Verifiable RNN-Based Policies for POMDPs Under Temporal Logic Constraints

Steven Carr[1] , Nils Jansen[2] and Ufuk Topcu[1]
[1]The University of Texas at Austin
[2]Radboud University, Nijmegen, The Netherlands
stevencarr@utexas.edu, n.jansen@science.ru.nl

Abstract

Recurrent neural networks (RNNs) have emerged as an effective representation of control policies in sequential decision-making problems. However, a major drawback in the application of RNN-based policies is the difficulty in providing formal guarantees on the satisfaction of behavioral specifications, e.g. safety and/or reachability. By integrating techniques from formal methods and machine learning, we propose an approach to automatically extract a finite-state controller (FSC) from an RNN, which, when composed with a finite-state system model, is amenable to existing formal verification tools. Specifically, we introduce an iterative modification to the so-called quantized bottleneck insertion technique to create an FSC as a randomized policy with memory. For the cases in which the resulting FSC fails to satisfy the specification, verification generates diagnostic information. We utilize this information to either adjust the amount of memory in the extracted FSC or perform focused retraining of the RNN. While generally applicable, we detail the resulting iterative procedure in the context of policy synthesis for partially observable Markov decision processes (POMDPs), which is known to be notoriously hard. The numerical experiments show that the proposed approach outperforms traditional POMDP synthesis methods by 3 orders of magnitude within 2% of optimal benchmark values.

1 Introduction

Research in the reinforcement and supervised learning communities has demonstrated the utility of recurrent neural networks (RNNs) in synthesizing control policies in domains that exhibit temporal behavior [Tsoi and Back, 1997; Bakker, 2001]. The internal memory states of RNNs, such as in long short-term memory (LSTM) architectures [Hochreiter and Schmidhuber, 1997], effectively account for temporal behavior by capturing the history from sequential information [Pascanu *et al.*, 2014]. Furthermore, in applications that suffer from incomplete information, RNNs leverage history to act as either a state or value estimator [Wierstra *et al.*, 2007] or as a control policy [Hausknecht and Stone, 2015].

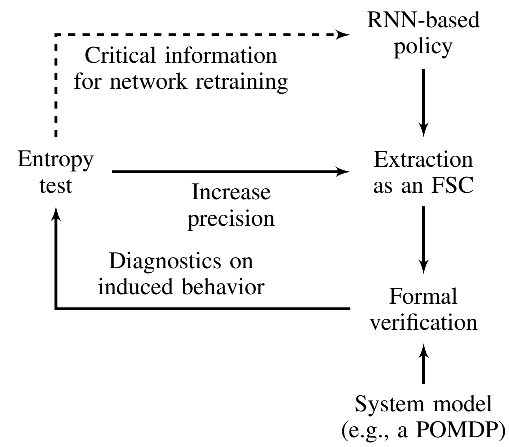


Figure 1: High-level iterative policy extraction process.

In safety-critical systems such as autonomous vehicles, policies that are guaranteed to prevent unsafe behavior are necessary. We seek to provide formal guarantees for policies represented by RNNs with respect to temporal logic [Pnueli, 1977] or reward specifications. Such a verification task is, in general, hard due to the complex, often non-linear, structures of RNNs [Mulder *et al.*, 2015]. Existing work directly employs satisfiability-modulo-theories (SMT) [Wang *et al.*, 2018] or mixed-integer linear program (MILP) [Akintunde *et al.*, 2019], however, such methods not only scale exponentially in the number of variables but also rely on constructions using only rectified linear units (ReLUs).

We take an iterative and model-based approach, see Fig. 1. We extract a policy from a given RNN in the form of a finite-state controller (FSC) [Poupart and Boutilier, 2003]. First, we employ a modification of a discretization technique called *quantized bottleneck insertion*, introduced in [Koul *et al.*, 2019]. Basically, the discretization facilitates a mapping of the continuous memory structure of the RNN to a pre-defined number of discrete memory states and transitions of an FSC.

However, this standalone FSC without a formal model is often not sufficient to prove meaningful properties. The proposed approach relies on the exact behavior a policy induces on a specific application that can be modeled formally. We apply the extracted FSC directly to a formal model, and the resulting restricted model is amenable for efficient verification techniques that certify whether a specification is satisfied [Baier and Katoen, 2008].

