Learning structured transition models for multi-object manipulation

Victoria Xia Zi Wang Leslie Pack Kaelbling

***Abstract*— We propose to learn the structure and parameters of rule-based probabilistic transition models for manipulation actions with the presence of multiple objects. For each rule, we use deictic references to myopically select relevant objects and learn a neural network predictor on the transition of their states. Our algorithm simultaneously assigns training data to each rule and learns the rule parameters through clustering and EM-like methods. We evaluate learned rules on variants of a simulated, 3D table-top pushing task involving stacks of objects. Comparing to a baseline that uses a vector representation of all the objects in the scene, our approach with a template representation of the model is more data-efficient and performs better under varying numbers of objects in the scene.**

I. Introduction

Consider a household robot preparing dinner in an unfamiliar kitchen, or organizing papers in a cluttered office space. The robot must be able to perform a wide variety of object-manipulation tasks, such as opening cabinets, turning the knob on the stove, gathering papers, and so on. More importantly, the robot must have an understanding of how an action will affect the world state, i.e. the robot configuration and the poses of all of the objects in the room. Only then can the robot plan a sequence of actions to achieve possibly long-horizon, high-level goals like preparing a meal. This work focuses on the acquisition of this understanding, in the form of a *transition model*. Specifically, given an initial world state and an action, we wish to predict the resulting state of the world after the action is taken.

Even simple actions have complex effects. Consider a robot pushing an object, i.e. the target, on a table. Because of the absence of detailed information about the world, e.g. the imprecise exertion of force or uneven friction of surfaces, the pushing action cannot produce deterministic effects even if only the target is on the table [1]. Moreover, given the presence of additional objects close to the target, the push can be prevented from succeeding or other objects may move as well, such as any on top of the target object or in the path of the push. As a consequence, the resulting change in the world state is not only stochastic, but it also includes the poses of multiple objects; hence, it is very challenging to represent and learn a probabilistic transition model.

In this work, we propose to represent the continuous-space transition distribution using a rule language that is a generalization of PPDDL (probabilistic planning domain description language) operator descriptions to continuous domains. While actions may produce complex effects, the type of effects that may result are often structured and can be grouped into a small number of categories that capture plausible outcomes. By characterizing these structured categories, we obtain *rules* for how the state of the world changes as a result of an action.

Continuing our simple example of pushing, though the push action may have an effect on the target object, the objects on top the target and the nearby objects, other objects far away or at the opposite direction of the push are less likely to be affected. By capturing these structural relations between objects, henceforth referred to as *deictic references* [2], [3], we can represent the possible outcomes of an action as a collection of rules. This collection of rules is in turn a representation of a transition model.

To predict how a particular action will affect the objects in a scene, a rule starts with the target object of the action and pull in as context some small number of additional objects related to the target in predefined ways (e.g., objects on top of the target, in contact with the target, within some given radius of the target, etc.). Based on properties of these selected objects only, the rule will make a prediction for how the action will affect some subset of objects in the scene, again chosen by structural relations starting from the target object of the action. All other objects that are not selected by the rule are assumed to remain unchanged.

Because rules are agnostic to the total number of objects in the scene, the same, relatively "small" rules can be applied to a variety of complicated scenes with many objects, making rules a compact and general representation of a transition model. Because each rule is only responsible for the objects selected by the deictic references (i.e., structural relations between objects), it requires fewer training examples to make good predictions and scales easily to a large number of objects in the scene.

Main contributions: We specify the language of rules and how they can be used to predict outcomes of actions. We then introduce our algorithm for learning such rules. Finally, we evaluate our algorithm and the performance of rules on a simulated 3D pushing task.

II. Related Work

The representation and learning of transition models for capturing the effects of actions in the environment has been studied extensively. A variety of representations have been presented, each with their own learning algorithms.

Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 77 Massachusetts Ave., Cambridge, MA 02139. {vxia, ziw, lpk}@mit.edu.

We gratefully acknowledge support from NSF grants 1420316, 1523767, and 1723381; from AFOSR grant FA9550-17-1-0165; from ONR grant N00014-18-1-2847; from Honda Research; and from Draper Laboratory. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

These representations include the representation of actions as STRIPS actions [4], for which proposed learning approaches include logical inference [5], [6], kernel perceptrons [7], etc, for deterministic actions. To consider stochastic action effects, action representations include "schemas" [8], [9] where states consist of some number of discrete sensor values, probabilistic STRIPS operators [10], and "affordances" [11], [12] that capture the relation between an object, an action, and an effect, so that given any two, predictions can be made for the third.

Our work on is mostly inspired by [3], which introduces and presents a learning method for *Noisy Deictic Rules* (NDRs), an extension of probabilistic STRIPS operators to include a possible noise outcome, as well as deictic references that refer to objects relative to the actions for which predictions are being made. NDRs are learned via a greedy search procedure involving three nested layers of search, where each layer progressively fills in more structure or details of the model. However, our rules differ substantially from NDRs in that, rather than using a predicate-based state description, our rules operate on continuous state spaces described by object features and relations, such as mass and relative pose. As a consequence, our rules represent predictions for the outcomes of stochastic actions as a continuous distribution over resultant object features, rather than as a collection of possible discrete outcomes with associated probabilities.

III. Problem formulation

In this section, we formalize this class of problems, define a new rule structure for specifying probabilistic transition models for these problems, and articulate an objective function for estimating these models from data.

