Deep Bayesian Nonparametric Learning of Rules and Plans from Demonstrations with a Learned Automaton Prior

Brandon Araki,1 Kiran Vodrahalli,2 Thomas Leech,1, 3 Cristian-Ioan Vasile,1 Mark Donahue,3 and Daniela Rus1
1CSAIL, Massachusetts Institute of Technology
2 Columbia University
3 MIT Lincoln Laboratory
araki@mit.edu, knv2109@columbia.edu, tleech@mit.edu, cvasile@mit.edu, mark.donahue@ll.mit.edu, rus@csail.mit.edu

Abstract

We introduce a method to learn imitative policies from expert demonstrations that are interpretable and manipulable. We achieve interpretability by modeling the interactions between high-level actions as an automaton with connections to formal logic. We achieve manipulability by integrating this automaton into planning, so that changes to the automaton have predictable effects on the learned behavior. These qualities allow a human user to first understand what the model has learned, and then either correct the learned behavior or zero-shot generalize to new, similar tasks. We build upon previous work by no longer requiring additional supervised information which is hard to collect in practice. We achieve this by using a deep Bayesian nonparametric hierarchical model. We test our model on several domains and also show results for a real-world implementation on a mobile robotic arm platform.

1 Introduction

Imitation learning (IL) is a method for producing desired behaviors in agents with expert demonstrations [Abbeel and Ng (2004), Daume III, Langford, and Marcu (2009), Ross, Gordon, and Bagnell (2011)]. IL is a successful approach in many tasks including camera control, speech imitation, and self-driving for cars [Taylor et al. (2017), Codecilla et al. (2018), Ho and Ermon (2016)]. Despite these successes, current approaches to IL fall short of human learning, particularly in the domain of multi-step tasks. When a human learns a complicated task, such as driving or preparing a meal, they can explain, at a high level, the rules they followed or the steps they took to perform the task. Furthermore, if the rules change or if they want to, say, cook a slightly different dish, they can easily alter their behavior to adapt to the new circumstances. Most current approaches to IL cannot achieve this level of flexibility because their learned policies are black boxes.

Our prior work, Araki et al. (2019b), introduces the notions of interpretability and manipulability. A policy is called interpretable if the relations between its high-level actions are defined by a finite state automaton (FSA), and it is manipulable if, by changing the transitions in the FSA, a human user can produce predictable changes in the learned behavior. These properties were achieved by designing a hierarchical model with high-level actions represented by an FSA and incorporating the FSA into a differentiable planning algorithm. However, in addition to expert trajectories, this model requires the trajectories to be labeled with the current automaton state at every time step, which is almost equivalent to requiring the user to know the automaton ahead of time.

In this paper, we remove the necessity of FSA state labels by interpreting the learning problem as solving a partially observable Markov decision process (POMDP) where the hidden states and transitions correspond to an unknown FSA. We model the FSA as part of a hidden Markov model (HMM), which represents the composition of a planning policy with the POMDP. Learning the HMM is challenging because the FSA state labels, transitions, and the number of FSA states are all unknown. We fit the HMM with stochastic variational inference (SVI) on expert demonstration data, simultaneously learning the unknown FSA and planning over the resulting MDP.

We also use spectral techniques applied to a matrix representation of an automaton to create a prior for the FSA. We report considerable improvements over baselines in several domains, including two robotic manipulation tasks with real-world experiments. Because our model satisfies the interpretability and manipulability properties from Araki et al. (2019b), we can zero-shot generalize to new tasks and fix mistakes in incorrect models without additional demonstrations.

1.1 Contributions

1. We solve logic-structured POMDPs by learning a planning policy from task demonstrations alone, extending ideas from the differentiable planning literature.
2. We use spectral techniques applied to a matrix representation of a finite automaton to design a novel prior for nonparametric hierarchical Bayesian models.
3. Our model learns the FSA transition matrix, allowing us to interpret the rules that the model has learned.
4. We explain how to modify the learned transition matrix to manipulate the behavior of the agent on a real-world robotic platform in the contexts of packing a lunchbox and opening a locked cabinet.
2 Related Work

Our model extends Araki et al. (2019b), which builds upon the Value Iteration Network (VIN) model (Tamar et al. 2016) by applying a more structured variant of VIN to the product of a low-level MDP with a logical specification defined by an FSA. Other works incorporating logical structure into the imitation learning setting include Paxton et al. (2017), Li, Ma, and Belta (2017), Hasanbeig, Abate, and Kroening (2018), Icarte et al. (2018), Burke, Penkov, and Ramamooorthy (2019), and Gordon, Fox, and Farhadi (2019). These models assume that at least part of the logical specification is known, and they are not interpretable and manipulable. By contrast, our model learns the FSA end-to-end in an unsupervised manner from demonstrations.

