NEUROSYMBOLIC REPRESENTATIONS FOR LIFELONG LEARNING

by

Alper Ahmetoğlu

B.S., Computer Engineering, Boğaziçi University, 2017

M.S., Computer Engineering, Boğaziçi University, 2019

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering
Boğaziçi University
2024

NEUROSYMBOLIC REPRESENTATIONS FOR LIFELONG LEARNING

APPROVED BY:

Assoc. Prof. Emre Uğur (Thesis Supervisor)	. . . .
Prof. Erhan Öztop (Thesis Co-supervisor)	. . . .
Prof. Lale Akarun	. . . .
Assist. Prof. İnci M. Baytaş	. . . .
Prof. Esra Erdem	. . . .
Prof. Sinan Kalkan	. . . .

DATE OF APPROVAL: 23.01.2024

to Çağla.

ACKNOWLEDGEMENTS

Not everyone has two great advisors! I would like to thank Emre Uğur and Erhan Öztop for their guidance and support throughout my PhD years. Thanks to them, I got proactive roles in many aspects of research, which significantly reshaped me from a simple student into a (hopefully) proper researcher. Thanks to them, I worked on a subject that I am deeply interested in. Thanks to them, I now look at problems from a wider perspective. Thanks to them, I have always looked forward to our discussion sessions. Thanks to them, I got the opportunity to study in Japan. Thanks to them, I connected with people. Thanks to them, ...

I would like to thank Arzucan Özgür, Esra Erdem, İnci Baytaş, Lale Akarun, and Sinan Kalkan for accepting to be a part of my PhD journey through the qualification exam, thesis progress, and defense. Their comments were always constructive and helpful. I am also grateful to Justus Piater, Minoru Asada, and Yukie Nagai for their comments and suggestions. Special thanks to Suzan Üsküdarlı, who was always supportive and constructive.

I also thank Ahmet Tekden, Yiğit Yıldırım, Muhammet Hatipoğlu, Utku Türk, and Tuluhan Akbulut for the joyful discussions.

I am grateful to my family and in-laws for their unconditional support. Not everyone is lucky to have two families strongly supporting their research. Lastly, this thesis would not be possible without my beloved Çağla, without her support, without her guidance, and without her joy.

This thesis was partially supported by TUBITAK ARDEB 1001 (120E274), International Joint Research Promotion Program, Osaka University, under the project "Developmentally and biologically realistic modeling of perspective invariant action understanding", the Japan Society for the Promotion of Science, Grant-in-Aid for

Scientific Research (22H03670), the project JPNP16007 commissioned by the New Energy and Industrial Technology Development Organization (NEDO), and JST, CREST (JP-MJCR17A4). The numerical calculations reported in this thesis were partially performed at TUBITAK ULAKBIM, High Performance and Grid Computing Center (TRUBA resources).

ABSTRACT

NEUROSYMBOLIC REPRESENTATIONS FOR LIFELONG LEARNING

This thesis presents a novel framework for robot learning that combines the advantages of deep neural architectures in processing high-dimensional vectors with classical AI search techniques to bridge the gap between continuous sensorimotor data of the robot and domains consisting of finite entities. The aim is to convert information about the environment collected through interactions into an appropriate symbolic form on which a search tree can be built to reach a desired state. The framework consists of an encoder-decoder type of network with binarized activations in the bottleneck layer. The state of the environment, represented as a set of object features, is given to the encoder as input. The output is a discrete vector, treated as the object's symbol, given to the decoder together with the action vector. The decoder predicts the effect observed by the agent due to the executed action. Once the network is trained, we can transform the continuously represented environment definition into symbolic vectors using the encoder. This allows us to build rules defining the transitions in the environment defined over these symbols. These rules can be translated into planning domain definition language (PDDL), allowing domain-independent off-the-shelf planners to be used to search for a goal state. Our experiments on tabletop object manipulation setups show that the system can learn appropriate symbols of the environment that allow it to build object towers with desired heights and complex object structures that require modeling the relations between objects by reasoning through the rules defined over the symbols learned in an unsupervised manner. As the framework is built with differentiable blocks, it affords appending recent advances in deep learning with ease, allowing it to be extensible in multiple directions.

ÖZET

HAYAT BOYU ÖĞRENME İÇİN NÖROSEMBOLİK GÖSTERİMLER

Bu tez, derin sinir ağı mimarilerinin yüksek boyutlu vektörleri işlemedeki avantajları ile klasik yapay zeka arama tekniklerini birleştirerek robot öğrenimi için yeni bir yöntem sunmaktadır. Bu yöntem, robotun sürekli gösterimdeki duyudevinimsel verileri ile sonlu sayıda nesneler içeren ortamlar arasında bir köprü kurmak için tasarlanmıştır. Önerilen yöntemin amacı ortam ile etkileşimler yoluyla toplanan verileri uygun sembolik yapılara çevirerek bu semboller üzerinden ağaç arama yöntemleri ile istenilen bir hedef durumu bulmaktır. Yöntemin genel yapısında gizyazıcı-gizçözücü tipinde bir sinir ağı bulunmaktadır. Sinir ağının darboğaz katmanında türev akışına izin veren ikili etkinleştirme hücreleri bulunmaktadır. Ortamdaki nesnelerin öznitelikleri ile temsil edilen ortam durumu, gizyazıcıya girdi olarak verilmektedir. Gizyazıcı her nesne için ayrık bir vektör üretir, ve bu vektörler nesnelerin sembolleri olarak kullanılır. Nesne sembolleri eylem vektörü ile birlikte gizçözücüye girdi olarak verilir, ve gizçözücü robotun eyleminin yol açtığı etkiyi tahmin eder. Tüm yapı eğitildikten sonra sürekli bir şekilde gösterilen ortamın tanımı gizyazıcı kullanılarak sembolik hale dönüştürülebilir. Bu sayede ortamdaki geçişler semboller üzerinden kurallar tanımlanarak gösterilebilir. Bu kurallar planlama alan tanım diline çevrildiğinde çeşitli planlama yöntemleri kullanılarak hedef bir durum aranabilir. Masaüstü nesne etkileşimi deney düzeneklerinde yaptığımız deneyler bu sistemin ortama dair uygun semboller öğrendiğini göstermiştir. Gözetimsiz bir şekilde öğrenilmiş bu semboller üzerinden tanımlanan kurallar kullanılarak çeşitli yüksekliklerde nesne kuleleri ve nesneler arası ilişkilerin modellenmesini gerektiren karmaşık nesne yapıları kurulmuştur. Bu yöntem türevlenebilir yapı taşları ile kurulduğu için derin öğrenmedeki yenilikler ile çeşitli yönlerde genişletilebilir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$\mathcal{A}$	Action set
$\mathbb{B}^d$	Set of d-dimensional binary vectors
$\mathbb{E}$	Expectation
$\mathbb{R}^d$	Set of d-dimensional real numbers
x	Real-valued vector
z	Discrete-valued (symbolic) vector
$\mathcal{Z}$	Set of symbolic vectors
δ	Cartesian difference
$\mathit{\Delta}$	Difference operator
σ	A binary function
$\mathit{\Sigma}$	A set of binary functions
ϕ	Symbolic operator
$\mathit{\Phi}$	Set of symbolic operators
$\|.\|_2$	Euclidean norm

LIST OF ACRONYMS/ABBREVIATIONS

2D	2-dimensional
AI	Artificial Intelligence
ASP	Answer Set Programming
CNN	Convolutional Neural Network
DCGAN	Deep Convolutional Generative Adversarial Network
DNN	Deep Neural Network
DQN	Deep Q-Network
GS	Gumbel-Sigmoid
IID	Independently and Identically Distributed
LLM	Large Language Model
MLP	Multi-Layer Perceptron
MSE	Mean-Squared Error
OBO	Object-Binary-Object
OCEC	Object-Continuous-Effect and Clustering
PCA	Principal Component Analysis
PDDL	Planning Domain Definition Language
PPDDL	Probabilistic Planning Domain Definition Language
ReLU	Rectified Linear Unit
RGB	Red Green Blue
RL	Reinforcement Learning
SFA	Slow Feature Analysis
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine

1. INTRODUCTION

The ultimate aim of artificial intelligence (AI) research is to create an agent that perceives its environment through its sensors and acts on it to solve a desired task with little or no human intervention. As the definition of a task might not be available a priori, or can change over time, an intelligent agent must have a method to learn the necessary *abstractions* to be able to carry out a given task. Therefore, learning appropriate abstractions for the task at hand has been a well-studied topic in AI.

The trending methodology in the last decade for learning abstractions for a specific task is optimizing the parameters of a differentiable function, such as a deep neural network (DNN), with variants of stochastic gradient descent (SGD). While the foundations of this strategy go well before the last decade, advances in the initialization techniques [1–4], adaptive optimization methods [5–7], normalization methods [8–10], and dedicated hardware for fast matrix multiplication combined with the availability of large data sets [11, 12] resulted in a step change in the performance in many domains including object detection [13–16], object segmentation [17–19], machine translation [20, 21], and game playing [22–25].

Even more recently, the introduction of the transformer architecture [26], a deep neural network with a self-attention module for modeling the interactions between input parts, enabled the training of very large DNNs [21], [27–33], with parameter sizes in billions, on internet-scale text data sets. These large networks, trained with the simple objective of predicting the next token given previous ones, called large language models (LLMs), can solve not only a single task but many tasks that are defined in natural language. LLMs showcased that a foundational model [34] trained on a very large data set with a generic objective can be fine-tuned to solve different downstream tasks, which shifted the current paradigm on solving the general problem of AI into training large networks with large data sets. The main motivation is that if we can learn generic *distributed representations*—multiple units responsible for the response to an input—that summarize the training data so well, then we can use them in many scenarios instead of learning an abstraction for each problem separately.

Even with billions of connections and internet-scale data sets, these models can perform poorly on simple questions that require several steps of reasoning [35]. It can be argued that this is caused by two fundamental problems: (1) neither the network architecture nor the training schema has an inductive bias toward any kind of abstract reasoning, and (2) the model, without any embodiment, is not trained in the real world but on a reflection of it described by humans with human symbols. Of the two, the latter is a serious problem as collecting a large data set of robot interactions in the wild, in the same amplitude with internet-scale text data with camera image as input and motor commands as output might not be feasible in the foreseeable future.

The current LLMs can be considered as instantiations of the Chinese Room Argument [36]—a thought experiment questioning whether an agent that has access to a virtually infinite set of input-output pairs can be considered intelligent. Harnad argues that a solution to the Chinese Room Argument is to learn the necessary symbols in a grounded, bottom-up fashion where symbols bind to sensory data [37]. Harnad's proposal can be interpreted as a neurosymbolic solution in which connectionist models are used as a tool to learn 'iconic representations' on which one can build higher-level symbolic representations that can be used for reasoning.

On the other hand, in robot learning, the main challenge in training an agent in the real world is not reasoning but rather perceiving the environment through data received from sensors and translating it into an appropriate task space. Once we convert sensorimotor data represented with continuous-valued vectors into symbolic entities in the task space, the problem almost becomes trivial. From the example in Konidaris [38], consider the task of teaching a robot how to play chess in the real world with a physical chessboard and chess pieces. Once the robot recognizes the pieces, converts them into symbols, and learns the transitions between these symbols (which can be thought of as high-level actions or skills), even a simple tree search algorithm such as A* with a simple heuristic would be sufficient to achieve the rating of a local club player. It is the abstraction of the sensory data—the representational gap between sensory data and task-level symbolic entities—that poses most of the challenge, and it is a type of problem that the deep learning methodology excels at. This is one of the core motivations that this thesis followed: combining deep architectures with classical AI search techniques to create a general recipe for scalable robot learning.