A. Relational domain

A problem *domain* is given by tuple $\mathcal{D} = (\Upsilon, \mathcal{P}, \mathcal{F}, \mathcal{A})$ where Υ is a countably infinite universe of possible objects, $\mathcal{P}$ is a finite set of properties $P_i : \Upsilon \mapsto \mathbb{R}, i \in [N_\mathcal{P}] = \{1, \cdots, N_\mathcal{P}\}$, and $\mathcal{F}$ is a finite set of deictic reference functions $F_i : \Upsilon^{m_i} \mapsto \wp(\Upsilon), i \in [N_\mathcal{F}]$ where $\wp(\Upsilon)$ denotes the powerset of Υ. Each function $F_i \in \Upsilon$ maps from an ordered list of objects to a set of objects, and we define it as

$$F_i(o_1,\ldots,o_{m_i}) = \{o | f_i(o,o_1,\ldots,o_{m_i}) = \text{True},\ o,o_j \in \Upsilon, \forall j \in [m_i]\}\ ,$$

where the relation $f_i : \Upsilon^{m_i+1} \mapsto \{\text{True, False}\}$ is defined in terms of the object properties in $\mathcal{P}$. For example, if we have a location property P_{loc} and $m_i = 1$, we can define $f_i(o, o_1) = \mathbb{1}_{\|P_{\text{loc}}(o) - P_{\text{loc}}(o_1)\| < 0.5}$ so that the function F_i associated with f_i maps from one object to the set of objects that are within 0.5 distance of its center; here $\mathbb{1}$ is an indicator function. Finally, $\mathcal{A}$ is a set of *action templates* $A_i : \mathbb{R}^{d_i} \times \Upsilon^{n_i} \mapsto \Psi, i \in [N_\mathcal{A}]$, where Ψ is the space of executable control programs. Each action template is a function parameterized by continuous parameters $\alpha_i \in \mathbb{R}^{d_i}$ and a tuple of n_i objects that the action operates on. In this work, we assume that $\mathcal{P}, \mathcal{F}$ and $\mathcal{A}$ are given.[1]

A problem *instance* is characterized by $\mathcal{I} = (\mathcal{D}, \mathcal{U})$, where $\mathcal{D}$ is a domain defined above and $\mathcal{U} \subset \Upsilon$ is a finite universe of objects with $|\mathcal{U}| = N_\mathcal{U}$. For simplicity, we assume that, for a particular instance, the universe of objects remains constant over time. In the problem instance $\mathcal{I}$, we characterize a *state* s in terms of the concrete values of all properties in $\mathcal{P}$ on all objects in $\mathcal{U}$; that is, $s = [P_i(o_j)]_{i=1,j=1}^{N_\mathcal{P}, N_\mathcal{U}} \in \mathbb{R}^{N_\mathcal{P} \times N_\mathcal{U}} = \mathfrak{S}$. A problem instance induces the definition of its *action* space $\mathfrak{A}$, constructed by applying every action template $A_i \in \mathcal{A}$ to all tuples of n_i elements in $\mathcal{U}$ and all assignments α_i to the continuous parameters; namely, $\mathfrak{A} = \{A_i(\alpha_i, [o_{ij}]_{j=1}^{n_i}) \mid o_{ij} \in \mathcal{U}, \alpha_i \in \mathbb{R}^{d_i}\}$.

B. Sparse relational transition models

In many domains, there is substantial uncertainty, and the key to robust behavior is the ability to model this uncertainty and make plans that respect it. A *sparse relational transition model* (SPARE) for a domain $\mathcal{D}$, when applied to a problem instance $\mathcal{I}$ for that domain, defines a probability density function on the resulting state s' resulting from taking action a in state s. Our objective is to specify this function in terms of domain elements $\mathcal{P}$, $\mathcal{R}$, and $\mathcal{F}$ in such a way that it will apply to any problem instance, independent of the number and properties of the objects in its universe. We achieve this by defining the transition model in terms of a set of *transition rules*, $\mathcal{T} = \{T_k\}_{k=1}^K$ and a score function $C : \mathcal{T} \times \mathfrak{S} \mapsto \mathbb{N}$. The score function takes in as input a state s and a rule $T \in \mathcal{T}$, and outputs a non-negative integer. If the output is 0, the rule does not apply; otherwise, the rule can predict the distribution of the next state to be $p(s' \mid s, a; T)$. The final prediction of SPARE is

$$p(s' \mid s, a; \mathcal{T}) = \begin{cases} \frac{1}{|\hat{\mathcal{T}}|} \sum_{T \in \hat{\mathcal{T}}} p\left(s' \mid s, a; T\right) & \text{if } |\hat{\mathcal{T}}| > 0 \\ \mathcal{N}(s, \Sigma_{\text{default}}) & \text{otherwise} \end{cases}\ , \tag{1}$$

where $\hat{\mathcal{T}} = \arg\max_{T \in \mathcal{T}} C(T, s)$ and the matrix $\Sigma_{\text{default}} = \boldsymbol{I}_{N_\mathcal{U}} \otimes \text{diag}([\sigma_i]_{i=1}^{N_\mathcal{P}})$ is the default predicted covariance for any state that is not predicted to change, so that our problem is well-formed in the presence of noise in the input. Here $\boldsymbol{I}_{N_\mathcal{U}}$ is an identity matrix of size $N_\mathcal{U}$, and $\text{diag}([\sigma_i]_{i=1}^{N_\mathcal{P}})$ represents a square diagonal matrix with σ_i on the main diagonal, denoting the default variance for property P_i if no rule applies. In the rest of this section, we formalize the definition of transition rules and the score function.