Shah et al. (2018) gives methods for learning a posterior over logic specifications from demonstrations, and Shah, Li, and Shah (2019) defines objectives for planning over a distribution of planning problems defined by logic specifications which reduces to the problem of solving MDPs. Our work directly considers the problem as a POMDP and introduces data-driven prior assumptions on the distribution of an FSA to mitigate the hardness of the problem. In addition, we never have to enumerate an exponentially-sized MDP. Furthermore, since our model is learned end-to-end, we recover an FSA specification that is tuned for the imitation policy that we fit, enhancing manipulability and interpretability simultaneously. Karkus, Hsu, and Lee (2017) also extends VIN to partially observable settings. This is achieved by learning the automaton transition function \( TM \) and the observation map \( O \).

With this notation, we formulate the POMDP learning problem as follows. Given \( N \) data points, data point \( i \) has \( T_i \) time steps. Dataset \( D = \{d_0, \ldots, d_N\} \), where \( d_i = (s_0^i, a_0^i), \ldots, (s_{T_i}^i, a_{T_i}^i) \). Expanding the POMDP tuple gives \( (S \times P \times F, A, T \times M \times TM, R, S \times P, O, \gamma_d) \).

Unknown elements have been underlined to emphasize the learning objective. We assume that the actions \( A \), the low-level transitions \( T \), the proposition map \( M \), the observation probabilities \( O \), and the discount factor \( \gamma_d \) are known. We also assume that \( O \) is deterministic – in other words, that the agent has sensors that can perfectly sense its state in the environment. The goal of solving a POMDP is to find a policy that maximizes reward – in this case, a policy that mimics the expert demonstrations. This is achieved by learning the automaton transition function \( TM \), the reward function \( R \), and the number of states of the automaton \( F \).

4 Method

To learn a policy for the POMDP, we formulate the POMDP composed with the policy as a hierarchical Hidden Markov Model (HMM) with recurrent transitions and autoregressive emissions, where the hidden states and transitions of the POMDP are latent variable parameters (shown in Fig. 2). The observed logic state at every time step is interpreted as a hidden state of the FSA, and \( TM, R, \) and \( F \) are interpreted as high-level latent variables.

A sketch of the learning algorithm is shown in Alg. 1, and Fig. 1 shows example inputs and outputs. The data consist of a set of expert trajectories over a domain. The domain shown is a \( 3 \times 3 \) gridworld where the agent must first go to \( a \), then to \( b \), while avoiding obstacles \( o, e \) is the “empty”
The data consist of a set of expert trajectories over a domain. Each trajectory has an associated proposition map that relates every position in the domain to a proposition. Spectral learning generates an automaton prior for the Bayesian model, which learns variational parameters that are equivalent to the reward function and transition matrix of the environment.

**Algorithm 1 Maximum A Posteriori Estimation on Bayesian LVIN model**

1: procedure MAP-ESTIMATION  
2: Training Inputs: \{\{(s_t, a_t)\}^{T_t}_{t=0}\}^{N-1}_{i=0}  
3: To learn:  
4: Number of automaton states prior \(\hat{\alpha} \in \mathbb{R}_{>0}^{2P-3}\)  
5: Transition matrix prior \(\hat{\beta}^F \in \mathbb{R}_{>0}^{F \times P \times F}\) for \(F = 4, \ldots, 2P\)  
6: Reward function prior \(\hat{\gamma}^F \in \mathbb{R}^{F} \times \mathbb{R}_{>0}^{F}\) for \(F = 4, \ldots, 2P\)  
7: Generate spectral learning prior (Sec. 4.3)  
8: Stochastic variational inference on the learning objective \(\mathcal{L}\) (Sec. 4.1)  
9: return \(\hat{\alpha}^*, \hat{\beta}^*, \hat{\gamma}^* = \arg \min (\alpha, \beta, \gamma) \mathcal{L}\)  
10: end procedure

The algorithm consists of approximating the posterior of the Bayesian model and returning the modes of the latent variables (See 4.1): LVIN, a differentiable variation of value iteration that incorporates both the low- and high-level transitions, plays an important role (Sec. 4.2). A common issue with learning complex Bayesian models is that they are very sensitive to their initialization. The discrete high-level transition function \(TM\) is particularly vulnerable to converging to local minima, so we use spectral learning to obtain a good prior (Sec. 4.3).