If the specification does not hold, verification methods typically provide diagnostic information on critical parts of the model in the form of so-called counterexamples. We propose to utilize such counterexamples to identify *improvements* in the extracted FSC or in the underlying RNN. First, increasing the amount of memory states in the FSC may help to approximate the behavior of the RNN more precisely [Koul *et al.*, 2019]. Second, the RNN may actually require further training data to induce higher-quality policies for the particular application. Existing approaches rely, for example, on loss visualization [Goodfellow and Vinyals, 2015], but we strive to exploit the information we can gain from the concrete behavior of the RNNs with respect to a formal model. Therefore, in order to decide whether more data are needed in the training of the RNN or whether first the number of memory states in the FSC should be increased, we identify those critical decisions of the current FSC that are "arbitrary". Basically, we measure the entropy [Cover and Thomas, 2012] of each stochastic choice over actions according to the current FSC-based policy at critical states. That is, if the entropy is high, the decision is deemed arbitrary despite its criticality and further training is required.

We showcase the applicability of the proposed method on *partially observable Markov decision processes (POMDPs).* With their ability to represent sequential decision-making problems under uncertainty and incomplete information, these models are of particular interest in planning and control [Cassandra, 1998]. Despite their utility as a modeling formalism and recent algorithmic advances, policy synthesis for POMDPs is hard both theoretically and practically [Meuleau *et al.*, 1999]. For reasons outlined earlier, RNNs have recently emerged as efficient policy representations for POMDPs [Hausknecht and Stone, 2015]. We detail the proposed approach on POMDPs and combine the scalability and flexibility of an RNN representation with the rigor of formal verification to synthesize POMDP policies that adhere to temporal logic specifications.

We demonstrate the effectiveness of the proposed synthesis approach on a set of POMDP benchmarks. These benchmarks allows for a comparison to well-known POMDP solvers, both with and without temporal logic specifications. The numerical examples show that the proposed method (1) is more scalable, by up to 3 orders of magnitude, than well-known POMDP solvers and (2) achieves higher-quality results in terms of the measure of interest than other synthesis methods that extract FSCs.

Related work. Closest to the proposed method is [Carr *et al.*, 2019], which introduced a verification-guided method to train RNNs as POMDP policies. In contrast to the proposed method, while polices are extracted from the RNNs, these policies do not directly exhibit the memory structure of the RNNs and are instead handcrafted based on knowledge about the particular application.

There are three lines of related research. The first one concerns the formal verification of neural network-based control policies. Two prominent approaches [Huang *et al.*, 2017; Katz *et al.*, 2017] for the class of feed-forward deep neural networks rely on encoding neural networks as SMT problems through adversarial examples or ReLUs architectures respectively. [Akintunde *et al.*, 2019] concerns the direct verification of RNNs with ReLU activation functions using SMT or MILP. However, the scalability of these solver-based methods suffer from the size of the input models. We circumvent this shortcoming by our model-based approach where verification is restricted to concrete applications followed by potential improvement of the RNNs.

The second relevant direction concerns the direct synthesis of FSCs for POMDPs without neural networks. For example, [Meuleau *et al.*, 1999] uses a branch-and-bound method to compute optimal FSCs, and [Junges *et al.*, 2018] constructs an FSC using parameter synthesis for Markov chains.

Third, existing work that concerns the extraction of FSCs from neural networks [Zeng *et al.*, 1993; Weiss *et al.*, 2018; Michalenko *et al.*, 2019], does not integrate with formal verification to provide formal guarantees.

2 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 $Distr(X)$. The support of a distribution μ is $\operatorname{supp}(\mu) = \{x \in X \mid \mu(x) > 0\}$. The entropy of a distribution μ is $\mathcal{H}(\mu) := -\sum_{x\in X} \mu(x) \log_{|X|} \mu(x)$.

POMDPs. A *Markov decision process (MDP)* M is a tuple $M = (S, Act, \mathcal{P})$ with a finite (or countably infinite) set S of *states*, a finite set Act of *actions*, and a *transition probability function* $\mathcal{P}\colon S \times Act \to Distr(S)$. The reward function for states and actions is given by $r\colon S \times Act \to \mathbb{R}$. A finite *path* π of an MDP M is a sequence of states and actions; $\operatorname{last}(\pi)$ is the last state of π. 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 POMDPs, observation-action sequences are based on a finite path $\pi \in \mathsf{Paths}^{M}_{\mathit{fin}}$ of M and have the form: $O(\pi) = O(s_0) \xrightarrow{a_0} O(s_1) \xrightarrow{a_1} \cdots O(s_n)$. The set of all finite observation-action sequences for a POMDP $\mathcal{M}$ is $\mathsf{ObsSeq}^{\mathcal{M}}_{\mathit{fin}}$.