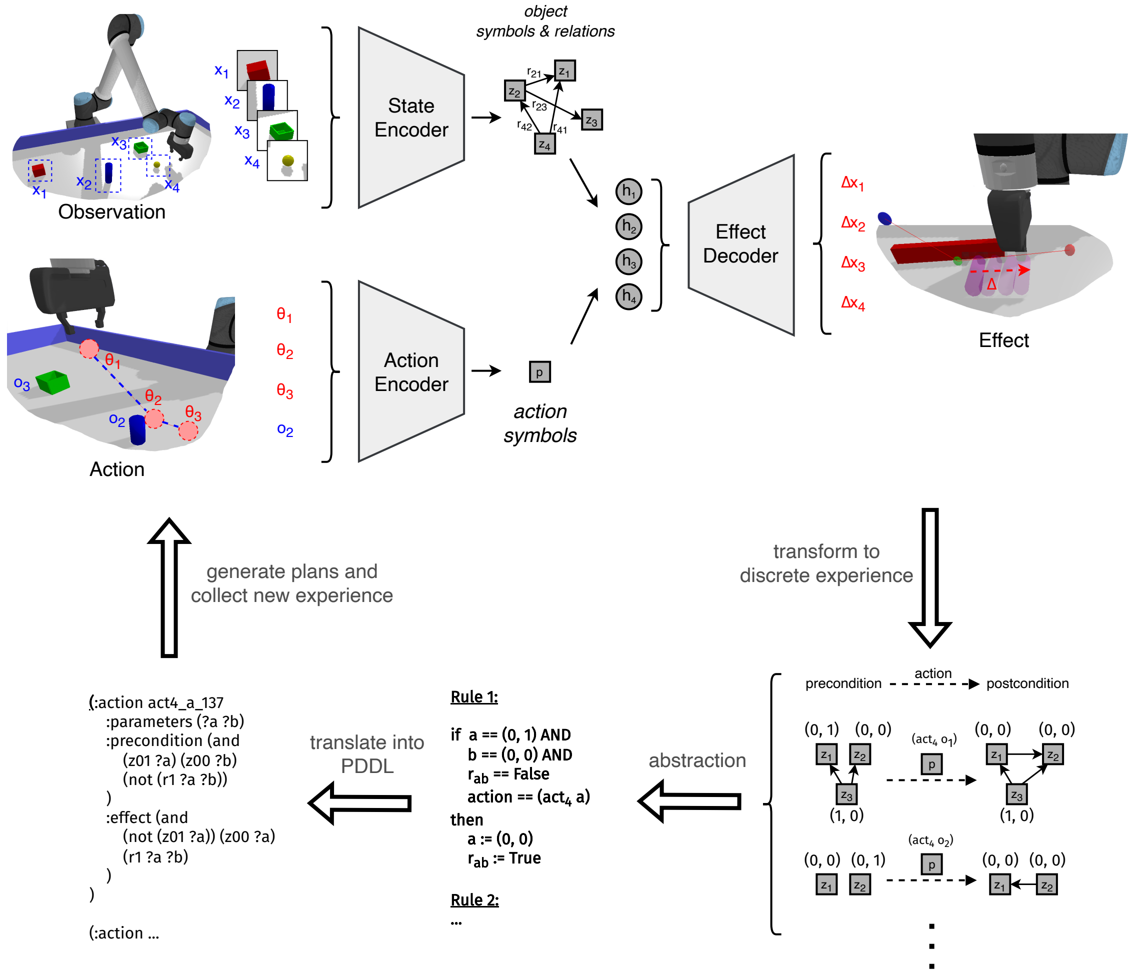
What can be a generic training objective, like the next token prediction for the text domain, to learn the necessary abstractions for a given task in robot learning in a bottom-up manner? In Konidaris *et al.* [39], it is proven that given an agent with a set of actions, it is necessary and sufficient to learn abstractions for pre- and post-conditions (or effects) of actions to enable domain-independent planning using tree search algorithms. Following the chess example, this would be equivalent to learning the rules of chess from images taken from a top-down camera by using the interaction data of the robot executing legal chess moves—or watching people playing chess. After learning abstractions to represent the environment symbolically, the robot can arrive at a desired board configuration by searching for the goal state in the symbolic space with a tree search algorithm.

While learning abstractions for pre- and post-conditions of actions might be a good strategy for robot learning as it equips the agent with a tool to predict what would happen if an action is taken, it would be an incremental change from the Chinese Room Argument if no assumption is made on the form of the learned abstractions. Some works in the reinforcement learning literature propose to learn a model of the world—a function that predicts the next state given the current state and the action—and use this model to maximize a reward function [40, 41]. Even though these works learn abstractions that allow us to predict the future, there is no clear evidence whether the learned representations can be manipulated, as in Harnad's proposal [37], to account for novel situations. In other words, even though the representations learned with these approaches are grounded in sensory data, they cannot be manipulated symbolically to create new representations that account for new situations.

On the other hand, when we consider the tasks that humans do in their daily lives, most of them consist of a finite number of entities, objects, and their relations that together compose a scene, and actions we take manipulate only a subset of this scene. For example, when we are making a cup of instant coffee, we open the cupboard, take the coffee can and a cup, open the coffee can, put a scoop of coffee into the cup, and pour down hot water from the kettle. It is, most of the time, object properties and relations that change due to our actions. As a corollary, preconditions and effects of actions that we take at each step depend only on a subset of objects—whether we have a cup or not does not affect our ability to open the coffee can.

Following this observation, learning object-oriented and effect-based abstractions [42–44] might be a reasonable and generic approach for robot learning, which can also be regarded as learning *affordances* of objects—action possibilities that the object offers to the agent. This definition essentially combines information from the features of the object (i.e., preconditions), actions that the agent possesses, and the resulting effect after executing the action on the object. The result is a clear objective: 'which actions can be applied to which objects, and what happens if applied,' which has a resemblance to the next token prediction in current LLMs. Representing objects by their affordances enables immediate transfer capability to novel objects since the robot can apply actions and observe their effects to assess the affordances of new objects or use previously learned affordances while developing new skills [45, 46]. In a sense, affordances provide a simple language defined by skills to represent objects.

A generally capable embodied agent that we target (e.g., consider an animal) should be able to interact in different environments and novel situations. This asserts that the learned representations should be adaptable—or transferrable—to new tasks to minimize learning. Utilizing from previously acquired knowledge to solve a new task is a well-studied topic in machine learning, commonly referred to as transfer learning [47], meta-learning [48], or lifelong learning [49], each of which treats the same problem from different perspectives. The lifelong learning problem of an embodied agent is essentially different from transfer learning—in the context of machine learning—since the agent, and therefore the actions that can be executed by the agent, is directly shared between tasks. As such, learning a new skill to solve a specific task directly affects the agent's ability to solve other tasks and, in turn, to collect qualitatively different data to learn new representations. For an agent that is operating in our everyday environments (e.g., consider a cleaning robot in the living room), learning object-level affordances creates a common substrate for different tasks as most of the tasks consider manipulating objects in the environment. Affordances, that are action possibilities offered by the environment to the agent, connect the agent's capabilities with its surroundings. This sets object-level, affordance-encoding, and symbolic representations as a good candidate for lifelong learning.



```
(:action act4_a_137
   :parameters (?a ?b)
   :precondition (and
      (z01 ?a) (z00 ?b)
      (not (r1 ?a ?b))
   )
   :effect (and
      (not (z01 ?a)) (z00 ?a)
      (r1 ?a ?b)
   )
)

(:action ...
```

```
Rule 1:

if  a == (0, 1) AND
    b == (0, 0) AND
    r_ab == False
    action == (act_4 a)
then
    a := (0, 0)
    r_ab := True

Rule 2:
...
```

Figure 1.1. The general outline of the proposed framework. The robot interacts with the environment and collects state–action–effect tuples. These tuples are used to train a neural network that learns object symbols and their relations with each other.

This thesis provides a new robot learning framework by combining three main motivations: (1) learning object-level symbolic representations (2) that would encode preconditions and effects of actions (3) with deep neural architectures that can be used for domain-independent planning. The general outline of the approach is shown in Figure 1.1. The proposal is a general design strategy that can be adapted to different situations by modifying the neural architecture. Essentially, we build an encoder-decoder network with discrete activations in the bottleneck layer (without breaking differentiability) to learn object and action symbols that predict the effect of the executed action in an end-to-end manner. After training these networks with the interaction data that the robot randomly executes in the environment, we convert the continuously represented experience of the robot (i.e., state vectors) into discrete transitions from which one can distill abstract rules about the environment. In the end, such abstract rules allows using tree search methods to search for an action sequence that reaches a target state.

The journey in this thesis starts off with a search for high-level, task-specific units in a neural network trained to solve a task with a reinforcement learning (RL) algorithm. The hypothesis is that if we can fish such units out with a simple method, then we can build a lifelong learning pipeline by building on top of previously learned (and selected) units. Given a neural network trained on a primary task with deep Q-network (DQN) algorithm [22, 23], we applied principal component analysis (PCA) for the minimum information loss, and slow feature analysis (SFA) for creating signals that change slowly on top of the last layer activations of the network. We compared units created by SFA and PCA with plain activations on a related but novel secondary task. In this new task, we train a new network from scratch while using units from SFA, PCA, or no transformation. Our results showed that units created by SFA are the most successful for skill transfer. SFA, as well as PCA, incur fewer resources compared to usual skill transfer where full layer outputs are used in the new task learning, whereby many units formed show a localized response reflecting end-effector-obstacle-goal relations in these tasks. This work, which is detailed in Chapter 3, is published in *Advanced Robotics* [50].

Even though PCA and SFA transformations on top of the last layer outputs provide task-specific activations, these are not very high-level abstractions that can be used for planning and search. In the next study, we instead focused on building an architecture with a constraint such that the learned abstractions are, by design, discrete, symbolic representations. This work planted the seeds for the general architecture in Figure 1.1. Namely, we built an encoder-decoder network with a discrete bottleneck layer that predicts the effect of the robot's executed action. The encoder takes the depth image of the object as input and outputs a binarized vector with Gumbel-sigmoid function [51, 52] for error backpropagation. These outputs, which can be thought of as object symbols encoding affordances, are given to the decoder together with the action vector (which is assumed to be given in this work, i.e., there is no action encoder) for the effect prediction. After training the network, we translate the state–action–effect tuple into its discrete counterpart and train a decision tree with the discrete data set. The motivation is to extract probabilistic rules from branches of the tree that encode discrete state–action–effect tuples. Lastly, we convert these rules into probabilistic planning domain definition language (probabilistic PDDL, PPDDL), used by the planning community to represent domains in a standardized language and show planning examples where the robot stacks cubes to build towers with a desired height. This work, DeepSym, constitutes the basis of the thesis, detailed in Chapter 4, and is published in *Journal of Artificial Intelligence Research* [46].

Experiments in DeepSym consider acting on a single object with no other objects in the environment. This is hardly the case in the real world, as objects are related to each other, and an action applied to an object might affect other objects. Consider a plate containing a fork and a knife. If the robot carries the plate to the kitchen, the fork and the knife are also carried with it. As such, we need a modification on the architecture to handle the effects of a varying number of objects. In this regard, we proposed Attentive DeepSym [53] incorporating self-attention layers [26] to model the relations of objects with each other. Here, the encoder takes a varying number of objects as input (in the batch dimension) and produces object symbols. Combined with the action vector (still fixed), these symbols propagate information to each other via the self-attention module to produce multi-object representations that are used by the decoder for effect prediction. As effects change with relations of objects (i.e., two objects move together if one is on top of the other), these multi-object representations implicitly hold relational information between objects. Attentive DeepSym [53], detailed in Chapter 5, is presented at *Signal Processing and Communications Applications Conference 2023* with its Turkish translation [54].

Attentive DeepSym provided a way to learn relational information between objects. However, these relations remain implicit in the weights of the self-attention module. In other words, even though the whole architecture can make accurate effect predictions when objects are in relation with each other, it does not output—and we do not have a simple way to extract—relations between objects, which makes it hard to convert the state information into rules that domain-independent planners can reason with. Following the previous example, the model does not output that there is a relation between the plate and the fork, and while creating the rule set, we have no way to determine whether, e.g., a cup on the table, should be included in the rule when the robot is interacting with the plate. As our problem was not having access to the learned relations, in Relational DeepSym [55], we explicitly output binary object relations from input object features using a modified self-attention module with Gumbel-sigmoid function, instead of softmax, as the attention activation. These binary relation weights are then used to aggregate object symbols together with the action vector to be given to the decoder for effect prediction. Now, this system does not only output object symbols but also relations between objects from state–action–effect tuple. This gives a more appropriate set of symbols to represent the current state of the environment: object symbols and their relations with each other. Relational DeepSym [55], detailed in Chapter 6, compares relational formulation with Attentive DeepSym and Vanilla DeepSym in terms of effect prediction performance. This work is accepted and will appear in *IEEE Robotics and Automation Letters.*

As previous chapters focus on learning object symbols and relations while using pre-defined actions for interacting with the environment, in Chapter 7, we focus on learning high-level skills from a continuously parameterized action space by extending Relational DeepSym with an action encoder. We show that with this architectural extension, high-level skills can be distilled from the action encoder after training. We show an example of how these skills can be used in further stages of learning.