Posterior inference finds likely values for the number of automaton states \(F\), the reward function \(R\), and the transition matrix \(TM\). In the figure, the TM is represented as a collection of matrices - each matrix is associated with the current logic state; columns correspond to propositions and rows correspond to the next logic state. The entry in each grid is the probability \(TM(s'|s, p)\). Black indicates 1 and white indicates 0. Therefore in the initial state \(S0\), \(a\) causes a transition to \(S1\), whereas \(o\) causes a transition to the trap state \(T\). The outputs of the algorithm are valuable for two reasons: 1) \(TM\) is relatively easy to interpret, giving insight into the rules that the expert is following; and 2) \(R\) and \(TM\) can be used for planning. Furthermore, modifications to \(TM\) result in predictable changes in the agent’s behavior.

### 4.1 Bayesian Model

We now define the Bayesian model for the policy class (all variables in this section are listed in Araki et al. (2019a)). The nonparametric model is stated in Araki et al. (2019a); in practice, we approximate the nonparametric model by assuming the possible number of FSA states is finite. We will discuss this approximate model for the rest of the paper.

One of the main challenges in this learning problem is to infer the number of states \(F\) of the transition matrix – this problem is nonparametric because \(F\) is theoretically unbounded. We approximate the nonparametric nature of the problem by assuming that \(F\) is upper-bounded by \(2P\). This upper bound implies that each proposition is not responsible for more than 2 transitions to distinct FSA states, which we believe is a reasonable assumption for normal domains. \(TM\) and \(R\) both rely on \(F\) to determine their dimensionality. Hidden-state transitions are a function of \(TM\) and the previous observation, among other things (see Fig. 2). Transitions that depend on variables besides the previous hidden state are called recurrent. Transitions between low-level states \(s_{t-1}\) and \(s_t\) are determined by \(T(s_t|a_t, s_{t-1})\). Actions are chosen by a policy found using value iteration. Our value iteration module incorporates \(TM\) and is called LVIN (Sec. 4.2). Both \(T\) and LVIN depend on variables besides the current hidden state and are therefore called autoregressive.

The full generative model is described below:

\[
\begin{align*}
\alpha & \in \mathbb{R}_{>0}^{2P-3}, \\
\beta & \in (\beta^4, \ldots, \beta^{2P}), \\
\gamma & \in (\gamma^4, \ldots, \gamma^{2P}) \\
\theta & \sim \text{Dirichlet}(\alpha) \\
F & \sim \text{Categorical}(\theta) \\
\beta^F & \in \mathbb{R}_{>0}^{F \times P \times F}, \\
\gamma^F & \in \mathbb{R}^{F} \times \mathbb{R}_{>0}^{F} \\
TM^F | \beta^F & \sim \text{Dirichlet}(\beta^F) \\
R^F | \gamma^F & \sim \text{Normal}(\gamma^F) \\
\pi & := \text{LVIN}(TM^F, R^F) \\
\rho_t | s_{t-1}, f_{t-1} & \sim \pi(s_{t-1}, f_{t-1}) \\
s_t & := T(a_t, s_{t-1}), p_t := M(s_t) \\
f_t | p_t, f_{t-1}, TM^F & \sim \text{Categorical}(TM^F(f_{t-1}, p_t))
\end{align*}
\]
\( \alpha \) is the prior over distributions \( \theta \) of potential numbers of FSA states \( F \). \( \beta \) is a collection of \( 2P - 3 \) priors \( \beta^F \), where \( \beta^F \) is the prior of the transition matrix \( TM^F \) of dimensionality \( F \times P \times F \). Similarly, \( \gamma \) is a collection of \( 2P - 3 \) priors \( \gamma^F \), where \( \gamma^F \) represents the mean and variance priors for the reward function \( R^F \). \( \pi \) is the policy found using LVIN; action \( a_t \) is drawn from the policy. State \( s_t \) and proposition \( p_t \) are given by the deterministic functions \( T \) and \( M \). Lastly, the current automaton state \( f_t \) is drawn from \( TM^F \).

