes at a cost of an approximately 150% increase in computation time. In the case of the larger environments such as *Delivery(10)* and *Slippery(20)*, using the product model only provides limited improvement. In the case of the former, the increased state space likely leads to a relatively sparse observation space, which limits the impact when learning the belief using the product model. In the latter, the policy was already performing near the maximal possible value of probability 1 and so additional improvement would be difficult to measure.

5.5 Precision-Performance Trade-Off

As discussed in Section 5.3, increasing the number of memory states in the FSC produces policies with higher probabilities of satisfying the specification and greater expected rewards. Table 3, we include the sizes of the FSCs for the handcrafted procedure to demonstrate the trade-off between computational tractability and expressivity: a larger FSC means that the policy can store more information, which may lead to better decisions. However, larger FSCs require more computational effort and may require more sequences of data for training the RNN-based policy. Figure 10 shows the automatically extracted FSC for the *Maze*(1) environment. Note that a 2-FSC can represent the optimal *Maze*(1) policy. The FSC shown in Figures 10b and 10c is very close to this optimal policy. The stochastic action choices at $(n_0, blue)$ and at $(n_1, yellow)$ create the suboptimality in this example with the optimal policy taking the respective up and $right$ actions at these points.

6. Conclusion

In this paper we presented a comprehensive and concerted method to train, verify, and improve policies based on recurrent neural networks in a model-based setting with POMDPs. The key result is that we are able to use formal verification to provide correctness guarantees and yet exploit the efficiency of recurrent neural networks as black-box policy representations. Our results show the effectiveness and efficiency of each part of the process and even demonstrate a clear competitiveness to state-of-the-art POMDP solvers. In the future, we will expand the proposed approach to exploit state-of-the-art approaches to deep reinforcement learning on POMDPs. In doing so, we require fewer sequences of data to train the RNN-based policies and subsequently reduce the time required synthesize policies that ensure that the agent satisfies the temporal logic specification.

Acknowledgements

Steven Carr and Ufuk Topcu were supported by the grants DARPA D19AP00004, ONR N00014-18-1-2829, and ARL ACC-APG-RTP W911NF. Nils Jansen was supported by the grant NWO OCENW.KLEIN.187.

References

Ahmadi, M., Jansen, N., Wu, B., & Topcu, U. (2020). Control theory meets pomdps: A hybrid systems approach. *IEEE Transactions on Automatic Control*.

Akintunde, M. E., Kevorchian, A., Lomuscio, A., & Pirovano, E. (2019). Verification of rnn-based neural agent-environment systems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (pp. 6006–6013). AAAI Press.

Amir, G., Wu, H., Barrett, C. W., & Katz, G. (2021). An smt-based approach for verifying binarized neural networks. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Vol. 12652, pp. 203–222). Springer.

Bai, H., Cai, S., Ye, N., Hsu, D., & Lee, W. S. (2015). Intention-aware online pomdp planning for autonomous driving in a crowd. In *International Conference on Robotics and Automation (ICRA)* (pp. 454–460). IEEE.

Baier, C., Größer, M., & Bertrand, N. (2012). Probabilistic ω-automata. *Journal of the ACM*, *59*(1), 1:1–1:52.

Baier, C., & Katoen, J.-P. (2008). *Principles of model checking*. MIT Press.

Bakker, B. (2001). Reinforcement learning with long short-term memory. In *Advances in Neural Information Processing Systems (NIPS)* (pp. 1475–1482). MIT Press.

Baum, E. B., & Wilczek, F. (1987). Supervised learning of probability distributions by neural networks. In *Advances in Neural Information Processing Systems (NIPS)* (pp. 52–61). American Institue of Physics.

Bouton, M., Tumova, J., & Kochenderfer, M. J. (2020). Point-based methods for model checking in partially observable markov decision processes. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (pp. 10061–10068). AAAI Press.

Carr, S., Jansen, N., & Topcu, U. (2020). Verifiable rnn-based policies for pomdps under temporal logic constraints. In *International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 4121–4127). IJCAI.org.

Carr, S., Jansen, N., Wimmer, R., Fu, J., & Topcu, U. (2018). Human-in-the-loop synthesis for partially observable Markov decision processes. In *American Control Conference (ACC)* (pp. 762–769). IEEE.

Carr, S., Jansen, N., Wimmer, R., Serban, A. C., Becker, B., & Topcu, U. (2019). Counterexample-guided strategy improvement for pomdps using recurrent neural networks. In *International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 5532–5539). IJCAI.org.

Cassandra, A. R. (1998). A survey of pomdp applications. In *Working Notes of the AAAI 1998 Fall Symposium on Planning with Partially Observable Markov Decision Processes* (Vol. 1724). AAAI Press.

Chatterjee, K., Chmelik, M., & Davies, J. (2016). A symbolic sat-based algorithm for almost-sure reachability with small strategies in pomdps. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (pp. 3225–3232). AAAI Press.