Transition rule $T = (A, \Gamma, \Delta, \phi_\theta, \boldsymbol{v}_{\text{default}})$ is characterized by an action template A, two ordered lists of *deictic references* Γ and Δ of size N_Γ and N_Δ, a predictor ϕ_θ and the default variances $\boldsymbol{v}_{\text{default}} = [v_i]_{i=1}^{N_\mathcal{P}}$ for each property P_i under this rule. The action template is defined as operating on a tuple of n object variables, which we will refer to

[1] There is a direct extension of this formulation in which we encode relations among the objects as well. Doing so complicates notation and adds no new conceptual ideas, and in our example domain it suffices to compute spatial relations from object properties so there is no need to store relational information explicitly, so we omit it from our treatment.

as $O^{(0)} = (O_i)_{i=1}^{n}, O_i \in \mathcal{U}, \forall i$. A reference list uses functions to designate a list of additional objects or sets of objects, by making *deictic* references based on previously designated objects. In particular, Γ generates a list of objects whose properties affect the prediction made by the transition rule, while Δ generates a list of objects whose properties are affected after taking an action specified by the action template A.

We begin with the simple case in which every function returns a single object, then extend our definition to the case of sets. Concretely, for the t-th element γ_t in Γ ($t \in [N_\Gamma]$), $\gamma_t = (F, (O_{k_j})_{j=1}^{m})$ where $F \in \mathcal{F}$ is a deictic reference function in the domain, m is the arity of that function, and integer $k_j \in [n + t - 1]$ specifies that object O_{n+t} in the object list can be determined by applying function F to objects $(O_{k_j})_{j=1}^{m}$. Thus, we get a new list of objects, $O^{(t)} = (O_i)_{i=1}^{n+t}$. So, reference γ_1 can only refer to the objects $(O_i)_{i=1}^{n}$ that are named in the action, and determines an object O_{n+1}. Then, reference γ_2 can refer to objects named in the action or those that were determined by reference γ_1, and so on.

When the function F in $\gamma_t = (F, (O_{k_j})_{j=1}^{m}) \in \Gamma$ returns a set of objects rather than a single object, this process of adding more objects remains almost the same, except that the O_t may denote sets of objects, and the functions that are applied to them must be able to operate on sets. In the case that a function F returns a set, it must also specify an *aggregator*, g, that can return a single value for each property $P_i \in \mathcal{P}$, aggregated over the set. Examples of aggregators include the mean or maximum values or possibly simply the cardinality of the set.

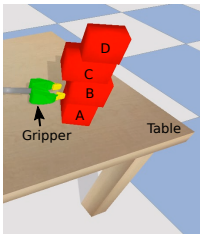

Fig. 1. A robot gripper is pushing a stack of 4 blocks on a table.

For example, consider the case of pushing the bottom (block A) of a stack of 4 blocks, depicted in Figure 1. Suppose the deictic reference is F =above, which takes one object and returns a set of objects immediately on top of the input object. Then, by applying F =above starting from the initial set $O_0 = \{A\}$, we get an ordered list of sets of objects (O_0, O_1, O_2) where $O_1 = F(O_0) = \{B\}, O_2 = F(O_1) = \{C\}$.

Returning to the definition of a transition rule, we now can see informally that if the parameters of action template A are instantiated to actual objects in a problem instance, then Γ and Δ can be used to determine lists of *input* and *output* objects (or sets of objects). We can use these lists, finally, to construct input and output vectors. The input vector $\boldsymbol{x}$ consists of the continuous action parameters α of action A and property $P_i(O_t)$ for all properties $P_i \in \mathcal{P}$ and objects $O_t \in O^{N_\Gamma}$ that are selected by Γ in arbitrary but fixed order. In the case that O_t is a set of size greater than one, the aggregator associated with the function F that computed the reference is used to compute $P_i(O_t)$. Similar for the desired output construction, we use the references in the list Δ, initialize $\hat{O}^{(0)} = O^{(0)}$, and gradually add more objects to

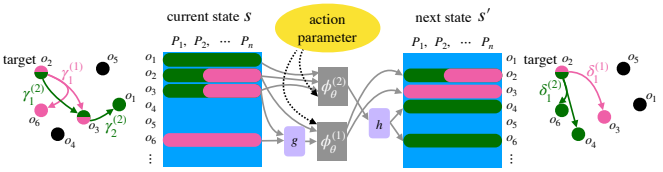

Fig. 2. Instead of directly mapping from current state s to next state s', our prediction model uses deictic references to find subsets of objects for prediction. In the left most graph, we illustrate what relations are used to construct the input objects with two rules for the same action template, $T_1 = (A, \Gamma^{(1)}, \Delta^{(1)}, \phi_\theta^{(1)}, \boldsymbol{v}_{\text{default}}^{(1)})$ and $T_2 = (A, \Gamma^{(2)}, \Delta^{(2)}, \phi_\theta^{(2)}, \boldsymbol{v}_{\text{default}}^{(2)})$, where the reference list $\Gamma^{(1)} = [(\gamma_1^{(1)}, o_2)]$ applied a deictic reference $\gamma_1^{(1)}$ to the target object o_2 and added input features computed by an aggregator g on o_3, o_6 to the inputs of the predictor of rule T_1. Similarly for $\Gamma^{(2)} = [(\gamma_1^{(2)}, o_2), (\gamma_2^{(2)}, o_3)]$, the first deictic reference selected o_3 and then $\gamma_2^{(2)}$ is applied on o_3 to get o_1. The predictors $\phi_\theta^{(1)}$ and $\phi_\theta^{(2)}$ are neural networks that map the fixed-length input to a fixed-length output, which is applied to a set of objects computed from a relational graph on all the objects, derived from the reference list $\Delta^{(1)} = [(\delta_1^{(1)}, o_2)]$ and $\Delta^{(2)} = [(\delta_1^{(2)}, o_2)]$, to compute the whole next state s'. Because $\delta_1^{(2)}(o_2) = (o_4, o_6)$ and the $\phi_\theta^{(2)}$ is only predicting a single property, we use a "de-aggregator" function h to assign its prediction to both objects o_4, o_6.