We incorporate known features of the environment into the model as priors. Many of these priors rely on the assumption that every automaton we consider has one initial state, one goal state, and one trap state. Our assumptions about the rules of the environment are built into each \( \beta^F \), which are the Dirichlet priors for \( TM^F \). Each \( \beta^F \) is populated with the value 0.5 before adding other values, since for Dirichlet priors, values below 1 encourage peaked/discrete distributions. Therefore, this prior biases the TM towards deterministic automata. We add a prior to the trap state that favors self-transitions because we know that the trap state is a “dead-end” state. We add an obstacle prior to bias the model in favor of automata where obstacles lead to the trap state. We add a goal state prior so that the model favors self-transitions for the goal state. We add an empty state prior to make sure the model favors self-transitions for all states given the empty proposition. We use spectral learning to give a prior for the other transitions as well as for \( \alpha \) (Sec. 4.3). We also give priors to the reward function so that the goal state has a positive reward and the trap state has a negative reward.

The joint distribution over the latent variables is shown below, with possible numbers of FSA states \( F \) and possible next states \( f \) marginalized out. A bar over a variable indicates that it is a list over possible values of \( F \). \( i \) represents the data index; \( t \) the time index; and \( F \) the number of-automaton-states index.

\[
p(D, R, TM, \theta|\alpha, \beta, \gamma) = \prod_{i=0}^{N-1} \prod_{t=2}^{T} \sum_{F=1}^{2P-3} \sum_{f_{i-1}^F=0}^{F-1} p(s_{i+1}^F|s_{i+1}^F, f_{i-1}^F, TM^F, R^F) p(f_{i-1}^F|f_{i-2}^F, TM^F, s_{i-1}^F) p(TM^F|F, \beta^F) p(R^F|F, \gamma^F) p(F|\theta)p(\theta|\alpha)
\]

The posterior can be derived from the joint distribution. Our variational approximation to the posterior is

\[
q(R, TM, \theta|D, \hat{\alpha}, \hat{\beta}, \hat{\gamma}) = \sum_{\alpha^*} \sum_{\beta^*} \sum_{\gamma^*} q(TM^F|F, \beta^F) q(R^F|F, \gamma^F)
\]

The variational approximation uses amortization (Ritchie, Horsfall, and Goodman 2016) to avoid having a huge number of variational parameters – without amortization, every next FSA state \( f_{t-1}^F \) would have to be drawn from an independent Dirichlet distribution with its own set of variational parameters. We avoid this situation by drawing \( f_{t-1}^F \) from \( TM^F(f_{t-2}^F, s_{i-1}^F) \), so that parameters are shared for a given FSA state-proposition pair. Also note that the variational approximation is the same as the joint likelihood, except that the first two data-dependent distributions are removed. Other works such as Damianou and Lawrence (2013) have also used this pattern for constructing variational approximations. The proposal distributions are the same as the corresponding distributions in the joint likelihood function. An illustration of the graphical model of the variational approximation is in Araki et al. (2019a).

The objective of the variational inference problem is to minimize the KL divergence between the true posterior and the variational distribution:

\[
\mathcal{L} = KL(q(R, TM, \theta|D, \hat{\alpha}, \hat{\beta}, \hat{\gamma})||p(R, TM, \theta|D, \alpha, \beta, \gamma))
\]

\[
\hat{\alpha}^*, \hat{\beta}^*, \hat{\gamma}^* = \arg\min_{(\alpha, \beta, \gamma)} \mathcal{L}
\]

\( \hat{\alpha}^*, \hat{\beta}^*, \) and \( \hat{\gamma}^* \) serve as approximations of \( \alpha, \beta, \) and \( \gamma \), and therefore define distributions over \( F, TM, \) and \( R \). Letting \( F^* = \arg\max_F \hat{\alpha}^* \), we get priors for \( TM^F^* \) (\( \hat{\beta}^F^* \)) and \( R^F^* \) (\( \hat{\gamma}^F^* \)) which can be used for planning.

The variational problem was implemented using Pyro and Pytorch. Pyro uses stochastic variational inference to approximate the variational parameters.
4.2 Logical Value Iteration Networks (LVIN)

At every time step, the model must choose an optimal action by calculating a policy. In this work, the policy is found using value iteration on the learned MDP. We use the “Logical Value Iteration Network” (Araki et al. 2019b), which integrates the high-level transitions $TM$ into value iteration. The modified value iteration equations are shown below. In the first step, the Q-function $Q$ is updated using reward function $R$, low-level transitions $T$ and value function $V$. Next, the value function is updated. LVIN adds a third step, where the values are propagated between logic states using $TM$. Note that propositions $P$ are not an input to the Q and value functions because each low-level state $s$ is deterministically associated with a single proposition $p$, so $p$ is a redundant input for a given $s$.