Definition 2 (POMDP Policy)**.** *An* observation-based policy *for a POMDP* $\mathcal{M}$ *is a function* $\gamma\colon \mathsf{ObsSeq}^{\mathcal{M}}_{\mathit{fin}} \to Distr(Act)$ *such that* $\operatorname{supp}\big(\gamma(O(\pi))\big) \subseteq Act\big(\operatorname{last}(\pi)\big)$ *for all* $\pi \in \mathsf{Paths}^{M}_{\mathit{fin}}$*.* $\Gamma^{\mathcal{M}}_{z}$ *is the set of observation-based policies for* $\mathcal{M}$*.*

A policy for a POMDP resolves the nondeterministic choices in the POMDP, based on the history of previous observations, by assigning distributions over actions. A *memoryless* observation-based policy $\gamma \in \Gamma^{\mathcal{M}}_{z}$ is given by $\gamma\colon Z \to Distr(Act)$, i. e., decisions are based on the current observation only. A POMDP $\mathcal{M}$ together with a policy γ yields an *induced discrete-time Markov chain (MC)* $\mathcal{M}^{\gamma}$. An MC does not contain any nondeterminism or partial observability. Our definition restricts POMDP policies to finite memory, which are typically represented as FSCs.

Definition 3 (Finite-state controller (FSC))**.** *A k-FSC for a POMDP 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 $\alpha\colon N \times Z \to Distr(Act)$ and δ is the memory update $\delta\colon N \times Z \times Act \to N$.*

An FSC has the observations Z as input and the actions Act as output. Upon an observation, depending on the current memory node the FSC is in, the action mapping α returns a distribution over Act followed by a change of memory nodes according to δ. FSCs are an extension of so-called *Moore machines*, where the action mapping is deterministic, that is, $\alpha\colon N \times Z \to Act$, and the memory update $\delta\colon N \times Z \to N$ does not depend on the choice of action.

Definition 4 (Specifications)**.** *We consider linear-time temporal logic (LTL) properties [Pnueli, 1977]. For a set of atomic propositions AP, which are either satisfied or violated by a state, and $a \in AP$, the set of LTL formulas is:*

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

Intuitively, a path π satisfies the proposition a if its first state does; $(\psi_1 \wedge \psi_2)$ is satisfied, if π satisfies both ψ_1 and ψ_2; $\neg\psi$ is true on π if ψ is not satisfied. The formula $\bigcirc\,\psi$ holds on π if the subpath starting at the second state of π satisfies ψ; π satisfies $\Box\,\psi$ if all suffixes of π satisfy ψ. Finally, π satisfies $(\psi_1 \,\mathsf{U}\, \psi_2)$ if there is a suffix of π that satisfies ψ_2 and all longer suffixes satisfy ψ_1. $\Diamond\,\psi$ abbreviates $(\mathrm{true} \,\mathsf{U}\, \psi)$.

For POMDPs, one wants to synthesize a policy such 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 $\lambda \in [0, 1]$. In addition, *undiscounted expected reward properties* $\varphi = \mathbb{E}_{\sim\lambda}(\Diamond\, a)$ require that the expected accumulated cost until reaching a state satisfying a respects $\lambda \in \mathbb{R}_{\geq 0}$.

A *specification* φ is *satisfied* for POMDP $\mathcal{M}$ and γ if it is satisfied in the MC $\mathcal{M}^{\gamma}$ ($\mathcal{M}^{\gamma} \models \varphi$). We now define a general notion of an RNN that represents a POMDP policy.

Definition 5 (Policy network)**.** *A policy network for a POMDP is a function $\hat{\gamma}\colon \mathsf{ObsSeq}^{\mathcal{M}}_{fin} \to Distr(Act)$.*

The underlying RNN which receives sequential input in the form of (finite) observation sequences from $\mathsf{ObsSeq}^{\mathcal{M}}_{fin}$, the output is a distribution over actions, see Fig. 4a. To be more precise, we identify the main components of such a network.

Definition 6 (Components of a policy network)**.** *A policy network $\hat{\gamma}$ is sufficiently described by a* hidden-state update function $\hat{\delta}\colon \mathbb{R} \times Z \times Act \to \mathbb{R}$ *and an* action mapping $\gamma_h\colon \mathbb{R} \to Distr(Act)$.

Consider the following observation sequence:

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

The policy network receives an observation and returns an action choice. Throughout the execution of the sequence, the RNN holds a continuous hidden state $h \in \mathbb{R}$, occasionally 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 transition function $\hat{\delta}$. From the prior observation sequence in (1), the corresponding hidden state sequence would be defined as:

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

Additionally, the output of the policy network is expressed by the action-distribution function γ_h, which maps the value of hidden state to a distribution over the actions. At internal memory states h_i, we have $\hat{\delta}(h_i, O(s_i), a_i) = h_{i+1}$ and $\gamma_h(h_{i+1}) = \mu(Act)$ for state s_i on path π. Note that a policy network characterizes a well-defined POMDP policy.