construct the output set of objects $\hat{O} = \hat{O}^{(N_\Delta)}$. The output vector is $\boldsymbol{y} = [P(\hat{o})]_{\hat{o} \in \hat{O}, P \in \mathcal{P}}$ where if $\hat{o}$ is a set of objects, we apply a mean aggregator on the properties of all the objects in $\hat{o}$.

The *predictor* ϕ_θ is some functional form ϕ (such as a feed-forward neural network) with parameters (weights) θ that will take values $\boldsymbol{x}$ as input and predict a distribution for the output vector $\boldsymbol{y}$. It is difficult to represent arbitrarily complex distributions over output values. In this work, we restrict ourselves to representing a Gaussian distributions on all property values in $\boldsymbol{y}$, encoded with a mean and independent variance for each dimension.

Now, we describe how a transition rule can be used to map a state and action into a distribution over the new state. A transition rule $T = (A, \Gamma, \Delta, \phi_\theta, \boldsymbol{v}_{\text{default}})$ *applies* to a particular state-action (s, a) pair if a is an instance of A and if none of the elements of the input or output object lists is empty. To construct the input (and output) list, we begin by assigning the actual objects $o_1, \ldots, o_n$ to the object variables $O_1, \ldots, O_n$ in action instance a, and then successively computing references $\gamma_i \in \Gamma$ based on the previously selected objects, applying the definition of the deictic reference F in each γ_i to the actual values of the properties as specified in the state s. If, at any point, a $\gamma_i \in \Gamma$ or $\delta_i \in \Delta$ returns an empty set, then the transition rule does not apply. If the rule does apply, and successfully selects input and output object lists, then the values of the input vector $\boldsymbol{x}$ can be extracted from s, and predictions are made on the mean and variance values $\Pr(\boldsymbol{y} \mid \boldsymbol{x}) = \phi_\theta(\boldsymbol{x}) = \mathcal{N}(\mu_{\theta_1}(\boldsymbol{x}), \Sigma_{\theta_2}(\boldsymbol{x}))$.

Let $\left(\mu_{\theta_1}^{(ij)}(\boldsymbol{x}), \Sigma_{\theta 2}^{(ij)}(\boldsymbol{x})\right)$ be the vector entry corresponding to the predicted Gaussian parameters of property P_i of j-th output object set $\hat{o}_j$ and denote $s[o, P_i]$ as the property P_i of object o in state s, for all $o \in \mathcal{U}$. The predicted distribution

of the resulting state $p(s' \mid s, a; T)$ is computed as follows:

$$p(s'[o,P_i]|s,a;T)=\begin{cases}\frac{1}{|J|}\sum_{j\in J}\mathcal{N}(\mu_{\theta_1}^{(ij)}(\boldsymbol{x}),\Sigma_{\theta 2}^{(ij)}(\boldsymbol{x})) & \text{if } |J|>0\\ \mathcal{N}(s[o,P_i],v_i) & \text{otherwise}\end{cases}$$

where $J = \{j : o \in \hat{o}_j\}$ and $v_i \in \boldsymbol{v}_{\text{default}}$ is the default variance of property P_i in rule T. There are two important points to note. First, it is possible for the same object to appear in the object-list more than once, and therefore for more than one predicted distribution to appear for its properties in the output vector. In this case, we use the mixture of all the predicted distributions with uniform weights. Second, when an element of the output object list is a set, then we treat this as predicting the same single property distribution for all elements of that set. This strategy has sufficed for our current set of examples, but an alternative choice would be to make the predicted values be *changes* to the current property value, rather than new absolute values. Then, for example, moving all of the objects on top of a tray could easily specify a change to each of their poses. We illustrate how we can use transition rules to build a SPARE in Fig. 2.

For each transition rule $T_k \in \mathcal{T}$ and state $s \in \mathfrak{S}$, we assign the ***score function*** value to be 0 if T_k does not apply to state s. Otherwise, we assign the total number of deictic references plus one, $N_\Gamma + N_\Delta + 1$, as the score. The more references there are in a rule that is applicable to the state, the more detailed the match is between the rules conditions and the state, and the more specific the predictions we expect it to be able to make.

C. Learning SPAREs from data

We frame the problem of learning a transition model from data in terms of conditional likelihood. The learning problem is, given a *problem domain description* $\mathcal{D}$ and a set of experience $\mathcal{E}$ tuples, $\mathcal{E} = \{(s^{(i)}, a^{(i)}, s'^{(i)})\}_{i=1}^n$, find a SPARE $\mathcal{T}$ that minimizes the loss function:

$$\mathcal{L}(\mathcal{T}; \mathcal{D}, \mathcal{E}) = -\frac{1}{n}\sum_{i=1}^{n} \log \Pr(s'^{(i)} \mid s^{(i)}, a^{(i)}; \mathcal{T}) \ . \quad (2)$$

Note that we require all of the tuples in $\mathcal{E}$ to belong to the same *domain* $\mathcal{D}$, and require for any $(s^{(i)}, a^{(i)}, s'^{(i)}) \in \mathcal{E}$ that $s^{(i)}$ and $s'^{(i)}$ belong to the same problem *instance*, but individual tuples may be drawn from different problem instances (with, for example, different numbers and types of objects). In fact, to get good generalization performance, it will be important to vary these aspects across training instances.