In DeepSym, rules are extracted from a decision tree trained on the symbolic state–action–effect data. When there are multiple object symbols and relations that are permutation invariant, training a decision tree becomes non-trivial as there is no fixed dimension. As an alternative, in our next work, we build the rule set by partitioning samples with the same abstract preconditions and actions, converting them into PDDL, and generating plans that involve the affordances of an arbitrary number of objects to achieve tasks. This work [56], detailed in Chapter 8, is submitted to *International Conference on Robotics and Automation 2024*.

We go over the important aspects of the proposed methodology in Chapter 9, discuss the initial assumptions that are addressed in this thesis, and provide promising and feasible future directions that would increase the applicability of the method for more generic scenarios. We give a summary and a conclusion of the thesis in Chapter 10.

Now, in the next chapter, we will go over the related works that study learning symbolic representations from continuous data for the purpose of planning in the context of robot learning.

2. RELATED WORK

Bridging the representational gap between the continuous sensorimotor world of a robotic system with the discrete symbols and rules has been a key research goal from the early days of intelligent robotics [57, 58]. While grounding predefined symbols in the sensorimotor experience of the robot has been widely used for intelligent robot control [59–63], some argue that symbols “are not formed in isolation”, and “they are formed in relation to the experience of agents” [64]. We share this viewpoint that has been investigated in a number of studies. Some studies realized systems that clustered low-level sensory experience into categories and performed subsymbolic planning in the continuous perceptual space [65], [66]. While simple planning capability was achieved, the use of continuous prediction and state transition operators limited the use of powerful off-the-shelf symbolic AI planners. In another line of research, Ozturkcu *et al.* [67] asked whether there are any symbols formed in a deep RL agent after training the agent for a given task without imposing any prior on the architecture or the objective.

Recently proposed hybrid approaches exploit the prior domain knowledge by combining non-monotonic logical reasoning with deep networks [68, 69]. In these architecture, there is a cascade of two models where the first model is the prior domain knowledge encoded as an Answer Set Programming (ASP) program [70], and the second model is a convolutional neural network (CNN). If the ASP program fails to classify an example, it redirects the necessary parts of the input to CNN for further processing. This pipeline results in better accuracy with less computation when compared with CNN classification. Furthermore, given labeled examples about the task, the ASP program can be further extended to include new rules about the environment by using the decision paths of a trained decision tree. These works primarily focus on integrating neural models with common-sense knowledge or domain knowledge to increase performance. Our work is similar to these works in the sense that they also learn previously unknown rules with decision trees from subsymbolic data that would help in planning.

On the other hand, we focus on learning symbols from the interaction data that the robot collects by executing random actions; learning symbols that depend on the action set of the agent.

The bottom-up generation of symbolic structures from the continuous interaction experience of a robot has started to draw attention in robotics [38], [71]. Konidaris *et al.* [39], [72] studied the construction of symbols that are directly used as preconditions and effects of actions for the generation of deterministic and probabilistic plans in 2D agent settings, and later extended the framework into a real-world robot setting [73]. However, these studies use a global state representation, and therefore, symbols learned in an environment cannot be used directly in a novel environment. In follow-up works, James *et al.* [44], [74] construct symbols with egocentric and object-centric representations to allow the transfer of previously learned symbols to new environments. These studies train an SVM classifier for each effect cluster to find groundings of precondition symbols. Ugur *et al.* [42], [45] formed symbols used in plan generation in manipulation using a combination of several ad-hoc machine learning techniques such as clustering with X-means [75] and classification with SVMs. Furthermore, they used hand-crafted features to represent scenes and effects. On the other hand, our proposed architecture simultaneously learns object categories (in the encoder output) and their corresponding effect categories (in the decoder output) without resorting to any clustering techniques on the object or effect space. The object and effect categories automatically emerge as the network with binary bottleneck units minimizes the effect prediction error. Furthermore, deep neural networks allow us to process high-dimensional image data efficiently using convolutional layers. This design approach offers a generic symbol formation engine that runs at the pixel level using deep neural networks. In terms of symbol multiplicity, our approach is more parsimonious, as we do not form symbols for each action as in [45] and [73]; but instead, we use a single decoder network that takes the action as part of the input. To be concrete, for n effect categories and k actions, our system generates nk symbols, whereas the aforementioned approaches generate n^k symbols. Learning a single model for all actions possibly allows internal representations learned for one action to be re-used directly for other actions. Another significant advantage of our model is that it is differentiable and thus can be integrated into gradient-based state-of-the-art machine learning architectures for further tackling more complex problems.

Asai and Fukunaga [76] realized a similar neural framework where they first train a state autoencoder with discrete latent units, then learn the action precondition-effect mappings. Follow-up works combine these two steps and learn the action mapping together with the state auto-encoder [77, 78]. These works are in the visual domain (for example, 2D puzzles) and achieve visualized plan execution while we focus on robot action planning and execution in the 3D world. Moreover, a critical difference of our method from the aforementioned work is that we learn object symbols by taking into account action and the effects in addition to object features, which facilitates the formation of symbols that are likely to capture object affordances [79, 80].

Another line of research focuses on bilevel planning, in which a symbolic plan is complemented by a motion and task planner [81, 82]. In these works, operators are learned for bilevel planning when given parameterized policies for continuous planning. A follow-up work learns these parameterized policies as well, completing the whole neurosymbolic planning pipeline [83]. While these works fix the state abstractions, Silver *et al.* [84, 85] also learn new state abstractions with a surrogate objective for planning. Achterhold *et al.* [86] learns policies parameterized by neural networks based on the symbolic state transitions. Quite similar to the rule learning procedure in Chapter 8, Kumar *et al.* [87] learn a set of operators by considering only a subset of abstract effects to prevent learning complex operators. The main difference between our work and theirs is that we select this subset as the relevant symbols for the action, whereas they consider universal quantifiers in the abstract effect. Another similar work trains a network that outputs relations between objects from RGB images given objects' canonical images [88].

3. HIGH-LEVEL FEATURES FOR RESOURCE ECONOMY AND FAST LEARNING IN SKILL TRANSFER

3.1. Introduction

Earlier robotic work indicates that units that capture high-level features are potential candidates to facilitate effective knowledge transfer [42]. From a neural computation point of view, this also makes sense as compact high-level representations would be more economical to process and pass around in the neural circuits of biological systems. Thus, the brain might adopt a strategy to represent sensorimotor information compactly with minimum information loss, in line with the information compression ideas [89, 90]. Another hypothesis can be obtained by generalizing the idea that slowly changing features in sensory data tend to correspond to more high-level concepts [91] to neural responses. According to this view, the brain might seek to exploit those neurons that are more stable over the others that show frequent changes. For example, when a hand waving action is observed, the earlier sites in the visual processing pathway (e.g., areas V1, V2) would show temporally changing activity, whereas at the end of the processing pipeline (i.e., in area IT), the recognition of the hand waving action would be represented with a few neurons, which would show stable activity during the most of the observation period over a variety hand waving actions.

Motivated by these hypotheses, in this chapter, we analyze different ways of transferring previously learned representations for a new task in an economical way. More specifically, we consider two methods: (1) principal component analysis (PCA) for the minimum information loss, and (2) slow feature analysis (SFA) [91] for creating signals that change slowly. PCA reduces the dimensionality of the data while preserving the information maximally, which makes it a suitable candidate for transferring compact features. On the other hand, SFA, which is not very well explored in the transfer learning context, creates slowly changing representations that are arguably more robust because objects of interest in the real world do not make abrupt changes, considering the notion of object permanency. In our experiments, we first train a deep Q-network [22] on a task where the goal is to move the robot arm to a desired target position in the presence of a rectangular obstacle that may appear at different locations. After training, we create separate skill transfer scenarios in which either PCA or SFA transformations are applied on the last hidden layer activations of the network and used in new task learning by augmenting the features found in the last hidden layer of the new network (see Figure 3.1). We compare these scenarios with the baselines of skill transfer with full-layer output and no-transfer scenarios. Our experimental results show that:

- Using features that are constructed with SFA is not only more economical in terms of the number of units but also better for skill transfer than the naive approach of using the full set of layer activations.
- PCA is also helpful for resource economy in skill transfer but not as good as SFA in terms of the success rate of the new task.
- SFA and PCA capture interpretable high-level features such as joint angles, tip locations, and the distance from the tip position to the goal position solely from the activation history of the network.

In the rest of this chapter, we detail our methodology in Section 3.2, define the experimental setup in Section 3.3, give the results in Section 3.4, and conclude in Section 3.5.

3.2. Methods

Suppose that an agent has developed a neural system to solve a specific task. When the agent encounters a similar task, neurons in this system will respond to the sensorimotor input, albeit possibly in a different way since the input will be different. Even though the system does not directly provide the appropriate motor control output, it would be economically viable to use a previously learned network instead of relearning everything from scratch. We focus on extracting features from this network in an economical way. Note that it is not important how this network has been formed; it might have been learned via stochastic gradient descent, pre-wired, or obtained through an evolutionary algorithm. The hypothesis is that on each case, the network should have developed task-specific responses in order to successfully solve the task.

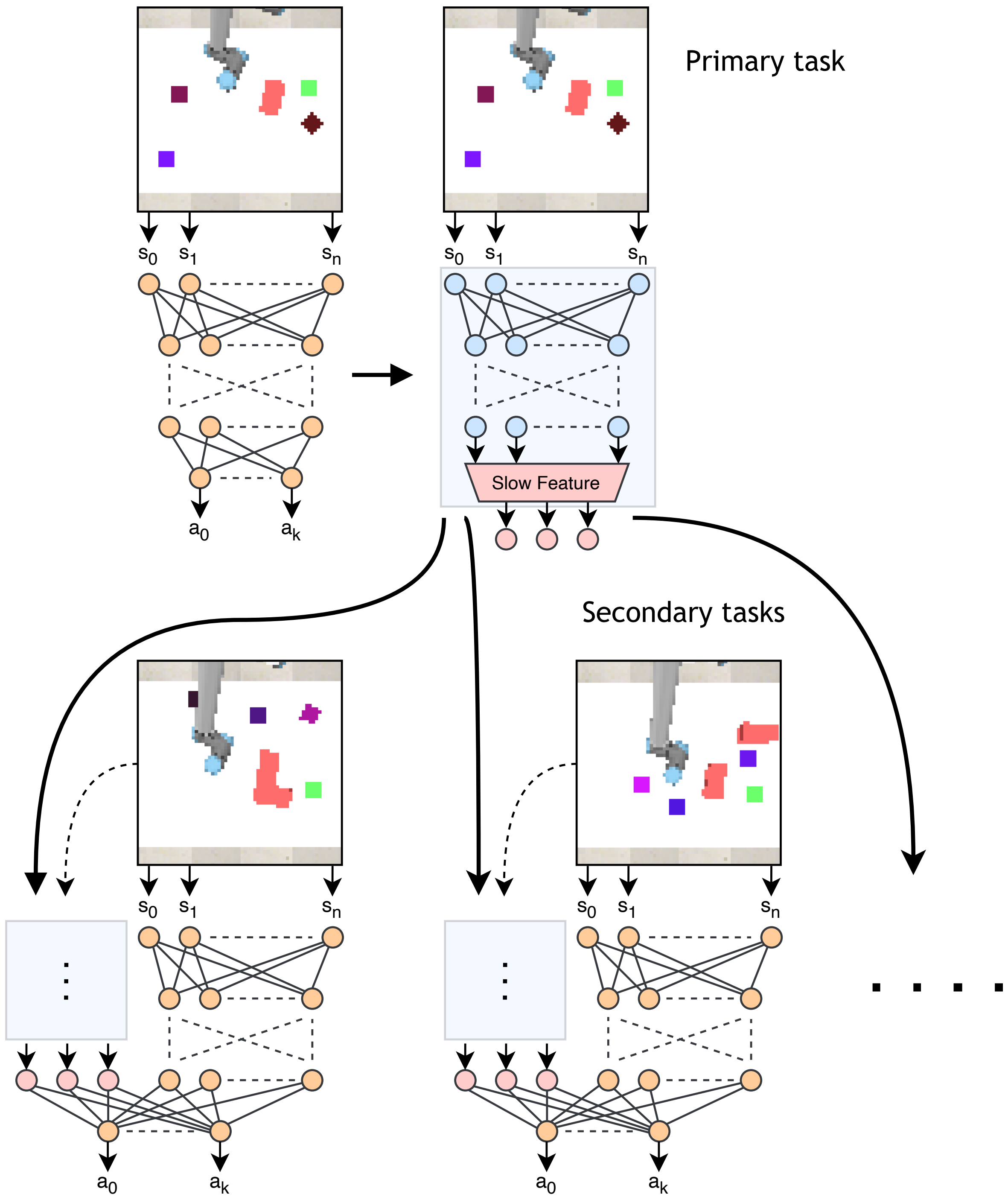


Figure 3.1. The general outline of the transfer method with slow features.