\[
\begin{align*}
Q^{t+1}(s, f, a) &\leftarrow R(s, f, a) + \gamma_d \sum_{s' \in S} T(s'|s, a)V^t(s', f) \\
\hat{V}^{t+1}(s, f) &\leftarrow \max_a Q^{t+1}(s, f, a) \\
V^{t+1}(s, f) &\leftarrow \sum_{f' \in F} TM(f'|f, M(s))\hat{V}^t(s, f')
\end{align*}
\]

4.3 Spectral Learning for Weighted Automata

One of the main issues of using variational inference on complex Bayesian models is its tendency to converge to undesirable local minima. To avoid this, we use the output of spectral learning for weighted automata as a prior for $TM$.

Spectral learning uses tensor decomposition to efficiently learn latent variables. We can use this technique by representing an automaton as a Hankel matrix (Arrivault et al. 2017). The Hankel matrix is a bi-infinite matrix with rows that correspond to prefixes and columns to suffixes of all possible input strings. The value of a cell is the probability of the corresponding string. The input string corresponds to the propositions that are true at each time step. A string from the environment in Fig. 1 could be $eaeeb$, indicating that the agent traversed an empty space before reaching $a$, and then traversed another three empty spaces before reaching $b$. The rank of the Hankel matrix is equal to the automaton’s number of states. An automaton with $m$ states can be reconstructed from a rank $m$ factorization of the Hankel matrix.

The problem is: given a rank $m$, find a rank factorization $H = WP$, $H$ is the Hankel matrix, $H \in \mathbb{R}^{n \times d}$, $W \in \mathbb{R}^{n \times m}$, $P \in \mathbb{R}^{m \times d}$. Let $h_{e,S} = H[0,:]$ and $h_{P,e} = H[:,0]$.

\[
H_{\sigma} = p(\mathbf{u} \sigma \mathbf{v}) \quad \text{in other words, } H_{\sigma} \text{ is a submatrix of } H \text{ where all prefixes end with } \sigma. \text{ Let } \Sigma \text{ be an alphabet with elements } \sigma (\Sigma \text{ corresponds to the propositions). } H \text{ can be divided into } |\Sigma| \text{ submatrices } H_{\sigma} \text{ and a submatrix } H_{\epsilon} \text{ (where } \epsilon \text{ corresponds to an empty string).}

When $W$ and $P$ are obtained, we can derive the Weighted Automaton (WA) $W = \langle m, I, F, \{M_{\sigma}\}_{\sigma \in \Sigma} \rangle$ corresponding to the Hankel matrix. $m$ is the number of states, equal to the rank of the Hankel matrix. $I$ is the $m \times 1$ vector of initial state probabilities, and $F$ is the $m \times 1$ vector of final state probabilities. $M_{\sigma}$ are the $m \times m$ transition weights from the current state to the next state for each proposition $\sigma$. $W$ can be derived using the following equations. ($^+$ stands for the Moore-Penrose pseudoinverse).

\[
I^T = h_{e,S}^TP^+, \quad F = W^+h_{P,e}, \quad M_{\sigma} = W^+H_{\sigma}P^+
\]

Let $1_I = [1, 0, \ldots, 0]$ and $1_F = [0, \ldots, 0, 1]$. $W$ and $P$ are obtained from the optimization problem:

\[
\begin{align*}
\minimize_{W,P} & \quad \frac{1}{2}||H - WP||_F^2 + \alpha_s \sum_{\sigma \in \Sigma} ||W^+H_{\sigma}P^+||_1 \\
\text{subject to} & \quad W \succeq 0, P \succeq 0
\end{align*}
\]

$\alpha_s$, $\beta_s$, and $\gamma_s$ are hyperparameter weights. The first term in the objective function corresponds to the matrix factorization; the second term is an $L_1$ regularization term on the transition weight matrices, and the third and fourth terms are constraints for the first and last states to be the initial and final states, respectively. The positivity constraints on $W$ and $P$ constrain the weights to be positive.

We use the open-source Sp2Learn toolbox (Arrivault et al. 2017) to process the data and generate the Hankel matrices. We use Tensorflow to perform the optimization problem above to factor the matrix. We can then obtain $I$, $F$, and $M_{\sigma}$.

The transition weights of the learned WA are not constrained to add to one, so they do not correspond to probabilities. The WA will also not include propositions that are not present in the data strings (such as the obstacle proposition). Therefore the learned WA is better suited as a prior rather than as the primary means of determining $TM$.