IV. Learning algorithm

We describe our learning algorithm in three parts. First, we introduce our strategy for learning ϕ_θ, which predicts a Gaussian distribution on $\boldsymbol{y}$, given $\boldsymbol{x}$. Then, we describe our algorithm for learning reference lists Γ and Δ for a single transition rule, which enable the extraction of $\boldsymbol{x}$ and $\boldsymbol{y}$ from $\mathcal{E}$. Finally, we present an EM method for learning multiple rules.

A. Distributional prediction

For a particular transition rule T with associated action template A, once Γ and Δ have been specified, we can extract input and output features $\boldsymbol{x}$ and $\boldsymbol{y}$ from a given set of experience samples $\mathcal{E}$. From $\boldsymbol{x}$ and $\boldsymbol{y}$, we would like to learn the transition rule's predictor ϕ_θ to minimize Eq. (2). Our predictor takes the form $\phi_\theta(\boldsymbol{x}) = \mathcal{N}(\mu_{\theta_1}(\boldsymbol{x}), \Sigma_{\theta_2}(\boldsymbol{x}))$. We use the method of [13], where two neural-network models are learned as the function approximators, one for the mean prediction $\mu_{\theta_1}(\boldsymbol{x})$ parameterized by θ_1 and the other for the diagonal variance prediction $\Sigma_{\theta_2}(\boldsymbol{x})$, parameterzied by θ_2. We optimize the negative data-likelihood loss function

$$\mathcal{L}(\theta,\Gamma,\Delta;\mathcal{D},\mathcal{E})=\tfrac{1}{n}\textstyle\sum_{i=1}^{n}((\boldsymbol{y}^{(i)}-\mu_{\theta_1}(\boldsymbol{x}^{(i)}))^{\mathrm{T}}\Sigma_{\theta_2}(\boldsymbol{x})^{-1}(\boldsymbol{y}^{(i)}-\mu_{\theta_1}(\boldsymbol{x}^{(i)})) + \log\det\Sigma_{\theta_2}(\boldsymbol{x}^{(i)}))$$

by alternatively optimizing θ_1 and θ_2. That is, we alternate between first optimizing θ_1 with fixed covariance, and then optimizing θ_2 with fixed mean.

Let $\mathcal{E}_T \in \mathcal{E}$ be the set of experience tuples to which rule T applies. Then once we have θ, we can optimize the default variance of the rule $T = (A, \Gamma, \Delta, \phi_\theta, \boldsymbol{v}_{\text{default}})$ by optimizing $\mathcal{L}(\{T\}; \mathcal{D}, \mathcal{E}_T)$. It can be shown that these loss-minimizing values for the default predicted variances $\boldsymbol{v}_{\text{default}}$ are the empirical averages of the squared deviations for all unpredicted objects (i.e., those for which ϕ_θ does not explicitly make predictions), where averages are computed separately for each object property.

We use $\theta, \boldsymbol{v}_{\text{default}} \leftarrow \textsc{LearnDist}(\mathcal{D}, \mathcal{E}, \Gamma, \Delta)$ to refer to this learning and optimization procedure for the predictor parameters and default variance.

Algorithm 1 Greedy procedure for constructing Γ.

procedure GREEDYSELECT($\mathcal{D}, \mathcal{E}, A, \Delta, N_\Gamma$)
  train model using $\Gamma_0 = \emptyset$, save loss L_0
  $i \leftarrow 1$
  while $i \leq N_\Gamma$ **do**
    $\gamma_i \leftarrow$ None ; $L_i \leftarrow \infty$
    for all $\gamma \in R_i$ **do**
      $\Gamma_i \leftarrow \Gamma_{i-1} \cup \{\gamma\}$
      $\theta, \boldsymbol{v}_{\text{default}} \leftarrow \textsc{LearnDist}(\mathcal{D}, \mathcal{E}_{train}, \Gamma_i, \Delta)$
      $l \leftarrow \mathcal{L}(\mathcal{T}_\gamma; \mathcal{D}, \mathcal{E}_{val})$
      if $l < L_i$ **then** $L_i \leftarrow l$; $\gamma_i \leftarrow \gamma$
    if $L_i < L_{i-1}$ **then** $\Gamma_i \leftarrow \Gamma_{i-1} \cup \{\gamma_i\}$; $i \leftarrow i+1$
    else break

B. Single rule

In the simple setting where only one transition rule T exists in our domain $\mathcal{D}$, we show how to construct the input and output reference lists Γ and Δ that will determine the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. Suppose for now that Δ and $\boldsymbol{v}_{\text{default}}$ are fixed, and we wish to learn Γ. Our approach is to incrementally build up Γ by adding $\gamma_i = (F, (O_{k_j})_{j=1}^m)$ tuples one at a time via a greedy selection procedure. Specifically, let R_i be the universe of possible γ_i, split the experience samples $\mathcal{E}$ into a training set $\mathcal{E}_{train}$ and a validation set $\mathcal{E}_{val}$, and initialize the list Γ to be $\Gamma_0 = \emptyset$. For each i, compute $\gamma_i = \arg\min_{\gamma \in R_i} \mathcal{L}(\mathcal{T}_\gamma; \mathcal{D}, \mathcal{E}_{val})$, where $\mathcal{L}$