Firstly, we train a deep Q-learning agent [22] on a primary task to have a network which we can extract features from. In the primary task, a robot arm tries to move its arm to the goal position while avoiding a rectangular obstacle. After training, we extract features from this network in three different ways and use them in a secondary task to assess the bootstrapping effect induced for the new task learning. As a baseline we also learn the new task from scratch without any transfer. So, overall, we have these cases:

(i) Learning from scratch (`transfer:none`). The new task is trained without any transfer to form a baseline for the next three transfer scenarios.

(ii) The last hidden layer activations (`transfer:full`). This is one of the most frequently used methods for transferring visual features from networks that are trained on large-scale data sets [92] and serves as a reference for the next two methods.

(iii) Transforming the last hidden layer with PCA (`transfer:PCA`). While the last layer of activations provides useful information, it is mostly distributed in many units. PCA is especially useful for concentrating the distributed information into fewer units, effectively reducing the number of neurons, thus reducing the number of weights that needs to be optimized for learning the new task. Let us denote our dataset as $X = \{H_i\}_{i=1}^{N}$ where N is the number of robot movement trajectories arising from executing the primary task of the robot, and H_i is an $T_i \times D$ matrix where T_i is the number of timesteps in ith trajectory and D is the dimensionality of the last hidden layer. We concatenate each H_i from the first dimension so that X is a $(T_1 + T_2 + \cdots + T_N) \times D$ matrix. We want to find a projection $z = Xw$ such that $\mathrm{Var}(z)$ is maximized. This leads to the objective,

$$\arg\max_{w} \quad \frac{w^T X^T X w}{w^T w}, \tag{3.1}$$

which is the Rayleigh quotient, and the expression takes its maximum value when w is equal to the eigenvector with the largest eigenvalue of $X^T X$ [93].

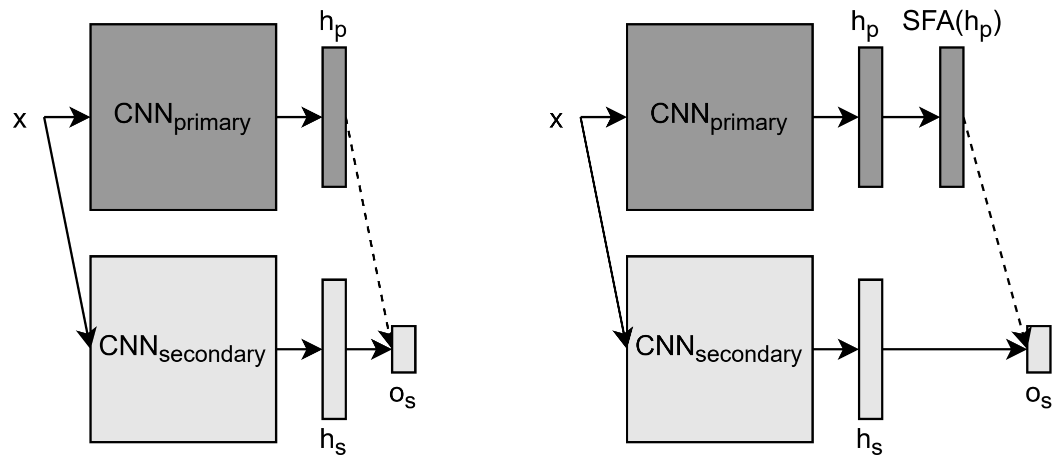
(iv) Transforming the last hidden layer with SFA (`transfer:SFA`). PCA puts an emphasis on the maximum information-preserving units. On the other hand, SFA tries to minimize the time derivative of the output signals. More specifically, SFA creates output features $z = Xw$ with the following objective and restrictions [91]:

$$\min \quad \mathbb{E}[\dot{z}_i^2] \tag{3.2}$$

subject to

$$\begin{aligned} \mathbb{E}[z_i] &= 0, & (3.3) \\ \mathbb{E}[z_i^2] &= 1, & (3.4) \\ \mathbb{E}[z_i z_j] &= 0 \quad \forall j < i. & (3.5) \end{aligned}$$

Here, Equations (3.3) and (3.4) prevent the trivial solution of a constant feature and also force the output to be normalized. Equation (3.5) forces features to be orthogonal to each other. The solution can be found by first whitening the data and then applying PCA to the time derivative of X. This method is shown to be useful for extracting the independent factors of an input signal.

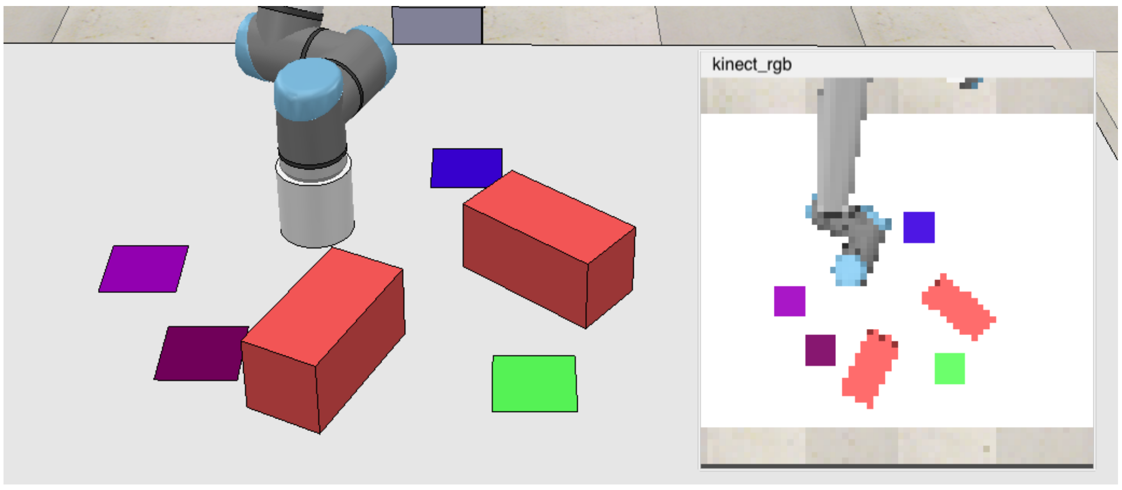


(a) (b)

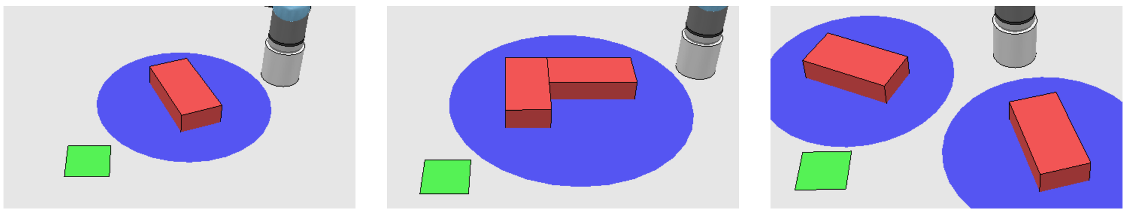
Figure 3.2. Different transfer methods are depicted. (a) `transfer:full`. (b) `transfer:sfa`. `transfer:pca` is identical to (b) except that the transformation is done with PCA instead of SFA.

From a network trained on a primary task, we extract the information implicitly represented in the weights with these three methods to use it on a new task by concatenating the features to the input of the last layer, as shown in Figure 3.2. In our experiments, we define the secondary task to be a reinforcement learning setup as well.

The environment is similar to the primary task, except that there are different types of obstacles to avoid.



(a)



(b) (c) (d)

Figure 3.3. The input from the camera is shown on the top-right inset. The regions of penalty around obstacles are visualized in blue (see text). (a) The experiment setup. (b) A rectangular obstacle. (c) L-shaped obstacle. (d) Two obstacles.

Note that both PCA and SFA define an affine transformation of the last layer and, thus, do not bring any advantage in terms of function complexity since two linear transformations can be combined into a single one. However, re-alignment of the feature space can greatly accelerate the speed of convergence with stochastic gradient descent [94]. Likewise, we expect that agents that use these transformations will exhibit a better performance with fewer samples. In the next section, we detail the experimental setup and the results of our experiments.

3.3. Experiment Setup

We perform our experiments on CoppeliaSim 4.2.0 simulator [95]. The experiment setup consists of a UR10 [96] robot arm with a solid cylinder attached to its end effector and a camera placed 180cm over the table for a top-down visual perception. The robot arm can make planar movements in eight different uniform directions in a 80cm $\times$ 100cm rectangular workspace. The camera input provides a 64 $\times$ 64 pixels colored image at each timestep (Figure 3.3). In the environment, there are red-colored rectangular and L-shaped obstacles, a green goal marker, and distractor markers that dynamically change colors in the RGB range of $(0, 0, 0) - (255, 0, 255)$. The general overview of the setup is shown in Figure 3.3.

In this environment, we define two types of tasks: a primary task and a secondary task. We train a network to solve the primary task and then try to transfer the knowledge inside the network to the secondary task using different methods. In the primary task, the robot tries to move its end-effector to the goal position while avoiding a rectangular obstacle. There are two secondary tasks that differ by their obstacle types. In the first one, there is an L-shaped obstacle instead of a rectangular one, and in the second one, there are two rectangular obstacles. Secondary tasks are deliberately selected to be a linear increment from the first one to see if there are high-level features in the network, such as an 'obstacle detector'. If there are such high-level units, then the superposition of two obstacles would correspond to a new one; thus, a transfer would increase the speed dramatically. In both primary and secondary tasks, we included distractors that change colors and move in random directions with a constant speed to increase the difficulty of the tasks.

The reward function for both environments is defined as

$$R(t) = \begin{cases} 10 \quad \text{if} \quad \|x_{\text{tip}}(t) - x_{\text{goal}}\|_2 < 5\text{cm} \\ 10\mathit{\Delta}_t(x_{\text{tip}}, x_{\text{goal}}) \quad \text{else if} \quad \|x_{\text{tip}}(t) - x_{\text{obstacle}}(t)\|_2 > \tau \\ 10(\mathit{\Delta}_t(x_{\text{tip}}, x_{\text{goal}}) - \text{relu}(\mathit{\Delta}_t(x_{\text{tip}}, x_{\text{obstacle}}))) \quad \text{otherwise,} \end{cases} \tag{3.6}$$

where τ is a threshold for obstacle proximity penalty and Δ_t denotes the change in the distance between the robot end-effector and the target over consecutive time steps:

$$\Delta_t(x, y) = \|x(t-1) - y(t-1)\|_2 - \|x(t) - y(t)\|_2. \tag{3.7}$$

We set τ to 21cm for the primary environment and 28cm for the secondary environments. The robot gets a penalty whenever it goes towards the obstacle when it is in the blue region visualized in Figure 3.3.