We learn automata with number of states ranging from 4 to 2$P$. We have observed that for every domain tested, the optimization loss drops by one or two orders of magnitude.
when the number of states reaches the correct number. We therefore use the optimization losses to create a prior on the number of states (defined as \( \alpha \) in Sec. 4.1). If the optimization loss for a certain number of states \( F \) is \( c_F \), then the prior for \( F \) states is \(-\log(c_F / c_{F-1})\). We also use the transition weights as prior values for \( \beta \) in the Bayesian model.

### 5 Experiments & Results

#### 5.1 Generating Expert Data

**Linear Temporal Logic** We use linear temporal logic (LTL) to formally specify tasks (Clarke, Grumberg, and Peled 2001). Formulae \( \phi \) have the syntax grammar

\[
\phi ::= p \mid \neg \phi \mid \phi_1 \lor \phi_2 \mid \diamond \phi \mid \phi_1 \mathcal{U} \phi_2
\]

where \( p \) is a **proposition** (a boolean-valued truth statement that can correspond to objects or goals in the world), \( \neg \) is negation, \( \lor \) is disjunction, \( \diamond \) is “next”, and \( \mathcal{U} \) is “until”. The derived rules are conjunction (\( \land \)), implication (\( \implies \)), equivalence (\( \iff \)), “eventually” (\( \diamond \phi \equiv \text{True}\mathcal{U}\phi \)) and “always” (\( \Box \phi \equiv \neg \diamond \neg \phi \)) (Baier and Katoen 2008). \( \phi_1 \mathcal{U} \phi_2 \) means that \( \phi_1 \) is true until \( \phi_2 \) is true, \( \diamond \phi \) means that there is a time where \( \phi \) is true and \( \Box \phi \) means that \( \phi \) is always true.

**Generating Data** We use SPOT (Duret-Lutz et al. 2016) and Lomap (Ulusoy et al. 2013) to convert LTL formulae into FSAs. Every FSA that we consider has a goal state \( G \) into FSAs. Every FSA that we consider has a goal state and Lomap (Ulusoy et al. 2013) to convert LTL formulae to formally specify tasks (Clarke, Grumberg, and Peled 2001). Formulae \( \phi \) have the syntax grammar

\[
\phi ::= p \mid \neg \phi \mid \phi_1 \lor \phi_2 \mid \diamond \phi \mid \phi_1 \mathcal{U} \phi_2
\]

where \( p \) is a **proposition** (a boolean-valued truth statement that can correspond to objects or goals in the world), \( \neg \) is negation, \( \lor \) is disjunction, \( \diamond \) is “next”, and \( \mathcal{U} \) is “until”. The derived rules are conjunction (\( \land \)), implication (\( \implies \)), equivalence (\( \iff \)), “eventually” (\( \diamond \phi \equiv \text{True}\mathcal{U}\phi \)) and “always” (\( \Box \phi \equiv \neg \diamond \neg \phi \)) (Baier and Katoen 2008). \( \phi_1 \mathcal{U} \phi_2 \) means that \( \phi_1 \) is true until \( \phi_2 \) is true, \( \diamond \phi \) means that there is a time where \( \phi \) is true and \( \Box \phi \) means that \( \phi \) is always true.

**Generating Data** We use SPOT (Duret-Lutz et al. 2016) and Lomap (Ulusoy et al. 2013) to convert LTL formulae into FSAs. Every FSA that we consider has a goal state \( G \), which is the desired final state of the agent, and a trap state \( T \), which is an undesired terminal state. We generate a set of environments in which obstacles and other propositions are randomly placed. Given the FSA and an environment, we run Dijkstra’s shortest path algorithm to create expert trajectories that we use as data for imitation learning.

**LSTM Baseline** We compare the performance of LVIN to an LSTM network, which we take to be a generic method for dealing with time-series data. The first layer of the network is a 3D CNN with 1024 channels. The second layer is an LSTM with 1024 hidden units. The hidden units do not directly correspond to logic states or a TM.

#### 5.2 Environments

**Lunchbox Domain** The lunchbox domain (Fig. 4a) is an \( 18 \times 7 \) gridworld where the agent must first pick up either a sandwich \( a \) or a burger \( b \) and put it in a lunchbox \( d \), and then pick up a banana \( c \) and put it in the lunchbox \( d \). The specification is \( \diamond((a \lor b) \land \diamond(d \land \diamond(c \land \diamond d))) \land \Box \neg \alpha \).