in Eq. (2) evaluates a SPARE $\mathcal{T}_\gamma$ with a single transition rule $T = (A, \Gamma_{i-1} \cup \{\gamma\}, \Delta, \phi_\theta, \boldsymbol{v}_{\text{default}})$, where θ and $\boldsymbol{v}_{\text{default}}$ are computed using the LEARNDIST described in Section IV-A[2]. If the value of the loss function $\mathcal{L}(\mathcal{T}_{\gamma_i}; \mathcal{D}, \mathcal{E}_{val})$ is less than the value of $\mathcal{L}(\mathcal{T}_{\gamma_{i-1}}; \mathcal{D}, \mathcal{E}_{val})$, then we let $\Gamma_i = \Gamma_{i-1} \cup \{\gamma_i\}$ and continue. Else, we terminate the greedy selection process with $\Gamma = \Gamma_{i-1}$, since further growing the list of deictic references hurts the loss. We also terminate the process when i exceeds some predetermined maximum allowed number of input deictic references, N_Γ. Pseudocode for this algorithm is provided in Algorithm 1.

In our experiments we set $\Delta = \Gamma$ and construct the lists of deictic references using a single pass of the greedy algorithm described above. This simplification is reasonable, as the set of objects that are relevant to predicting the transition outcome often overlap substantially with the objects that are affected by the action. Alternatively, we could learn Δ via an analogous greedy procedure nested around or, as a more efficient approach, interleaved with, the one for learning Γ.

C. Multiple rules

Our training data in robotic manipulation tasks are likely to be best described by many rules instead of a single one, since different combinations of relations among objects could be present in different states. For example, we may have one rule for pushing a single object and another rule for pushing a stack of objects. We now address the case where we wish to learn K rules from a single experience set $\mathcal{E}$, for $K > 1$. We do so via initial clustering to separate experience samples into K clusters, one for each rule to be learned, followed by an EM-like approach to further separate samples and simultaneously learn rule parameters.

To facilitate the learning of our model, we will additionally learn *membership probabilities* $Z = ((z_{i,j})_{i=1}^{|\mathcal{E}|})_{j=1}^{K}$, where $z_{i,j}$ represents the probability that the i-th experience sample is assigned to transition rule T_j, and $\sum_{j=1}^{K} z_{i,j} = 1$ for all $i \in [|\mathcal{E}|]$. We initialize membership probabilities via clustering, then refine them through EM.

Because the experience samples $\mathcal{E}$ may come from different problem instances and involve different numbers of objects, we cannot directly run a clustering algorithm such as k-means on the (s, a, s') samples themselves. Instead we first learn a single transition rule $T = (A, \Gamma, \Delta, \phi_\theta, \boldsymbol{v}_{\text{default}})$ from $\mathcal{E}$ using the algorithm in Section IV-B, use the resulting Γ and Δ to transform $\mathcal{E}$ into $\boldsymbol{x}$ and $\boldsymbol{y}$, and then run k-means clustering on the concatenation of $\boldsymbol{x}$, $\boldsymbol{y}$, and values of the loss function when T is used to predict each of the samples. For each experience sample, the squared distance from the sample to each of the K cluster centers is computed, and membership probabilities for the sample to each of the K transition rules to be learned are initialized to be proportional to the (multiplicative) inverses of these squared distances.

[2]When the rule T does not apply to a training sample, we use for its loss the loss that results from having empty reference lists in the rule. Alternatively, we can compute the default variance Σ_{default} to be the empirical variances on all training samples that cannot use rule T.

Before introducing the EM-like algorithm that simultaneously improves the assignment of experience samples to transition rules and learns details of the rules themselves, we make a minor modification to transition rules to obtain *mixture rules*. Whereas a probabilistic transition rule has been defined as $T = (A, \Gamma, \Delta, \phi_\theta, \boldsymbol{v}_{\text{default}})$, a mixture rule is $T = (A, \pi_\Gamma, \pi_\Delta, \Phi)$, where π_Γ represents a *distribution* over all possible lists of input references Γ (and similarly for π_Δ and Δ), of which there are a finite number, since the set of available reference functions $\mathcal{F}$ is finite, and there is an upper bound N_Γ on the maximum number of references Γ may contain. For simplicity of terminology, we refer to each possible list of references Γ as a *shell*, so π_Γ is a distribution over possible shells. Finally, $\Phi = (\Gamma^{(k)}, \Delta^{(k)}, \phi_{\theta^{(k)}}, \boldsymbol{v}_{\text{default}}^{(k)})_{k=1}^{\kappa}$ is a collection of κ transition rules (i.e., predictors $\phi_{\theta^{(k)}}$, each with an associated $\Gamma^{(k)}$, $\Delta^{(k)}$, and $\boldsymbol{v}_{\text{default}}^{(k)}$). To make predictions for a sample (s, a) using a mixture rule, predictions from each of the mixture rule's κ transition rules are combined according to the probabilities that π_Γ and π_Δ assign to each transition rule's $\Gamma^{(k)}$ and $\Delta^{(k)}$. Rather than having our EM approach learn K transition rules, we instead learn K mixture rules, as the distributions π_Γ and π_Δ allow for smoother sorting of experience samples into clusters corresponding to the different rules, in contrast to the discrete Γ and Δ of regular transition rules.