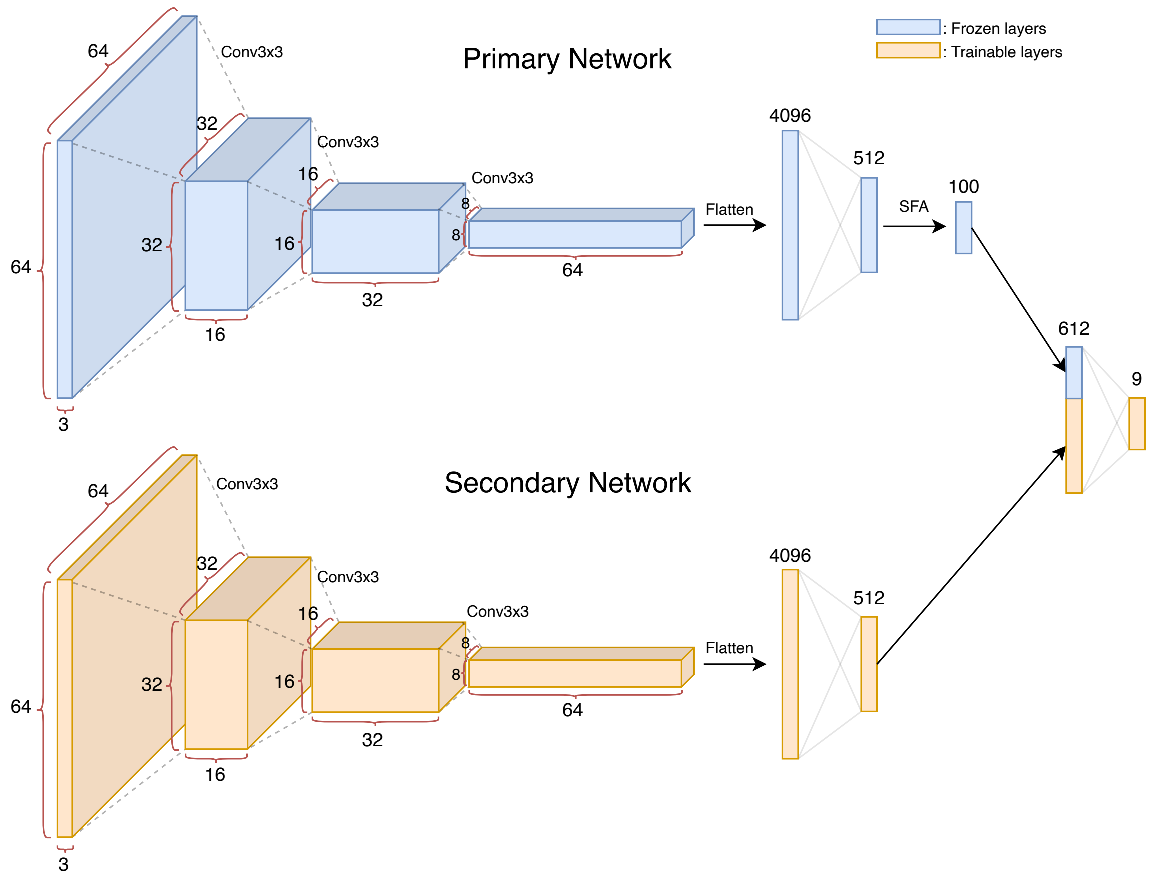


Figure 3.4. The transfer network architecture for the `transfer:sfa` method. `transfer:pca` is the same, except there is a PCA transformation instead of SFA. Conv3x3 are convolutional layers with 3×3 sized kernels.

We use a convolutional neural network (CNN) architecture, where three convolutional layers followed by two fully connected layers are employed in each architecture. Convolutional layers have 16, 32, and 64 channels successively. Each layer has a kernel size of 3×3 with a stride of two and a padding of one. Fully connected layers have 512–9 units. We concatenate the transferred features to the last hidden layer except the learning from scratch case. For example, if we transfer 100 slow features, the dimensionality of the last hidden layer becomes $512 + 100 = 612$. The outline of the architecture is shown in Figure 3.4. For PCA, we selected the first 100 components, which cover more than 99% of the variation in the data. Likewise, we selected the first 100 slow features for a fair comparison.

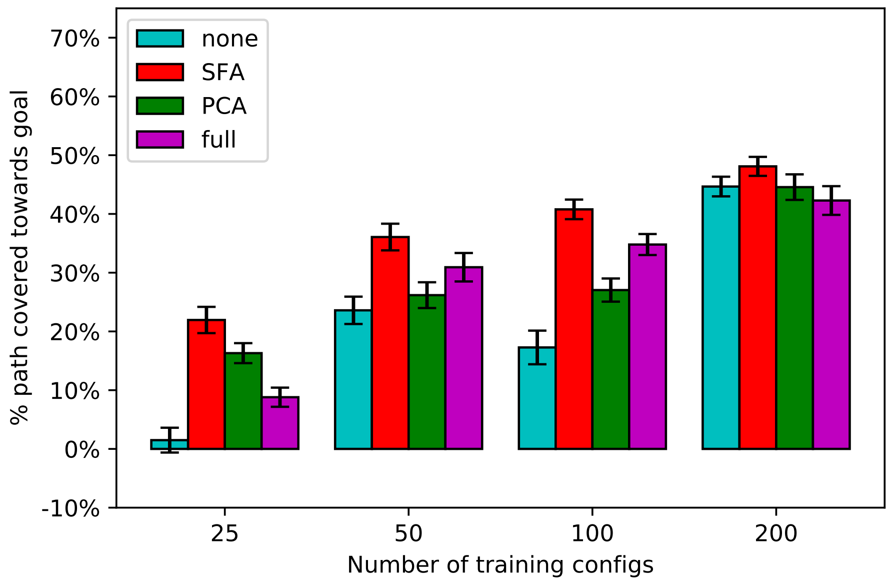
We train each model as a deep Q-network [22] for 2000 episodes with a replay buffer of size 50,000. Each episode lasts for at most 200 timesteps; after that, the episode terminates. We used the reward function defined in Equation (3.6). The last fully connected layer is used for the Q-value estimation. There are eight different actions for eight different directions and an additional action for no operation. For the optimization, we use Adam optimizer [7] with a learning rate of 0.001 with no learning rate decay.

3.4. Results

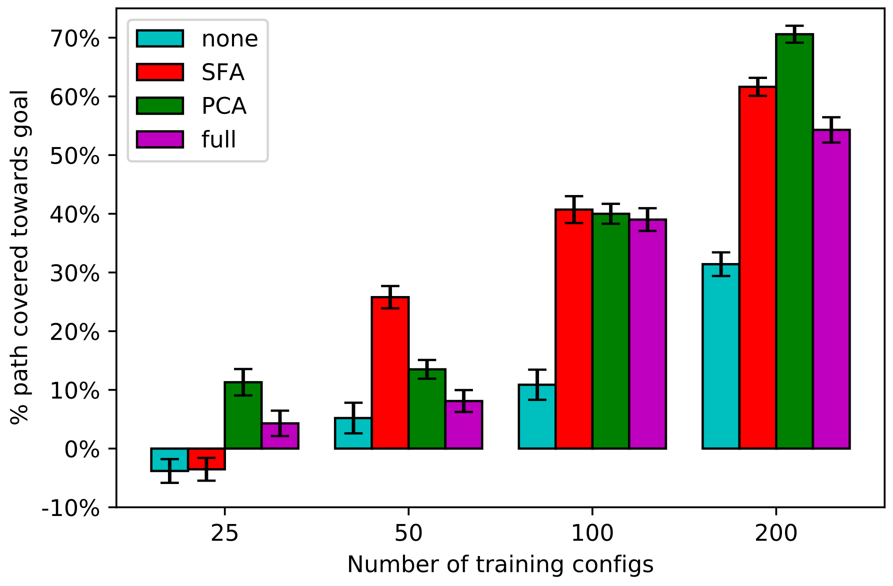
In this section, we first test the generalization performance attained by the transfer of features acquired with different methods to a new task in 3.4.1. Next, we analyze correlations between the proposed features and high-level task-related features in Section 3.4.2. Lastly, in Section 3.4.3, we visualize the responses of neurons for varying inputs to get an insight into their functionalities.

3.4.1. Transfer Performance

The aim of this experiment is to test the bootstrapping effect of using a previously learned representation on learning a similar task from scratch. To this end, we concatenate the activations of a previously trained network (described in the previous section) to the last hidden layer. This method is one of the basic transfer learning methods used in various applications [92]. We compare the transfer of plain activations, features created with PCA, and with SFA. In addition, we train a network from scratch with no transfer to assess whether transfer methods bring any advantage for learning the new task.



(a)



(b)

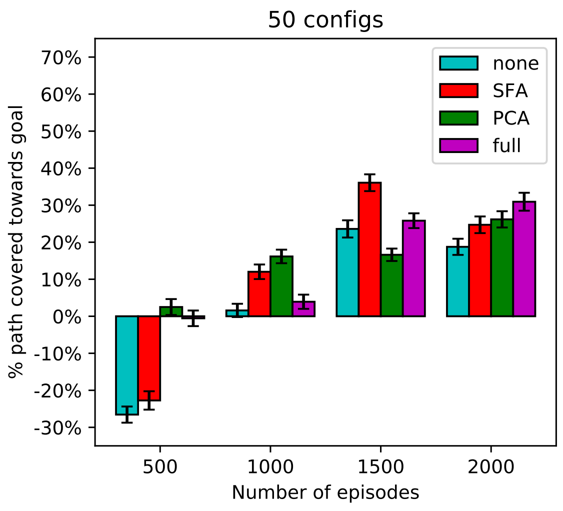
Figure 3.5. The generalization performance measured as distance covered towards the goal vs. number of configurations experienced during learning in the new task. A secondary environment with (a) an L-shaped obstacle, (b) two rectangular obstacles.

Each model is trained with 25, 50, 100, and 200 number of training configurations for 2000 episodes. Here, a training configuration refers to an environment setting (i.e., the initial position of the objects). Each episode is initialized with a configuration sampled from this set. After convergence, we test each model at every 500 episodes on the previously unseen environment settings (i.e., configurations) to observe the generalization performance. For testing, we collect 100 runs with each model and calculate the average percentage of path covered towards the goal, and treat this estimate as one test result. The percentage of the path covered toward the goal is calculated as

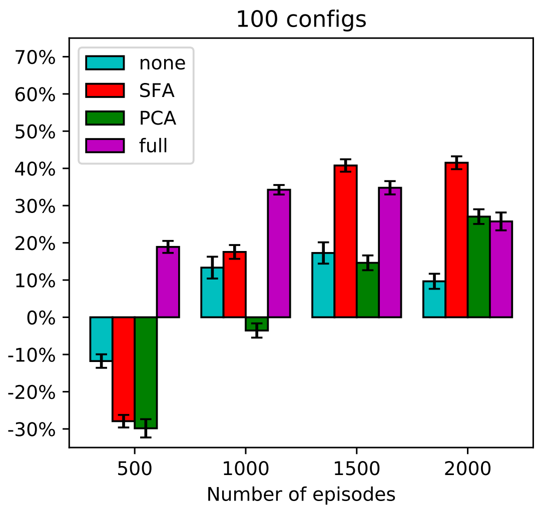
$$\frac{\text{initial distance} - \text{final distance}}{\text{initial distance}} \times 100. \tag{3.8}$$

Here, the distance is the Euclidean distance from the robot's tip position to the goal position. The average of 15 different test results at best-performing episodes (validated with five results) for each model is reported in Figure 3.5.

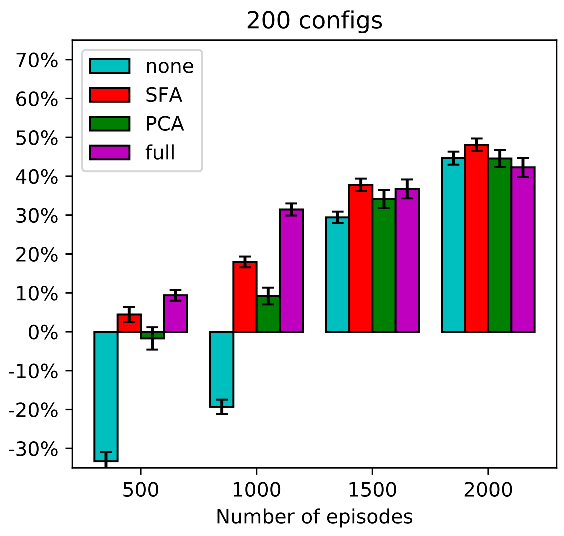
We first see that all transfer methods increase the performance even though they are more susceptible to overfitting with more parameters and fewer training configurations. This result suggests that there might be indeed high-level features in previously learned networks that help learn new skills from a small number of examples. We observe that using `transfer:SFA` gives better performance compared to `transfer:PCA`, `transfer:full`, and `transfer:none` except for 25 and 200 configurations in the secondary environment with two rectangular obstacles (Figure 3.5b). Welch's t-test is used to understand the significance of the results [97]. We use $p \leq 0.05$ as our threshold for significance. In Figure 3.5a for the environment with L-shaped obstacle, Welch's t-test shows a significant difference between `transfer:SFA` and `transfer:full` for 25, 100, and 200 configurations ($p = 0.0008$, $p = 0.022$, and $p = 0.041$, respectively), and almost significant difference for 50 configurations ($p = 0.082$). For two rectangular obstacles in Figure 3.5b, `transfer:SFA` is significantly better than `transfer:full` for 50 and 200 configurations ($p = 0.0001$ and $p = 0.0125$, respectively). The `transfer:PCA` method also has a competitive performance with less number of units when compared to `transfer:full`. Note that SFA and PCA features are 100-dimensional vectors, while plain activations are 512-dimensional vectors. These results show that SFA indeed creates more condensed features that are appropriate for skill transfer. The bootstrapping effect fades away when we increase the number of training configurations to 200 configurations. This is an expected result since every model gets better when we increase the variation in the training set. However, the usage of low training configurations is desirable in many real-world settings. On the other hand, there is still a performance gain even for 200 configurations for two rectangular obstacles.