**Cabinet Domain** The cabinet domain is a \( 10 \times 10 \) gridworld where the agent must open a cabinet. First it must check if the cabinet is locked \( (c) \). If the cabinet is locked \( (l) \), the agent must get the key \( (gk) \), unlock the cabinet \( (uc) \), and open it \( (op) \). If the cabinet is unlocked \( (uo) \), then the agent can open it \( (ap) \). The specification is \( \diamond(c \land \diamond((uо \land \diamond op) \lor (lо \land (gk \land \diamond(uc \land \diamond op)))))) \land \Box \neg \alpha \).

**Driving Domain** The driving domain (Fig. 4c) is a \( 14 \times 14 \) gridworld where the agent must obey three “rules of the road” – prefer the right lane over the left lane \( (l) \); \( \Box \neg l \); stop at red lights \( (r) \) until they turn green \( (h) \); \( \Box (r \implies (r U h)) \); and reach the goal \( (g) \) while avoiding obstacles \( (o) \); \( \diamond g \land \Box \neg o \). Unlike the other domains, this domain has a time-varying element (the red lights turn green); it also has an extra action – “do not move” – since the car must sometimes wait at the red light.

**Performance** We ran experiments using an Intel i9 processor and an Nvidia 1080Ti GPU. The simplest environment takes \( \sim 1.4 \) hours to train; the most complicated takes \( \sim 7.5 \) days to train. The KL divergence for all environments shows a typical training pattern in which the divergence rapidly decreases before flattening out. Runtimes and loss curves for all environments can be found in Araki et al. (2019a).

<table>
<thead>
<tr>
<th></th>
<th>Training</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LVIN</td>
<td>LSTM</td>
</tr>
<tr>
<td>Lunchbox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set size</td>
<td>500</td>
<td>9000</td>
</tr>
<tr>
<td>Success Rate</td>
<td>99.60%</td>
<td>91.44%</td>
</tr>
<tr>
<td></td>
<td>1800</td>
<td>1800</td>
</tr>
<tr>
<td></td>
<td>1800</td>
<td>1800</td>
</tr>
<tr>
<td></td>
<td>100.00%</td>
<td>93.60%</td>
</tr>
<tr>
<td></td>
<td>1800</td>
<td>1800</td>
</tr>
<tr>
<td></td>
<td>82.44%</td>
<td>82.44%</td>
</tr>
<tr>
<td></td>
<td>1800</td>
<td>1800</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>6000</td>
</tr>
<tr>
<td>Success Rate</td>
<td>100.00%</td>
<td>58.54%</td>
</tr>
<tr>
<td></td>
<td>1800</td>
<td>1800</td>
</tr>
<tr>
<td></td>
<td>100.00%</td>
<td>58.60%</td>
</tr>
<tr>
<td></td>
<td>1800</td>
<td>1800</td>
</tr>
</tbody>
</table>

Table 1: Training and test performance of LVIN vs. LSTM

Figure 4: Example instances of three domains
Performance of LVIN (shorthand for our model) vs. the LSTM network is shown in Table 1. We measure “success rate” as the proportion of trajectories where the agent satisfies the environment’s specification. LVIN achieves virtually perfect performance on every domain with relatively little data. The LSTM network achieves fairly high performance on the lunchbox and cabinet domains, but has poor performance in the time-varying driving domain. On top of achieving better performance than the LSTM network, the LVIN model also has an interpretable output that can be modified to change the learned policy.

The LVIN model requires much less data than the LSTM network for two reasons. One is that the LVIN model can take advantage of the spectral learning prior to reduce the amount of data needed to converge to a solution, whereas the LSTM network cannot use the prior. The second is that since the LVIN model is model-based, once it learns an accurate model of the rules it can generalize to unseen permutations of the environment better than the LSTM network, which in a sense only interpolates between data points.

5.3 Interpretability

Our method learns an interpretable model of the rules of an environment in the form of a transition matrix (TM). Learned vs. true TMs are shown in Fig. 5 (we leave out the goal and trap states of the TMs, since they are trivial). The plots show values of the learned variational parameter $\beta$. Therefore the plots do not show the values of the actual TM but rather the values of the prior of the TM, giving an idea of how “certain” the model is of each transition.