As before, we focus on the case where for each mixture rule, $\Gamma^{(k)} = \Delta^{(k)}$, $k \in [\kappa]$, and $\pi_\Gamma = \pi_\Delta$ as well. Our EM-like algorithm is then as follows:

1) For each $j \in [K]$, initialize distributions $\pi_\Gamma = \pi_\Delta$ for mixture rule T_j as follows. First, use the algorithm in Section IV-B to learn a transition rule on the *weighted* experience samples $\mathcal{E}_{Z_j}$ with weights equal to the membership probabilities $Z_j = (z_{i,j})_{i=1}^{|\mathcal{E}|}$. In the process of greedily assembling reference lists $\Gamma = \Delta$, data likelihood loss function values are computed for multiple *explored* shells, in addition to the shell $\Gamma = \Delta$ that was ultimately selected. Initialize $\pi_\Gamma = \pi_\Delta$ to distribute weight proportionally, according to data likelihood, for these explored shells: $\pi_\Gamma(\Gamma) = \exp(-\mathcal{L}(\mathcal{T}_\Gamma; \mathcal{D}, \mathcal{E}_{Z_j}))/\chi$, where $\mathcal{T}_\Gamma$ is the SPARE model with a single transition rule $T = (A, \Gamma, \Delta = \Gamma, \phi_\theta)$, and $\chi = (1-\epsilon)\sum_\Gamma \exp(-\mathcal{L}(\mathcal{T}_\Gamma; \mathcal{D}, \mathcal{E}_{Z_j}))$, with the summation taken over all explored shells Γ, is a normalization factor so that the total weight assigned by π_Γ to explored shells is $1 - \epsilon$. The remaining ϵ probability weight is distributed uniformly across unexplored shells.

2) For each $j \in [K]$, let $T_j = (A, \pi_\Gamma, \pi_\Delta, \Phi)$, where we have dropped subscripting according to j for notational simplicity:

a) For $k \in [\kappa]$, train predictor $\Phi_k = (\Gamma^{(k)}, \Delta^{(k)}, \phi_{\theta^{(k)}}, \boldsymbol{v}_{\text{default}}^{(k)})$ using the procedure in Section IV-B on the weighted experience samples $\mathcal{E}_{Z_j}$, where we choose $\Gamma^{(k)} = \Delta^{(k)}$ to be the list of references with k-th highest weight according to $\pi_\Gamma = \pi_\Delta$.

b) Update $\pi_\Gamma = \pi_\Delta$ by redistributing weight among the top κ shells according to a voting procedure where each

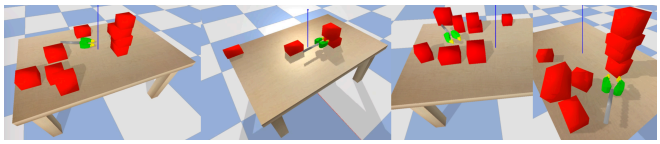

Fig. 3. Representative problem instances sampled from the domain.

training sample "votes" for the shell whose predictor minimizes the validation loss for that sample. In other words, the i-th experience sample $\mathcal{E}^{(i)}$ votes for mixture rule $v(i) = k$ for $k = \arg\min_{k \in [\kappa]} \mathcal{L}(\Phi_k; \mathcal{D}, \mathcal{E}^{(i)})$. Then, shell weights are assigned to be proportional to the sum of the sample weights (i.e., membership probability of belonging to this rule) of samples that voted for each particular shell: the number of votes received by the k-th shell is $V(k) = \sum_{i=1}^{|\mathcal{E}|} \mathbb{1}_{v(i)=k} \cdot z_{i,j}$, for indicator function $\mathbb{1}$ and $k \in [\kappa]$. Then, $\pi_\Gamma(k)$, the current k-th highest value of π_Γ, is updated to become $V(k)/\xi$, where ξ is a normalization factor to ensure that π_Γ remains a valid probability distribution. (Specifically, $\xi = (\sum_{k=1}^{\kappa} \pi_\Gamma(k))/(\sum_{k=1}^{\kappa} V(k))$.)

c) Repeat Step 2a, in case the κ shells with highest π_Γ values have changed, in preparation for using the mixture rule to make predictions in the next step.

3) Update membership probabilities by scaling by data likelihoods from using each of the K rules to make predictions: $z_{i,j} = z_{i,j} \cdot \exp(-\mathcal{L}(T_j; \mathcal{D}, \mathcal{E}^{(i)}))/\zeta$, where $\exp(-\mathcal{L}(T_j; \mathcal{D}, \mathcal{E}^{(i)}))$ is the data likelihood from using mixture rule T_j to make predictions for the i-th experience sample $\mathcal{E}^{(i)}$, and $\zeta = \sum_{j=1}^{K} z_{i,j} \cdot \exp(-\mathcal{L}(T_j; \mathcal{D}, \mathcal{E}^{(i)}))$ is a normalization factor to maintain $\sum_{j=1}^{K} z_{i,j} = 1$.

4) Repeat Steps 2 and 3 some fixed number of times, or until convergence.

V. Experiments

We apply our approach, SPARE, to a challenging problem of predicting pushing stacks of blocks on a cluttered table top. The object universe Υ is composed of blocks of different sizes and weight, the property set $\mathcal{P}$ includes shapes of the blocks (width, length, height) and the 3-D position of the block. We have one action template, *push*(α, o), which pushes toward a *target object* o with parameters $\alpha \in \mathbb{R}^4$ We simulate this 3D domain using the physically realistic PyBullet [14] simulator with Gaussian noise on the action parameters. We consider the following deictic references in the reference collection $\mathcal{F}$: (1) *above* (O), which returns the objects immediately above O; (2) *above** (O), which returns all of the objects that are above O; (3) *below* (O), which returns the objects immediately below O; (4) *nearest* (O), which returns the object that is closest to O.