(a)

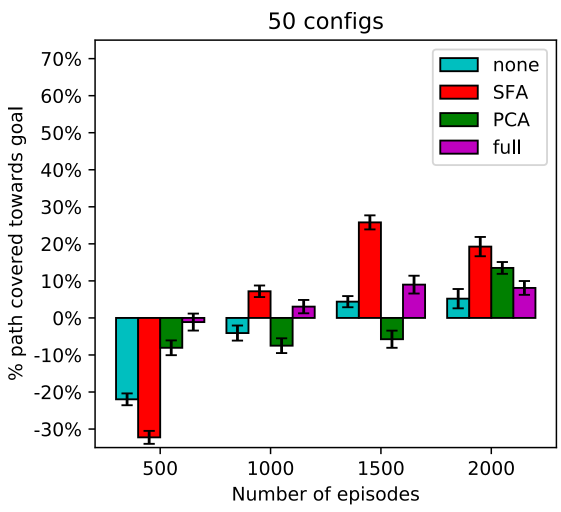


(b)

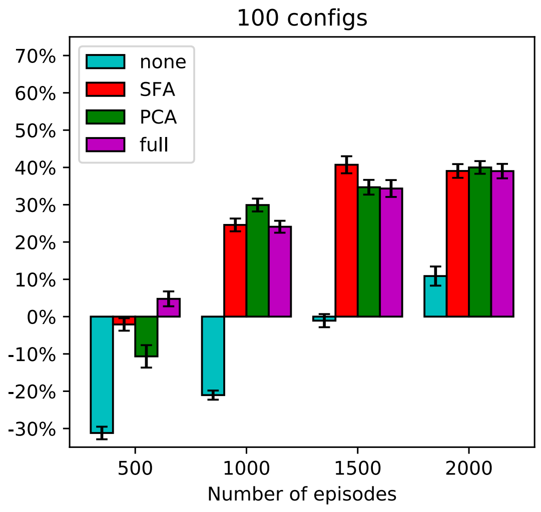


(c)

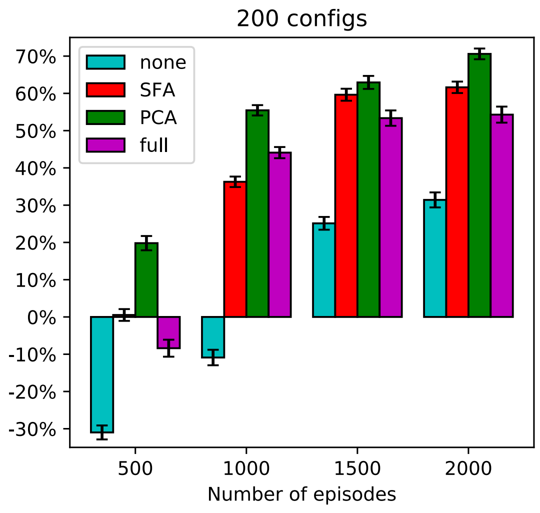
Figure 3.6. Test performance for different episodes with different methods and configuration numbers in the secondary environment with the L-shaped obstacle. (a) 50 configurations, (b) 100 configurations, and (c) 200 configurations.



(a)



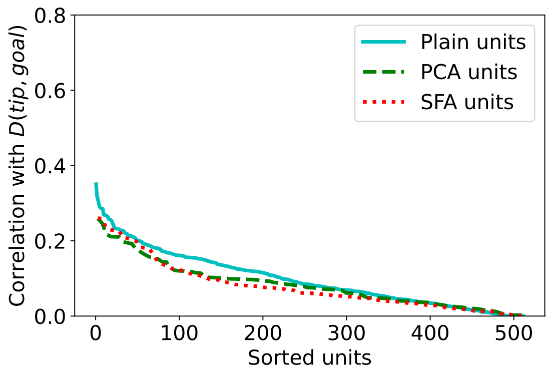
(b)



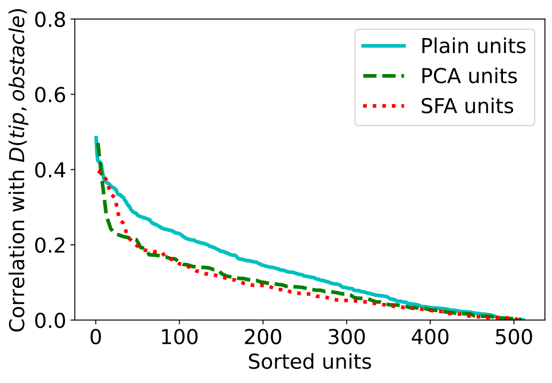
(c)

Figure 3.7. Test performance vs. number of training episodes for different methods and configuration numbers in the secondary environment with two rectangular obstacles. (a) 50 configurations, (b) 100 configurations, and (c) 200 configurations.

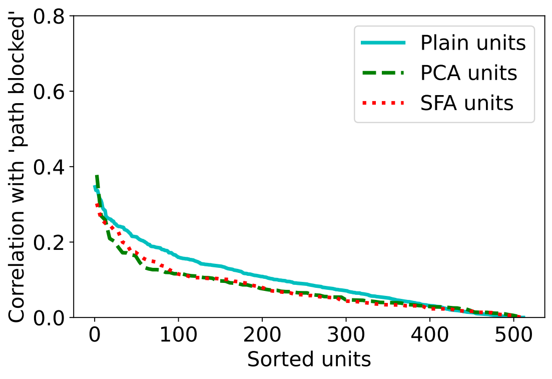
We also report the test results at different episodes in Figures 3.6 and 3.7. Figures suggest that using an additional set of features from a previously learned task helps generalization, especially when there are fewer number of configurations. We see that `transfer:sfa` performs on par with `transfer:none` in the initial stages of the training, then achieves the peak of its performance after 1500 episodes, which is generally less than other methods. On the other hand, the contribution of `transfer:full` steadily increases. This might be due to the distribution of information in the neurons. Only a few SFA units contain useful information for the task, and when these neurons are discovered with gradient descent in later steps, the performance increases rapidly. Moreover, `transfer:sfa` learns faster compared to other methods for less number of configurations (reaching the peak around 1500 episodes for 50 and 100 configs in Figures 3.6 and 3.7) and also requires less parameter update compared to `transfer:full` as there are fewer units, which is an essential property for small, autonomous systems with no access to a graphics processing unit (GPU). See Section 3.4.4 for an empirical analysis regarding the computational needs for SFA.



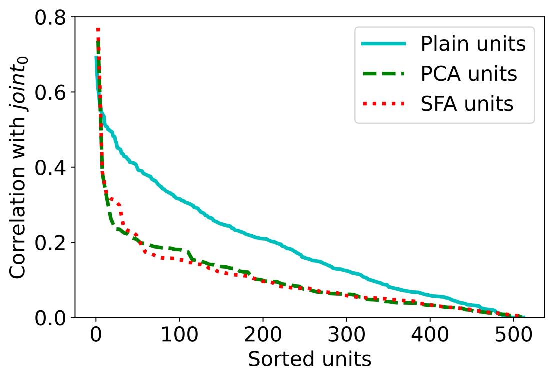
(a)



(b)



(c)



(d)

Figure 3.8. Units are sorted by their correlation with high-level features. Plain activations, SFA and PCA units are highlighted in blue, red and green, respectively. (a)-(c) High-level features, (d) Joint 0.

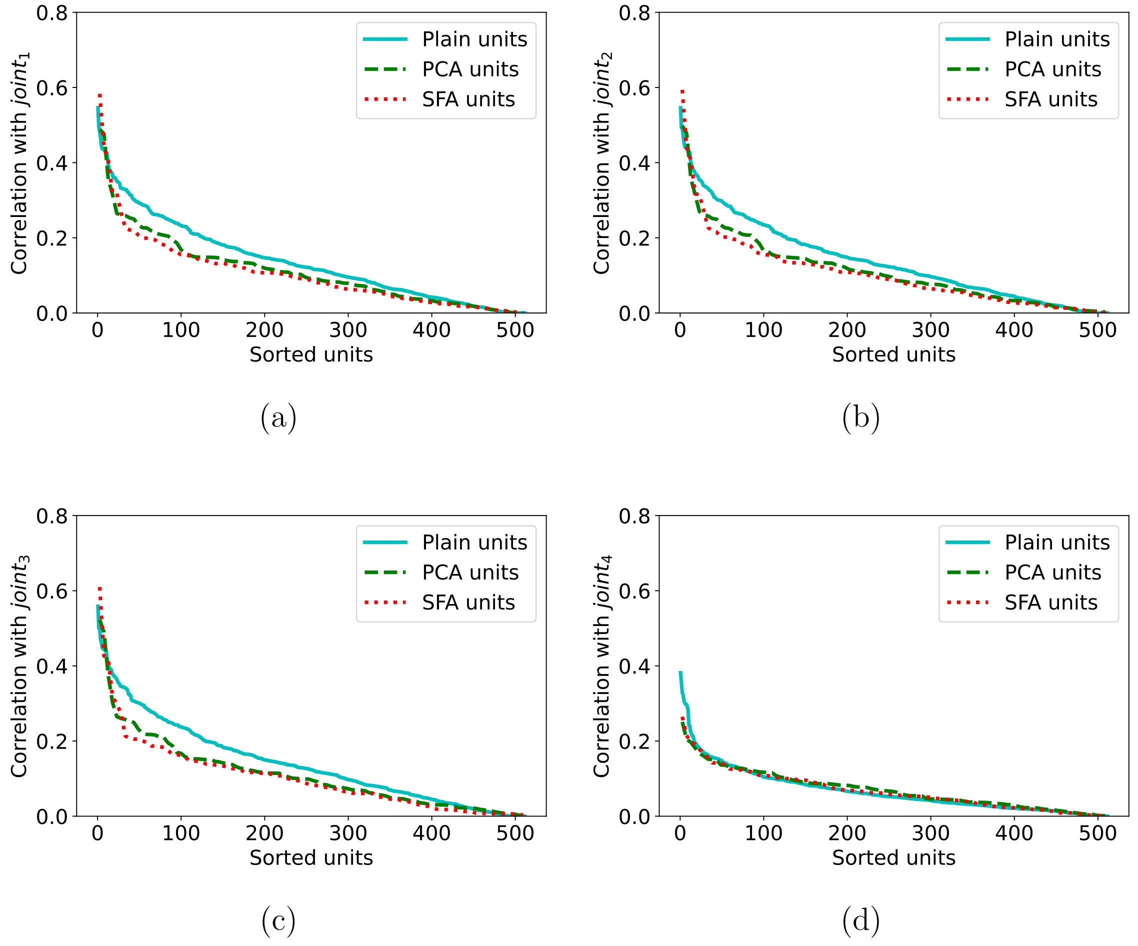


Figure 3.9. Units are sorted by their correlation. SFA and PCA units are highlighted in red and green, respectively. (a)-(d) Joints 1-4.

3.4.2. Correlation with High-Level Task Properties

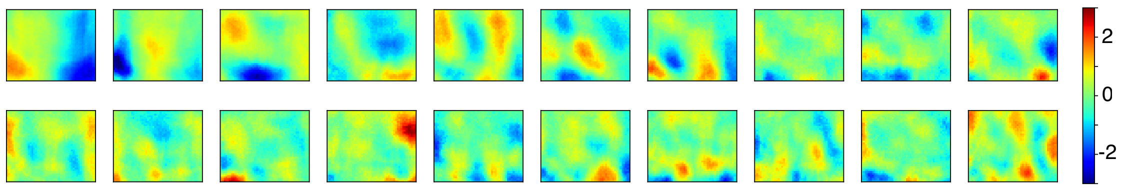
In this experiment, we investigate correlations between SFA/PCA units and high-level features. To this end, we freeze the network and run the policy for 100 episodes to collect hidden layer activations for PCA and SFA calculations. Then, we apply PCA and SFA with 100 components to the last hidden layer activations. To understand whether the transformed features capture any high-level features, such as relative goal position, obstacle position, and joint angles that are helpful to solve the task, we calculate the correlation between those high-level features and PCA/SFA features. We also compare the correlations between the last layer activations to see the effect of these transformations on the correlations.