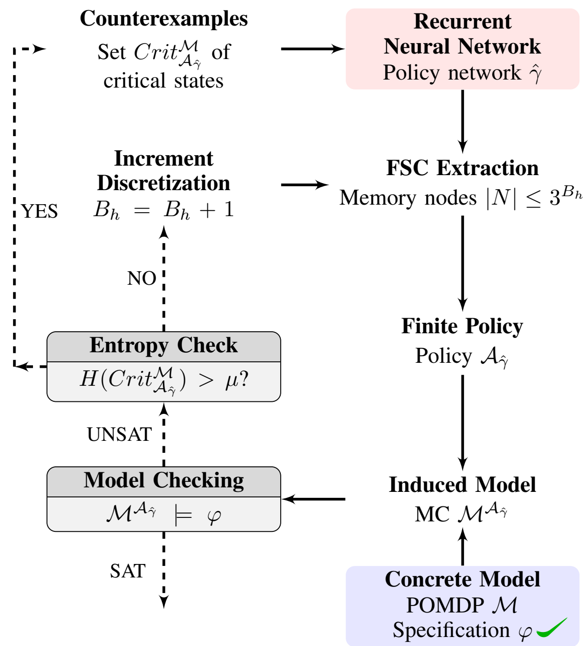


Figure 2: Procedural flow for the iterative FSC extraction and RNN-based policy improvement.

3 Problem Statement

We attempt to solve two separate but related problems: (1) For a POMDP $\mathcal{M}$, a policy network $\hat{\gamma}$ and a specification φ, the problem is to extract an FSC $\mathcal{A}_{\hat{\gamma}} \in \Gamma^{\mathcal{M}}_{z}$ such that $\mathcal{M}^{\mathcal{A}_{\hat{\gamma}}} \models \varphi$. (2) If the extraction process fails to produce a suitable candidate, then we determine an improved policy network $\hat{\gamma}^*$ for which we can solve (1).

3.1 Outline

Fig. 2 illustrates the workflow of the proposed approach for a given POMDP $\mathcal{M}$, policy network $\hat{\gamma}$ and specification φ. We summarize the individual steps below and provide the technical details in the subsequent sections.

FSC extraction. We first quantize the memory nodes of the policy network $\hat{\gamma}$, that is, we discretize the memory update of the continuous memory state h. From this discrete representation of the memory update, we construct an FSC $\mathcal{A}_{\hat{\gamma}} \in \Gamma^{\mathcal{M}}_{z}$. The procedure has as input the number B_h of neurons which defines a bound on the number of memory nodes in the FSC.

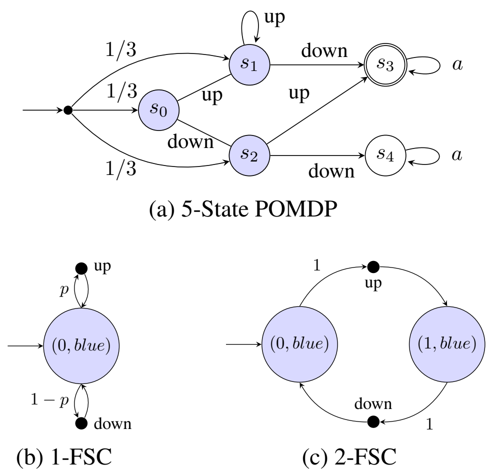


Figure 3: (a) POMDP for Example 1 with (b) 1-FSC and (c) 2-FSC. Both FSCs are defined for observing "blue" and subsequent action choices that may result in a change of memory node for the 2-FSC.

Verification. We use the FSC $\mathcal{A}_{\hat{\gamma}}$ to resolve partial information and nondeterministic choices in the POMDP $\mathcal{M}$, resulting in an induced MC $\mathcal{M}^{\mathcal{A}_{\hat{\gamma}}}$. We evaluate whether the given specification φ is satisfied for this induced MC using a formal verification technique called *model checking* [Baier and Katoen, 2008]. If the specification φ holds, then the synthesis is complete with output policy $\mathcal{A}_{\hat{\gamma}}$. However, if φ does not hold, then we decide if we shall increase the bound B_h on the number of memory nodes or if the network needs retraining. In particular, we examine whether or not the entropy over the FSC's action distribution is above a prescribed threshold.