We first compare our single-rule learning approach to the baseline, a neural network function approximator that takes in as input the current state and action parameter, and outputs the predicted Gaussian distribution of the next state. In each problem instance, a robot pushes a block with the presence of other blocks (Fig. 3). Figure 4(a) shows the performance, as measured by the log data likelihood, as a function of

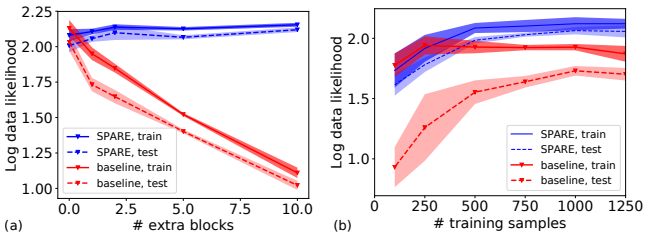

Fig. 4. (a) Comparing performance as a function of number of distractors with a fixed amount of training data. (b) Comparing sample efficiency of SPARE to the baseline. Shaded regions represent 95% confidence interval.

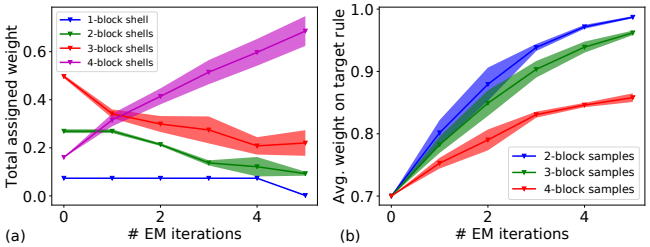

Fig. 5. (a) Shell weights per iteration of our EM-like algorithm. (b) Membership probabilities of training samples per iteration.

the number of extra blocks when a stack of 3 blocks is pushed. As more objects are added to the table, baseline performance drops as the presence of these additional objects appear to complicate the scene and the baseline is forced to consider more objects when making its predictions. However, performance of the SPARE approach remains unchanged, as deictic references are used to select just the three blocks in the stack as input in all cases, regardless of the number of extra blocks on the table. Fig. 4(b) plots the data likelihood as a function of the number of training samples. Both our approach and the baseline benefit from having more training samples, but our approach is much more sample efficient and achieves good performance within only a thousand training samples.

Now we put our approach in a more general setting where multiple transition rules need to be learned for prediction of the next state. Our approach adopts an EM-like procedure to assign each training sample its distribution on the transition rules and learn each transition rule with re-weighted training samples. First, we construct a training dataset and 70% of it is on pushing 4-block stack. Our EM approach is able to concentrate to the 4-block case as shown in Fig. 5(a).

Fig. 5(b) tracks the assignment of samples to rules over the same five runs of our EM procedure. The three curves correspond to the three stack heights in the original dataset, and each shows the average weight assigned to the "target" rule among samples of that stack height, where the target rule is the one that starts with a high concentration of samples of that particular height. At iteration 0, we see that the rules were initialized such that samples were assigned 70% probability of belonging to specific rules, based on stack height. As the algorithm progresses, the samples separate further, suggesting that the algorithm is able to separate samples into the correct groups.

REFERENCES

[1] Z. Wang, S. Jegelka, L. P. Kaelbling, and T. Lozano-Pérez, "Focused model-learning and planning for non-gaussian continuous state-action systems," in *ICRA*, 2017.

[2] P. E. Agre and D. Chapman, "Pengi: An implementation of a theory of activity." in *AAAI*, vol. 87, 1987, pp. 286–272.

[3] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling, "Learning symbolic models of stochastic domains," *Journal of Artificial Intelligence Research*, vol. 29, pp. 309–352, 2007.

[4] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[5] E. Amir and A. Chang, "Learning partially observable deterministic action models," *Journal of Artificial Intelligence Research*, vol. 33, pp. 349–402, 2008.

[6] H. H. Zhuo, Q. Yang, D. H. Hu, and L. Li, "Learning complex action models with quantifiers and logical implications," *Artificial Intelligence*, vol. 174, no. 18, pp. 1540–1569, 2010.

[7] K. Mourao, R. P. Petrick, and M. Steedman, "Using kernel perceptrons to learn action effects for planning," in *International Conference on Cognitive Systems (CogSys 2008)*, 2008, pp. 45–50.

[8] G. L. Drescher, *Made-up minds: a constructivist approach to artificial intelligence*. MIT press, 1991.

[9] M. P. Holmes and C. L. I. Jr., "Schema learning: Experience-based construction of predictive action models," in *Advances in Neural Information Processing Systems*, 2005, pp. 585–592.

[10] T. Oates and P. R. Cohen, "Searching for planning operators with context-dependent and probabilistic effects," in *AAAI/IAAI, Vol. 1*, 1996, pp. 863–868.

[11] E. Şahin, M. Çakmak, M. R. Doğar, E. Uğur, and G. Üçoluk, "To afford or not to afford: A new formalization of affordances toward affordance-based robot control," *Adaptive Behavior*, vol. 15, no. 4, pp. 447–472, 2007.

[12] L. Montesano, M. Lopes, A. Bernardino, and J. Santos-Victor, "Learning object affordances: from sensory–motor coordination to imitation," *IEEE Transactions on Robotics*, vol. 24, no. 1, pp. 15–26, 2008.

[13] K. Chua, R. Calandra, and S. Levine, "On the importance of uncertainty for control with deep dynamics models," in *NIPS Workshop on Acting and Interacting in the Real World*, 2017.

[14] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2018.