Fig. 5a shows the TM of the lunchbox domain. Each matrix corresponds to the transitions associated with a given automaton state. Columns correspond to propositions ($e$ is the empty proposition) and rows correspond to the probability that, given current state $f$ and proposition $p$, the next state is $f'$. Inspection of the learned TM shows that in the initial state $S0$, picking up the sandwich or burger ($a$ or $b$) leads to state $S1$. In $S1$, putting the sandwich/burger into the lunchbox ($d$) leads to $S2$. In $S2$, picking up the banana $c$ leads to $S3$, and in $S3$, putting the banana in the lunchbox $d$ leads to the goal state $G$.

The other domains can be interpreted similarly, and most of them closely match their expected TMs (see Araki et al. (2019a) for more examples). One exception is the driving domain (Fig. 5c). In the expected TM, the initial state $S0$ transitions to a lower-reward state $S1$ when the car enters the left lane (indicating that the left lane is allowed but unideal); $S0$ transitions to $S2$ when the car is at a red light, and then back to $S0$ when the green light $h$ turns on. Our model learns a different TM – but due to the nature of the TM, it can still be interpreted. Unlike in the “true” TM, in the learned TM, the green light acts as a switch – the agent cannot reach the goal state unless it has encountered the green light. This is an artifact of the domain, since the agent always passes a green light before reaching the goal. The red light leads from $S0$ to $S1$, which is a lower-reward duplicate of $S0$. The agent will wait for the red light to turn green because it thinks it must encounter a green light before it can reach the goal. Regarding the left lane, the TM places significant weight on a transition to low-reward $S1$ when in $S0$, which discourages the agent from entering the left lane. Therefore although not as tidy as the true TM, the learned TM is still interpretable.

5.4 Manipulability Experiments on Jaco Arm

Our method allows the learned policy to be manipulated to produce reliable new behaviors. We demonstrate this ability on a real-word platform, a Jaco arm. The Jaco arm is a 6-DOF arm with a 3-fingered hand and a mobile base. An Optitrack motion capture system was used to track the hand and manipulated objects. The system was implemented using ROS (Quigley et al. 2009). The Open Motion Planning
We modified the learned TMs of the lunchbox and cabinet domains. We call the original lunchbox specification $\phi_1$. We tested three modified specifications – pick up the sandwich first, then the banana ($\phi_{12}$); pick up the burger first, then the banana ($\phi_{13}$); and pick up the banana, then either the sandwich or the burger ($\phi_{14}$). These experiments are analogous to the ones in Araki et al. (2019b) and are meant to show that though significantly less information was given to our model in the learning process, it can still perform just as well.

We also modified the learned cabinet TM ($\phi_{c,1}$) – if we know that the cabinet is locked, we have the agent pick up the key first before checking the cabinet ($\phi_{c,2}$). The modifications to the TM are shown in Fig. 6. The TM must be modified so that the agent will get the key ($g_k$) before checking the cabinet ($cc$). So in the initial state $S_0$, $cc$ is set to go to the trap state so that the agent will avoid it; $g_k$ is set to transition to $S_2$, indicating to the agent that it should get the key first. In $S_2$, we then modify the TM so that $g_k$ is no longer the goal but rather $cc$ is – in other words, the agent will then head to the cabinet and check it. Finally, in $S_4$, once the agent has checked the cabinet, it must unlock the cabinet and it does not need to get the key, so we set $g_k$ to the trap state so the agent will be certain to unlock the cabinet and not try to get the key. These modifications, as shown in our experiments, successfully change the behavior of the agent to always get the key before checking the cabinet.

Each specification was tested 20 times on our experimental platform; as shown in Table 2 there were only a few failures, and these were all due to mechanical failures of the Jako arm, such as the manipulator dropping an object or losing its grasp on the cabinet key.

<table>
<thead>
<tr>
<th>Lunchbox</th>
<th>Cabinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi_{11}$</td>
<td>$\phi_{c,1}$</td>
</tr>
<tr>
<td>$\phi_{12}$</td>
<td>$\phi_{c,2}$</td>
</tr>
<tr>
<td>$\phi_{13}$</td>
<td>$cc$</td>
</tr>
<tr>
<td>$\phi_{14}$</td>
<td>$g_k$</td>
</tr>
</tbody>
</table>

Table 2: Performance of Jako robot in executing learned lunchbox and cabinet tasks

We are grateful to the many people who gave us invaluable advice on this project, including Leslie Kaelbling, Scott Linderman, Keyon Vafa, and David Blei. This material is supported by NSF grant 1723943, ONR N00014-18-1-2830, and the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. K.V. is supported by an NSF Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References