Policy improvement. In the high entropy case, we increase the discretization level, that is, we increase B_h, and construct the FSC $\mathcal{A}_{\hat{\gamma}}$ with additional memory states at its disposal. Whereas in the other case, additional memory nodes may cause the extracted FSC to be drawn from extrapolated information and we instead seek to improve the policy network. For that, we use diagnostic information in the form of counterexamples to generate new data [Carr *et al.*, 2019].

Example 1. *We consider the POMDP in Fig. 3 as a motivating example for the necessity of memory-based FSCs. 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 $\varphi = \Pr_{\geq 0.9}(\lozenge\, s_3)$, so the agent is to reach state s_3 with at least probability 0.9. In a 1-FSC (i.e. one memory node 0), we can describe an FSC $\mathcal{A}_1$ by:*

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

A 2-FSC with two memory nodes (0 and 1), see Fig. 3c, 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:

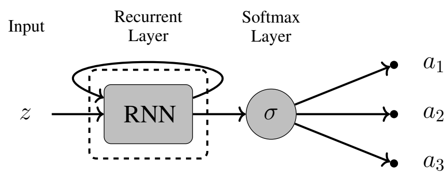
$$\alpha(0, blue) = \begin{cases} up & \textit{with probability} \quad 1, \\ down & \textit{with probability} \quad 0, \end{cases}$$
$$\alpha(1, blue) = \begin{cases} up & \textit{with probability} \quad 0, \\ down & \textit{with probability} \quad 1, \end{cases}$$
$$\delta(0, blue, up) = 1,$$
$$\delta(1,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

4 Policy Extraction

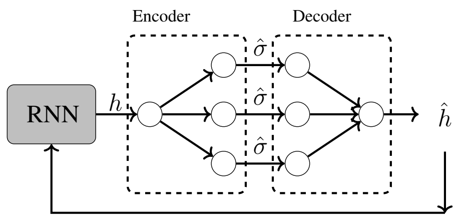
In this section we describe how we adapt the method called quantized bottleneck insertion [Koul *et al.*, 2019] to extract an FSC from a given RNN. Let us first explain the relationship between the main components of a policy network $\hat{\gamma}$ (Definition 6) and an FSC $\mathcal{A}$ (Definition 3). In particular, the hidden-state update function $\hat{\delta}\colon \mathbb{R} \times Z \times Act \to \mathbb{R}$ takes as input a real-valued hidden state of the policy network, while the memory update function of an FSC takes a memory node from the finite set N. 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.

Policy network modification. To obtain the above linkage, we leverage an autoencoder [Goodfellow *et al.*, 2016] in the form of a *quantized bottleneck network* (quantized bottleneck network (QBN)) [Koul *et al.*, 2019]. This QBN, consisting of an encoder and a decoder, is inserted into the policy network directly before the softmax layer, see Fig. 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 Fig. 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] has another QBN for a continuous observation space, however, we focus on discrete observations and 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 concrete application, e.g. using a POMDP model. We form a dataset of consecutive pairs $(\hat{h}_t, \hat{h}_{t+1})$ of discrete 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 policy network. 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 $N \times (|Z| \times |Act|)$ transaction table, for a detailed description see [Koul *et al.*, 2019]. We can additionally construct the action mapping $\alpha\colon N \times Z \to Distr(Act)$ with $\alpha(n_t, z_t) = \mu \in Distr(Act)$ by querying the softmax-output layer (see Fig. 4a) for each memory state and observation.



(a) RNN Policy.



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

Figure 4: Policy network structure without and with a QBN.

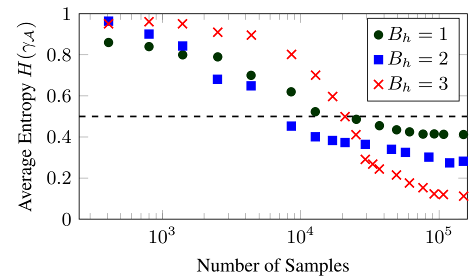


Figure 5: Entropy of the extracted FSCs from an RNN as it is trained with more samples. For each sequence we fix the discretization and add more samples guided by the counterexamples.

5 Policy Evaluation and Improvement

Evaluation using formal verification. We assume that for POMDP $\mathcal{M} = (M, Z, O)$ and specification φ, we have an extracted FSC $\mathcal{A}_{\hat{\gamma}} \in \Gamma_z^{\mathcal{M}}$ as in Definition 3. We use the policy $\mathcal{A}_{\hat{\gamma}}$ to obtain the induced MC $\mathcal{M}^{\mathcal{A}_{\hat{\gamma}}}$. For this MC, formal verification through model checking checks whether $\mathcal{M}^{\mathcal{A}_{\hat{\gamma}}} \models \varphi$ and thereby provides hard guarantees about the quality of the extracted FSC $\mathcal{A}_{\hat{\gamma}}$ 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 and Katoen, 2008].