Chatterjee, K., Chmelík, M., Gupta, R., & Kanodia, A. (2015). Qualitative analysis of pomdps with temporal logic specifications for robotics applications. In *International Conference on Robotics and Automation (ICRA)* (pp. 325–330). IEEE.

Chatterjee, K., Chmelík, M., Gupta, R., & Kanodia, A. (2016). Optimal cost almost-sure reachability in pomdps. *Artificial Intelligence*, *234*, 26–48.

Clarke, E. M., Henzinger, T. A., Veith, H., & Bloem, R. (2018). *Handbook of model checking*

(Vol. 10). Springer.

Cover, T. M., & Thomas, J. A. (2012). *Elements of information theory*. John Wiley & Sons.

Cubuktepe, M., Jansen, N., Junges, S., Marandi, A., Suilen, M., & Topcu, U. (2021). Robust finite-state controllers for uncertain pomdps. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (pp. 11792–11800). AAAI Press.

Dehnert, C., Junges, S., Katoen, J., & Volk, M. (2017). A storm is coming: A modern probabilistic model checker. In *Computer Aided Verification (CAV)* (Vol. 10427, pp. 592–600). Springer.

Ghosh, B., & Neider, D. (2020). A formal language approach to explaining rnns. *CoRR*, *abs/2006.07292*.

Goodfellow, I. J., Bengio, Y., & Courville, A. C. (2016). *Deep learning*. MIT Press.

Goodfellow, I. J., & Vinyals, O. (2015). Qualitatively characterizing neural network optimization problems. In *International Conference on Learning Representations (ICLR).*

Hadfield-Menell, D., Milli, S., Abbeel, P., Russell, S. J., & Dragan, A. D. (2017). Inverse reward design. In *Advances in Neural Information Processing Systems (NIPS)* (pp. 6765–6774). MIT Press.

Hausknecht, M. J., & Stone, P. (2015). Deep recurrent Q-learning for partially observable MDPs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (pp. 29–37). AAAI Press.

Heess, N., Hunt, J. J., Lillicrap, T. P., & Silver, D. (2015). Memory-based control with recurrent neural networks. *CoRR*, *1512.04455*.

Heess, N., Wayne, G., Silver, D., Lillicrap, T., Erez, T., & Tassa, Y. (2015). Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems (NIPS)* (pp. 2944–2952). MIT Press.

Hernandez-Gardiol, N., & Mahadevan, S. (2000). Hierarchical memory-based reinforcement learning. In *Advances in Neural Information Processing Systems (NIPS)* (pp. 1047–1053). MIT Press.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780.

Huang, X., Kwiatkowska, M., Wang, S., & Wu, M. (2017). Safety verification of deep neural networks. In *Computer Aided Verification (CAV)* (Vol. 10426, pp. 3–29). Springer.

Jaakkola, T., Singh, S. P., & Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable markov decision problems. In *Advances in Neural Information Processing Systems (NIPS)* (pp. 345–352). MIT Press.

Jansen, N., Wimmer, R., Ábrahám, E., Zajzon, B., Katoen, J., Becker, B., & Schuster, J. (2014). Symbolic counterexample generation for large discrete-time markov chains. *Science of Computer Programming*, *91*, 90–114.

Junges, S., Jansen, N., & Seshia, S. A. (2021). Enforcing almost-sure reachability in pomdps. In *Computer Aided Verification (CAV)* (Vol. 12760, pp. 602–625). Springer.

Junges, S., Jansen, N., Wimmer, R., Quatmann, T., Winterer, L., Katoen, J.-P., & Becker, B. (2018). Finite-state controllers of pomdps via parameter synthesis. In *Conference on Uncertainty in Artificial Intelligence (UAI)* (pp. 519–529). AUAI Press.

Junges, S., Katoen, J., Pérez, G. A., & Winkler, T. (2021). The complexity of reachability in parametric markov decision processes. *Journal of Computer and System Sciences*, *119*, 183–210.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, *101*(1), 99–134.

Katz, G., Barrett, C., Dill, D. L., Julian, K., & Kochenderfer, M. J. (2017). Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification (CAV)* (pp. 97–117). Springer.

Kearns, M. J., Mansour, Y., & Ng, A. Y. (1999). Approximate planning in large pomdps via reusable trajectories. In *Advances in Neural Information Processing Systems (NIPS)* (pp. 1001–1007). MIT Press.

Ketkar, N. (2017). Introduction to keras. In *Deep learning with python* (pp. 97–111). Springer.

Khmelnitsky, I., Neider, D., Roy, R., Barbot, B., Bollig, B., Finkel, A., ... Ye, L. (2020). Property-directed verification of recurrent neural networks. *CoRR*, *abs/2009.10610*.

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

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

Kretínský, J., Meggendorfer, T., Sickert, S., & Ziegler, C. (2018). Rabinizer 4: From LTL to your favourite deterministic automaton. In *Computer Aided Verification (CAV)* (Vol. 10981, pp. 567–577). Springer.