In Figures 3.8 and 3.9, each subplot shows the distribution of correlations for each different high-level feature. Units are sorted by their correlation value for better visualization. After the transformation with PCA and SFA, most of the units become less correlated while only a few of them become highly correlated. This is probably due to PCA and SFA objectives forcing components to be decorrelated. From Figures 3.8 and 3.9, we can say that PCA and SFA capture the general neural response; they eliminate redundant neural responses and focus on the important ones by keeping the general response profile of the hidden layer. Out of the two, the SFA unit distribution has more units with high correlation, especially for joint angles. This might be one of the reasons that the learning performance of `transfer:sfa` is comparable to `transfer:full` and even better with fewer configurations since redundant and possibly noisy variations are filtered out by SFA.

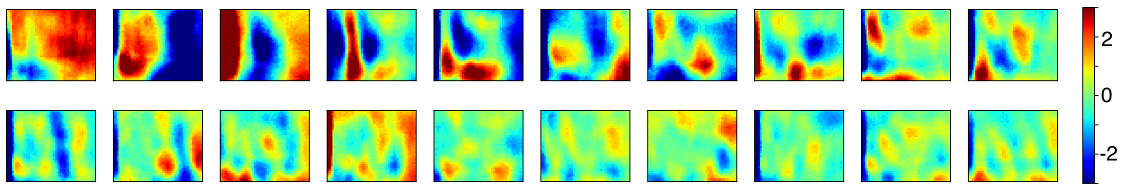
Table 3.1. Units that correlate the maximum with high-level features are reported. The numbers on the left and right denote the correlation and index, respectively.

High-level feature	Full activations	PCA features	SFA features
Joint 1	0.69 / 297	0.74 / 2	0.77 / 1
Joint 2	0.55 / 249	0.49 / 5	0.58 / 3
Joint 3	0.55 / 249	0.50 / 5	0.59 / 3
Joint 4	0.56 / 249	0.52 / 5	0.61 / 3
Joint 5	0.38 / 471	0.20 / 3	0.19 / 2
Joint 6	0.69 / 297	0.74 / 2	0.77 / 1
$D(x_{\text{tip}}, x_{\text{goal}})$	0.35 / 303	0.26 / 2	0.26 / 2
$D(x_{\text{tip}}, x_{\text{obstacle}})$	0.48 / 147	0.47 / 3	0.40 / 3
‘path blocked’	0.35 / 147	0.38 / 3	0.25 / 2

In Table 3.1, we select the most correlated units from each method and report their correlations. We see that all methods have a high correlation with joint angles and a slightly lower correlation with high-level features that relate the tip position to other objects. The high correlation with joint angles is an expected result, as the agent should know about the location of the arm in order to navigate it to the goal position. Since even for full activations we see a high correlation for task-level features, there might indeed be single, symbolic units specialized for a specific task. On the other hand, SFA and PCA further refine these features and create more compact representations.



(a)

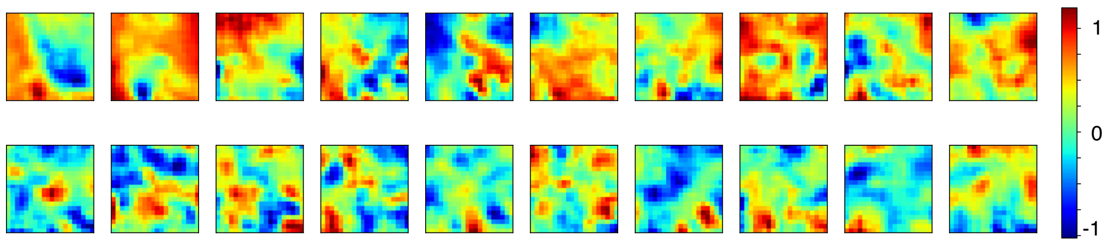


(b)

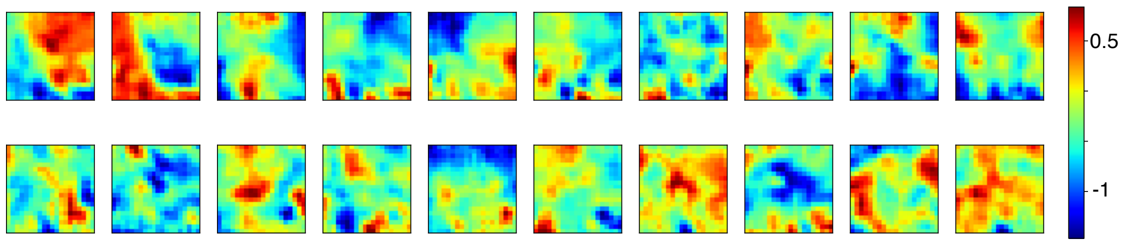
Figure 3.10. The first 20 (a) SFA and (b) PCA units' responses to tip positions. Units are sorted from left to right and top to bottom by their eigenvalues.

3.4.3. Visualizing Features

We created heatmaps for the responses of different SFA and PCA units to varying inputs. In Figure 3.10, each rectangle represents a unit's average response to different tip locations. For example, for the top left unit in Figure 3.10a, the response is high when the tip is located around the bottom left corner of the table, and it is low around the bottom right corner. To create these figures, we averaged out other variables (i.e., the goal position and the obstacle position). We see that both SFA and PCA units respond to blob-like regions of the table. The first few units are very compact and can be partially treated as symbolic representations (e.g., in Figure 3.10a, the first and the third units detect the position in the x-axis and y-axis, respectively.) As the unit number increases, these regions start to become scattered.

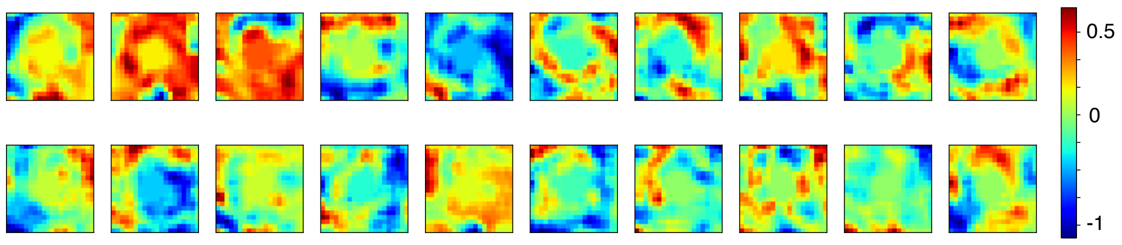


(a)

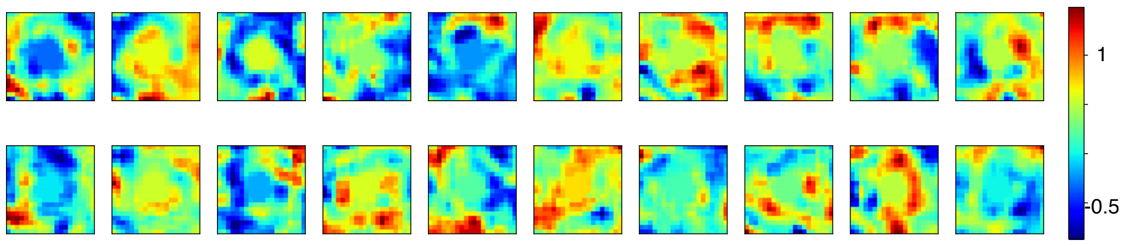


(b)

Figure 3.11. The first 20 (a) SFA and (b) PCA units' responses to the relative goal position. At the center, the distance is zero. Units are sorted from left to right and top to bottom by their eigenvalues.



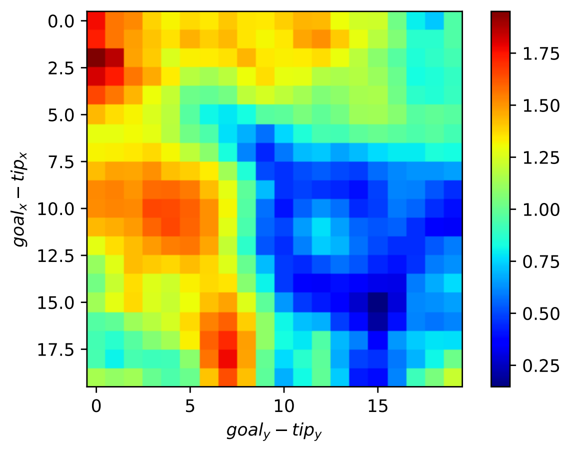
(a)



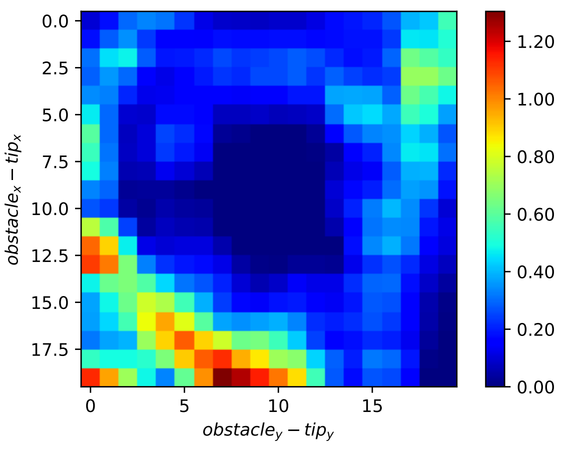
(b)

Figure 3.12. The first 20 (a) SFA and (b) PCA units' responses to the relative obstacle position. At the center, the distance is zero.

While the tip position is a piece of useful information to solve the task, the agent should also know the relative position of the goal and the obstacle with respect to its tip to successfully navigate in the environment. To understand the responses for relative distance to the goal and the obstacle, we created heatmaps with a similar procedure in Figures 3.11 and 3.12. The center point of each subplot represents the zero distance (i.e., $x_{\text{goal}} - x_{\text{tip}} = (0, 0)$). In Figure 3.11, most of the units are scattered except the first few ones. This figure suggests that both SFA and PCA are not very successful at covering the relative goal position compactly; the information is distributed into many units as in plain activations. However, in Figure 3.12, we see that both methods generate neuron responses that are inactive when the tip is close to the obstacle.



(a)



(b)

Figure 3.13. Neurons that correlate the most with (a) the relative goal position and (b) the relative obstacle position.

To understand whether these two high-level information are present in the network prior to transformation, we visualized the neurons that correlate the most with the relative goal position and the relative obstacle position in Figure 3.13. For the bottom half in Figure 3.11, we see that the most correlating neuron partially responds to the x-axis location of the relative goal position. On the other hand, in Figure 3.12, the response is similar to SFA and PCA units; it becomes inactive when the distance is below some threshold.

We conclude that even though the plain activations carry a distributed representation in general, they might represent some high-level information as a by-product that can be further processed and refined. We see that SFA and PCA transformations create high-level features to detect the tip position; however, this is not as compact as for the relative positions. Note that both SFA and PCA are unsupervised methods, not specifically tailored for creating more symbolic information. However, since they automatically create low-complexity signals in a principled way, they are good candidates for a more complex system.

3.4.4. Computation Time of Slow Features

The computation of slow features is a one-time process that depends on the singular value decomposition of the trajectory array. For 200 trajectories with a total of 20961 timesteps, the computation of slow features takes 0.36 seconds on an Intel i7-8700K CPU. The only computational overhead after extracting slow features is matrix multiplication and addition. For comparison, we analyzed the inference time of `transfer:sfa` and `transfer:full`. For `transfer:sfa`, 10,000 forward iterations took 6.409 ± 0.022 seconds on average out of 20 runs. If we convert these numbers to frequencies, then the 99% confidence interval is 1544-1576Hz. On the other hand, the same experiment took 6.223 ± 0.019 seconds for `transfer:full`, which translates to 1592-1621Hz. These experiments are done on an Intel i7-8700K CPU. On a comparably older CPU (i5-5257U), `transfer:sfa` has a confidence interval of 713-738Hz, and `transfer:full` has 741-766Hz. The frequency drop due to the additional computation is tolerable since the frequency is still higher than the maximum control frequency of the robot (500Hz for UR10). Moreover, SFA only creates an affine transformation, which can be integrated with the learned last layer after the training.