Example 1 (cont.)**.** *Consider the case in the 1-FSC $\mathcal{A}_1$ (Fig. 3b) where $p = 1$, the probability of reaching the state s_3 in the induced MC 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 $\mathcal{A}_{\hat{\gamma}}$: $(0, s_0)$ 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 upper bound B_h on the number of memory nodes to extract a new FSC. At each iteration of the inner loop in Fig. 2, we modify the QBN for the new, increased, level of discretization and obtain a new FSC using the process outlined in Sect 4. On the other hand, we may decide via a formal entropy check whether new data need to be generated to actually improve the policy.

Improving the policy network. Our goal is to determine whether a policy network requires more training data or not. Existing approaches in supervised learning methods leverage benchmark comparisons between a train-test set using a loss function [Baum and 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 we seek. 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 $\mathcal{A}_{\hat{\gamma}}$. 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 $\mathit{Crit}_{\mathcal{A}_{\hat{\gamma}}}^{\mathcal{M}} \subseteq N \times S$ that serves us as a counterexample. These pairs carry the joint information of critical states and memory nodes from the policy applied to the MC and may be formalized using a so-called product construction.

Entropy measure. The average entropy across the distributions over actions at the choices induced by the counterexample set $\mathit{Crit}_{\mathcal{A}_{\hat{\gamma}}}^{\mathcal{M}}$ is our measure of choice to determine the level of training for the policy network. Specifically, for each pair $(n, s) \in \mathit{Crit}_{\mathcal{A}_{\hat{\gamma}}}^{\mathcal{M}}$, we collect the distribution $\mu \in \mathit{Distr}(\mathit{Act})$ over actions that $\mathcal{A}_{\hat{\gamma}}$ 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:

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

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

In Fig. 5, we observe that when there are fewer samples and higher discretization, the extracted FSC tends to perform arbitrarily. We lift the function H to the full set $\mathit{Crit}_{\mathcal{A}_{\hat{\gamma}}}^{\mathcal{M}}$:

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

We compare the average entropy over all components of the counterexample against a threshold $\eta \in [0, 1]$, that is, if $H(\mathit{Crit}_{\mathcal{A}_{\hat{\gamma}}}^{\mathcal{M}}) > \eta$, we will provide more training data. Vice versa, if $H(\mathit{Crit}_{\mathcal{A}_{\hat{\gamma}}}^{\mathcal{M}}) \geq \eta$, we will increase the upper bound on the number of memory states in the FSC.

Example 1 (cont.)**.** *Under the working example, the policy $\mathcal{A}_1$ was the 1-FSC with $p = 1$ (Fig. 3b), which*

				Extraction Approach			Handcrafted			PRISM-POMDP		SolvePOMDP	
Problem	$\lvert S\rvert$	$\lvert Z\rvert$	Type	Memory	Value	Time (s)	Memory	Value	Time (s)	Value	Time (s)	Value	Time (s)
Maze(1)	11	7	Min	2	4.33	80.31	2	4.31	30.70	**4.30**	**0.09**	4.30	0.30
Maze(2)	14	7	Min	3	5.34	114.23	3	5.31	46.65	5.23	2.176	**5.23**	**0.67**
Maze(5)	23	7	Min	3	13.29	160.12	6	14.40	68.09	13.00*	4110.50	**12.04**	**134.46**
Maze(10)	38	7	Min	5	**23.02**	210.01	11	100.21	**158.33**	MO	MO	MO	MO
Grid(3)	9	2	Min	3	2.90	87.31	2	2.90	38.94	2.88	2.332	**2.88**	**0.06**
Grid(4)	16	2	Min	7	4.20	124.31	3	4.32	79.99	4.13	1032.53	**4.13**	**0.73**
Grid(5)	25	2	Min	9	5.91	250.14	4	6.623	91.42	MO	MO	**5.42**	**1.97**
Grid(10)	100	2	Min	9	**12.92**	1031.21	9	13.63	**268.40**	MO	MO	MO	MO
Grid(25)	625	2	Min	16	**35.32**	6514.30	24	531.05	**622.31**	MO	MO	MO	MO
Navigation (4)	256	256	Max	8	0.92	160.32	8	0.92	80.26	**0.93***	**1034.64**	NA	NA
Navigation (5)	625	256	Max	8	**0.95**	311.65	8	0.92	**253.11**	MO	MO	NA	NA
Navigation (10)	10^4	256	Max	8	**0.90**	2561.02	4	0.85	**1471.17**	MO	MO	NA	NA
Navigation (20)	1.6×10^5	256	Max	9	**0.98**	8173.03	4	0.96	**7068.24**	MO	MO	NA	NA

Table 1: Synthesizing strategies for examples with expected reward and LTL specifications.