Kurniawati, H., Hsu, D., & Lee, W. S. (2008). SARSOP: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *Robotics: Science and Systems (RSS).* MIT Press.

Kwiatkowska, M., Norman, G., & Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification (CAV)* (Vol. 6806, pp. 585–591). Springer.

Leike, J., Martic, M., Krakovna, V., Ortega, P. A., Everitt, T., Lefrancq, A., ... Legg, S. (2017). AI safety gridworlds. *CoRR*, *abs/1711.09883*.

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

Madani, O., Hanks, S., & Condon, A. (1999). On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (pp. 541–548). AAAI Press.

Meuleau, N., Kim, K., Kaelbling, L. P., & Cassandra, A. (1999). Solving pomdps by searching the space of finite policies. In *Conference on Uncertainty in Artificial Intelligence (UAI)* (pp. 417–426). Morgan Kaufmann.

Michalenko, J. J., Shah, A., Verma, A., Baraniuk, R. G., Chaudhuri, S., & Patel, A. B. (2019). *CoRR*, *abs/1902.10297*.

Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529.

Mulder, W. D., Bethard, S., & Moens, M. (2015). A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language*, *30*(1), 61–98.

Norman, G., Parker, D., & Zou, X. (2017). Verification and control of partially observable probabilistic systems. *Real-Time Systems*, *53*(3), 354-402.

Pascanu, R., Gulcehre, C., Cho, K., & Bengio, Y. (2014). How to construct deep recurrent neural networks. In *International Conference on Learning Representations (ICLR).*

Pnueli, A. (1977). The temporal logic of programs. In *Annual Symposium on Foundations of Computer Science (FOCS)* (pp. 46–57). IEEE Computer Society.

Poupart, P., & Boutilier, C. (2003). Bounded finite state controllers. In *Advances in Neural Information Processing Systems (NIPS)* (pp. 823–830). MIT Press.

Puterman, M. L. (1994). *Markov decision processes*. John Wiley and Sons.

Shani, G., Pineau, J., & Kaplow, R. (2013). A survey of point-based pomdp solvers. *Autonomous Agents and Multi-Agent Systems*, *27*(1), 1–51.

Sharan, R., & Burdick, J. (2014). Finite state control of pomdps with ltl specifications. In *American Control Conference (ACC)* (pp. 501–508). IEEE.

Sherstinsky, A. (2020). Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, *404*, 132306.

Silver, D., & Veness, J. (2010). Monte-carlo planning in large pomdps. In *Advances in Neural Information Processing Systems (NIPS)* (pp. 2164–2172). MIT Press.

Smith, T., & Simmons, R. (2004). Heuristic search value iteration for pomdps. In *Conference on Uncertainty in Artificial Intelligence (UAI)* (pp. 520–527). AUAI Press.

Spaan, M. T. J., & Vlassis, N. (2005). Perseus: Randomized point-based value iteration for pomdps. *Journal of Artificial Intelligence Research (JAIR)*, *24*, 195–220.

Suilen, M., Jansen, N., Cubuktepe, M., & Topcu, U. (2020). Robust policy synthesis for uncertain pomdps via convex optimization. In *International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 4113–4120). ijcai.org.

Walraven, E., & Spaan, M. (2017). Accelerated vector pruning for optimal pomdp solvers. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (pp. 3672–3678). AAAI Press.

Wang, Q., Zhang, K., Liu, X., & Giles, C. L. (2018). Verification of recurrent neural networks through rule extraction. *CoRR*, *abs/1811.06029*.

Wang, Y., Chaudhuri, S., & Kavraki, L. E. (2018). Bounded policy synthesis for pomdps with safe-reachability objectives. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)* (pp. 238–246). ACM.

Weiss, G., Goldberg, Y., & Yahav, E. (2018). Extracting automata from recurrent neural networks using queries and counterexamples. In *International Conference on Machine Learning (ICML)* (Vol. 80, pp. 5244–5253). PMLR.

Wierstra, D., Förster, A., Peters, J., & Schmidhuber, J. (2007). Solving deep memory pomdps with recurrent policy gradients. In *International Conference on Artificial Neural Networks (ICANN)* (pp. 697–706). Springer.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*, 229–256.

Wimmer, R., Jansen, N., Ábrahám, E., Katoen, J., & Becker, B. (2014). Minimal counterexamples for linear-time probabilistic verification. *Theoretical Computer Science*, *549*, 61–100.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2021). A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, *32*(1), 4–24.

Zeng, Z., Goodman, R. M., & Smyth, P. (1993). Learning finite state machines with self-clustering recurrent networks. *Neural Computation*, *5*(6), 976–990.

Zhang, Z., Hsu, D., & Lee, W. S. (2014). Covering number for efficient heuristic-based pomdp planning. In *International Conference on Machine Learning (ICML)* (Vol. 32, pp. 28–36). JMLR.org.