Component	Time (s)	%
Total	311.65	100
Training/Retraining RNN	205.77	66.0
Extracting/Applying FSC	80.21	25.7
Verification of MC	2.23	0.7
Data from countexamples/entropy check	7.05	2.2

Table 2: Breakdown of process execution time by component for the Navigation (5) benchmark.

produces two counterexample memory and state pairs: $Crit^{\mathcal{M}}_{\mathcal{A}_1} = \{(0, s_0), (0, s_1)\}$. *The procedure would then examine the policy's average entropy at these critical components* $(n, s) \in Crit^{\mathcal{M}}_{\mathcal{A}_1}$*, which in this trivial example is given by* $H(Crit^{\mathcal{M}}_{\mathcal{A}_1}) = -p \log_2(p) - (1-p) \log_2(1-p) = 0$ *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 Fig. 3c.*

Collecting new training data. There are several ways to perform retraining of the policy network. One option is sampling using the Q_{MDP} value method [Littman *et al.*, 1995]. Alternatively, one can generate data through bootstrapped random sampling as in [Hausknecht and Stone, 2015]. We follow the approach from [Carr *et al.*, 2019] and start with a policy for the underlying MDP M that satisfies φ. This policy is gradually improved using a counterexample-guided technique employing the set $Crit^{\mathcal{M}}_{\mathcal{A}_{\hat{\gamma}}}$.

6 Numerical Examples

We evaluate the proposed verification and synthesis approach by via a series of benchmark examples that are subject to either LTL or expected cost specifications. For both sets we compare to two synthesis tools: PRISM-POMDP [Norman *et al.*, 2017] and SolvePOMDP [Walraven and Spaan, 2017] from the respective formal methods and planning communities. We further compare to another RNN-based synthesis procedure with a handcrafted memory structure for FSCs [Carr *et al.*, 2019]. For proper comparison with the synthesis tools we adopt the experiment setup of [Carr *et al.*, 2019], whereby we always iterate for 10 instances of retraining the RNN from counterexample data. Similarly, the proposed approach is not guaranteed to reach the optimum, but shall rather improve as far as possible within 10 iterations.

Implementation and Setup. We provide a Python toolchain that employs the probabilistic model checker Storm [Dehnert *et al.*, 2017]. We train and encode the RNN policy networks $\hat{\gamma}$ using the deep learning library Keras. All experiments are run on a 2.3 GHz machine with a 12 GB memory limit and a max computation time of 10^5 seconds.

Settings. We analyze the method on three different settings: Maze(c), Grid(c) and Navigation(c), for detailed descriptions of these examples see [Norman *et al.*, 2017] and [Carr *et al.*, 2019], respectively. In each of these settings the policies have an action space of the cardinal directions navigating through a grid-based environment. For the former two examples, we attempt to synthesize a policy that minimizes an *expected cost* subject to reaching a goal state a: $\mathbb{E}^{\mathcal{M}}_{\min}(\lozenge\, a)$. In the latter example, we seek a policy that maximizes the probability of satisfying an *LTL specification*, in particular avoiding obstacles X, both static and randomly moving, until reaching a target area a: $\mathbb{P}^{\mathcal{M}}_{\max}(\neg X \cup a)$.

Discussion. The results are shown in Table 1. The proposed extraction approach 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 extraction method produces higher-quality policies - within 2% of the optimum. That effect is due to our automatic extraction of suitable FSCs. 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 synthesizes higher values. Yet, with larger environments the fixed memory structure produces poor policies as more memory states are beneficial to account for the past behavior.

7 Conclusion

We introduced a novel synthesis procedure for extracting and verifying RNN-based policies. In comparison to other approaches to verify RNNs, we take a model-based approach and provide guarantees for concrete applications modeled by POMDPs. Based on the verification results, we propose a way to either improve the extracted policy or the RNN itself. Our results demonstrate the effectiveness of the approach and that we are competitive to state-of-the-art synthesis tools.

Acknowledgements

This work was supported by DARPA D19AP00004, ONR N00014-18-1-2829, and ARL ACC-APG-RTP W911NF.

References

[Akintunde *et al.*, 2019] Michael E. Akintunde, Andreea Kevorchian, Alessio Lomuscio, and Edoardo Pirovano. Verification of RNN-based neural agent-environment systems. In *AAAI*, pages 6006–6013. AAAI Press, 2019.

[Baier and Katoen, 2008] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[Bakker, 2001] Bram Bakker. Reinforcement learning with long short-term memory. In *NIPS*, pages 1475–1482. MIT Press, 2001.

[Baum and Wilczek, 1987] Eric B. Baum and Frank Wilczek. Supervised learning of probability distributions by neural networks. In *NIPS*, pages 52–61. American Institue of Physics, 1987.

[Carr *et al.*, 2019] Steven Carr, Nils Jansen, Ralf Wimmer, Alexandru Constantin Serban, Bernd Becker, and Ufuk Topcu. Counterexample-guided strategy improvement for POMDPs using recurrent neural networks. In *IJCAI*, pages 5532–5539, 2019.

[Cassandra, 1998] Anthony R Cassandra. A survey of POMDP applications. In *AAAI*, volume 1724, 1998.

[Cover and Thomas, 2012] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[Dehnert *et al.*, 2017] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *CAV (2)*, volume 10427 of *LNCS*, pages 592–600. Springer, 2017.

[Goodfellow and Vinyals, 2015] Ian J. Goodfellow and Oriol Vinyals. Qualitatively characterizing neural network optimization problems. In *ICLR*, 2015.

[Goodfellow *et al.*, 2016] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

[Hausknecht and Stone, 2015] Matthew J. Hausknecht and Peter Stone. Deep recurrent Q-learning for partially observable MDPs. In *AAAI*, pages 29–37. AAAI Press, 2015.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[Huang *et al.*, 2017] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *CAV (1)*, volume 10426 of *LNCS*, pages 3–29. Springer, 2017.

[Junges *et al.*, 2018] Sebastian Junges, Nils Jansen, Ralf Wimmer, Tim Quatmann, Leonore Winterer, Joost-Pieter Katoen, and Bernd Becker. Finite-state controllers of POMDPs via parameter synthesis. In *UAI*, 2018.

[Katz *et al.*, 2017] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV*, pages 97–117. Springer, 2017.

[Koul *et al.*, 2019] Anurag Koul, Alan Fern, and Sam Greydanus. Learning finite state representations of recurrent policy networks. In *ICLR*, 2019.

[Littman *et al.*, 1995] Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In *ICML*, pages 362–370. Morgan Kaufmann, 1995.

[Meuleau *et al.*, 1999] Nicolas Meuleau, Kee-Eung Kim, Leslie Pack Kaelbling, and Anthony R. Cassandra. Solving POMDPs by searching the space of finite policies. In *UAI*, pages 417–426. Morgan Kaufmann, 1999.

[Michalenko *et al.*, 2019] Joshua J. Michalenko, Ameesh Shah, Abhinav Verma, Richard G. Baraniuk, Swarat Chaudhuri, and Ankit B. Patel. Representing formal languages: A comparison between finite automata and recurrent neural networks. In *ICLR*, 2019.

[Mulder *et al.*, 2015] Wim De Mulder, Steven Bethard, and Marie-Francine Moens. A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language*, 30(1):61–98, 2015.

[Norman *et al.*, 2017] Gethin Norman, David Parker, and Xueyi Zou. Verification and control of partially observable probabilistic systems. *Real-Time Systems*, 53(3):354–402, 2017.

[Pascanu *et al.*, 2014] Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. In *ICLR*, 2014.

[Pnueli, 1977] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[Poupart and Boutilier, 2003] Pascal Poupart and Craig Boutilier. Bounded finite state controllers. In *NIPS*, pages 823–830. MIT Press, 2003.

[Tsoi and Back, 1997] Ah Chung Tsoi and Andrew D. Back. Discrete time recurrent neural network architectures: A unifying review. *Neurocomputing*, 15:183–223, 1997.

[Walraven and Spaan, 2017] Erwin Walraven and Matthijs T. J. Spaan. Accelerated vector pruning for optimal pomdp solvers. In *AAAI*, pages 3672–3678, 2017.

[Wang *et al.*, 2018] Qinglong Wang, Kaixuan Zhang, Xue Liu, and C. Lee Giles. Verification of recurrent neural networks through rule extraction. *CoRR*, abs/1811.06029, 2018.

[Weiss *et al.*, 2018] Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 5244–5253. PMLR, 2018.

[Wierstra *et al.*, 2007] Daan Wierstra, Alexander Förster, Jan Peters, and Jürgen Schmidhuber. Solving deep memory POMDPs with recurrent policy gradients. In *ICANN*, pages 697–706. Springer, 2007.

[Zeng *et al.*, 1993] Zheng Zeng, Rodney M. Goodman, and Padhraic Smyth. Learning finite machines with self-clustering recurrent networks. *Neural Comput.*, 5(6):976–990, November 1993.