3.5. Conclusion

As the literature expands rapidly, the integration of architectures that are optimized for a specific scenario will gain more importance. In this chapter, we provide a principled way for transferring the existing knowledge in a network to new problems. Our experimental results show that applying SFA, an unsupervised method, to the last hidden layer of a trained network generates features that are useful for skill transfer. Transfer with SFA performs better with less number of units compared to transfer with the full layer. This is a desirable property for lifelong learning systems because of its resource economy. We see that features that are generated from SFA are more interpretable and can be treated as quasi-symbolic information. Although not fully symbolic, they are of low complexity, i.e., the response of the units does not change abruptly as we change the input (especially the first few units). These might be precursors for fully symbolic systems because of their low complexity response to input. Moreover, due to its formulation, the components with the lowest eigenvalue will contain the most useful and symbol-like information. Therefore, one can always tune the number of units by simply picking the first k components with the lowest eigenvalues, a procedure familiar to PCA.

Nevertheless, as features generated from SFA or PCA are not fully symbolic, we cannot directly use them for the purpose of planning; planning, in the context of classical AI, is a search on a tree data structure, which is composed of nodes (state) and edges (actions). If these activations are not fully symbolic, then the nodes in the tree would not be finite. In the next chapter, we propose an architecture that imposes a discreteness constraint on the learned representations so that we can easily convert and use them for domain-independent planning.

4. DEEPSYM: DEEP SYMBOL GENERATION AND RULE LEARNING FROM UNSUPERVISED CONTINUOUS ROBOT INTERACTION FOR PLANNING

4.1. Introduction

In this chapter, we address the challenging problem of discovering discrete symbols and unsupervised learning of rules from the low-level interaction experience of a self-exploring robot. For this purpose, we propose a novel deep neural architecture for symbol formation and rule extraction. At the core of our method, the symbols are discovered in the discrete latent space formed by the bottleneck layer of a predictive, deep encoder-decoder network that takes the image of an object and the action applied as the input and produces the effect generated by the action as the output. Symbols, which are the output of the encoder network, hold information for the effect prediction for a given action. Furthermore, our architecture allows for transforming the complete low-level sensorimotor experience into a symbolic experience, facilitating direct rule extraction for AI planning. To this end, decision tree models are trained to learn probabilistic rules that are translated to Probabilistic Planning Domain Definition Language (PPDDL; [98]) operators that are standard in probabilistic planning. Note that the predicates that appear in the PPDDL operators correspond to the discovered symbols.

In order to realize this framework, we created a setup where a simulated robot manipulator interacts with objects, poking them in different directions and stacking them on top of each other to collect interaction experience for object categorization and rule learning. Our system successfully constructs a latent representation through which object and relational symbols are discovered, which can be interpreted by humans as ‘rollable’, ‘insertable’, or ‘larger-than’. Contrary to symbols generated by systems that disregard actions and effects, our architecture is shown to generate action-effect-regulated symbols that are more effective in abstract reasoning over the actions of the robot and the consequences in the environment. Furthermore, the number of symbols is determined automatically by optimizing the trade-off between prediction capability and bottleneck size. Finally, the system acquired the capability to generate effective plans to achieve goals such as building towers of desired heights from given cubes, balls, and cups using off-the-shelf probabilistic planners. To show the generality of the proposed approach, we also conduct a second set of experiments in a non-robotic domain. To be concrete, we test our approach in the adapted MNIST 8-tile puzzle domain [76]. Our experiments show that the system learns symbols that allow for creating plans to move the empty tile into arbitrary positions.

Our primary contribution is a generic neural solution for mapping raw sensorimotor experience into the symbolic domain. The same architecture can be used to discover object symbols, effect symbols, and object-object relational symbols. The proposed network further allows progressive learning of increasingly complex abstractions, exploiting previously learned abstractions as inputs. The learned symbols allow the abstraction of the interaction of the robot with its environment as a Markov decision process, which allows the use of symbolic planning systems for goal satisfaction. In the current study, to show this, we transformed the learned rules into probabilistic PDDL operators, which allowed probabilistic plan generation and execution achieving goals beyond what was possible with the direct use of the training data.

4.2. Problem Formulation

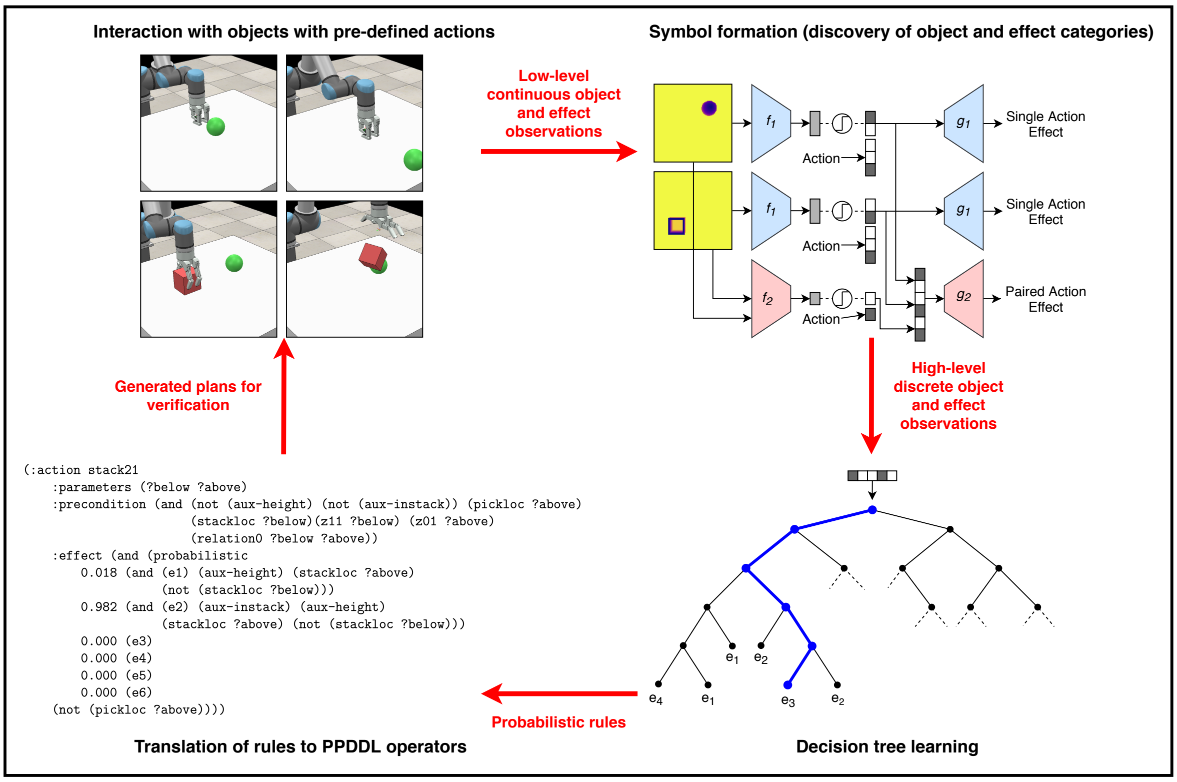
In this chapter, we refer to symbols as discrete low-dimensional vectors extracted from deep neural networks for the current state and used to predict the observed effect of specific actions. More formally, a symbol $\mathbf{z} \in \mathcal{Z}$ is a discrete representation that represents a subset $\mathcal{P}$ of a continuous high-dimensional space $\mathbb{R}^n$ (e.g., the state-space, or the effect-space). The symbol-space $\mathcal{Z}$ can be defined as a set of k-dimensional boolean vectors $\mathcal{Z} = \mathbb{B}^m = \{0, 1\}^m$, or as a set of atoms $\mathcal{Z} = \{z_1, z_2, \ldots, z_m\}$. The important condition here is that the symbol space should be finite, and its cardinality $|\mathcal{Z}|$ should preferably be small. In general, the symbol learning problem refers to finding the mapping $f : \mathbb{R}^n \rightarrow \mathcal{Z}$, which would allow us to do logical reasoning in the symbolic domain.

Given a set of discrete actions $\mathcal{A} = \{a_1, a_2, ..., a_k\}$, continuous object (or state) space $\mathbb{R}^n$, and continuous effect space $\mathbb{R}^m$, we are interested in learning an encoder function $f : \mathbb{R}^n \rightarrow \mathcal{Z}$ and a decoder function $g : \mathcal{Z} \times \mathcal{A} \rightarrow \mathbb{R}^m$ from samples $\{\mathbf{o}^{(i)}, a^{(i)}, \mathbf{e}^{(i)}\}_{i=1}^{N}$ collected by interacting with the environment. Essentially, the encoder outputs symbol $\mathbf{z}$ given the object state $\mathbf{o} \in \mathbb{R}^n$, and the decoder outputs effect $\mathbf{e} \in \mathbb{R}^m$ for symbol $\mathbf{z}$ and action a. After learning the encoder and the decoder function by iteratively optimizing an objective (which will be discussed in Section 4.3), $\mathbf{z}$ corresponds to an object symbol, and c corresponds to an effect symbol that has the grounding $\mathbf{e} = g(\mathbf{z}, a)$ (note that c is an atom while $\mathbf{e}$ is a continuous vector). Once we have such symbols, we can construct a high-level plan in the symbolic space by transforming the environment to a probabilistic PDDL domain defined over the symbols and then use state-of-the-art off-the-shelf planners to find an action sequence that arrives at the desired goal state.

The experiments reported in this chapter involve two environments from different domains, namely, a tabletop robotic manipulation environment and the adapted MNIST 8-puzzle environment [76]. The former is an embodied robotic environment in which symbols that emerge depend on the actions executed by a robotic arm and their corresponding effects. In the MNIST 8-puzzle environment, an agent without an embodiment executes actions and observes the corresponding effects as the visual change in the environment. Symbols are learned with respect to these actions and visual effects.

For simplification, we make the following assumptions in the tabletop manipulation environment. The agent is assumed to have a small number of actions, such as poking and stacking an object. Such an action repertoire can be autonomously acquired through a developmental progression [99] or obtained through learning from demonstration and reinforcement learning [100, 101]. In Chapter 7, we show how such actions can be learned from parameterized actions. The agent is equipped with image processing capability to detect the objects in the camera image and also calculate their pixel coordinates. Furthermore, using the same object-tracking method, the agent can take cropped images as input. In the tabletop setup, we realized this with a simple algorithm, as the background is uncluttered. In a real-world scenario, state-of-the-art computer vision techniques can be used to detect and track objects in the 3D world.

In the MNIST 8-puzzle environment, the only assumption is that the agent has access to the action repertoire (e.g., ‘slide-left’, ‘slide-down’), which it can execute to see the effects of its actions.


Interaction with objects with pre-defined actions
Symbol formation (discovery of object and effect categories)
Low-level continuous object and effect observations
f_1
g_1
Action
Single Action Effect
f_2
g_2
Paired Action Effect
High-level discrete object and effect observations
Generated plans for verification
e_4
e_1
e_2
e_3
Probabilistic rules
Translation of rules to PPDDL operators
Decision tree learning

```
(:action stack21
    :parameters (?below ?above)
    :precondition (and (not (aux-height) (not (aux-instack)) (pickloc ?above)
                       (stackloc ?below)(z11 ?below) (z01 ?above)
                       (relation0 ?below ?above))
    :effect (and (probabilistic
        0.018 (and (e1) (aux-height) (stackloc ?above)
                   (not (stackloc ?below)))
        0.982 (and (e2) (aux-instack) (aux-height)
                   (stackloc ?above) (not (stackloc ?below)))
        0.000 (e3)
        0.000 (e4)
        0.000 (e5)
        0.000 (e6)
    (not (pickloc ?above))))
```

Figure 4.1. General system overview of rule generation and refinement.

4.3. Methods

Figure 4.1 provides the overall learning architecture of our proposed system in the robotic manipulation environment; the application of the architecture to the MNIST 8-puzzle domain is given in Section 4.5. In the environment interaction phase, the robot chooses an action from its action repertoire $a \in \mathcal{A} = \{a_1, a_2, \ldots, a_k\}$, observes the object state $\mathbf{o}$, executes the action, and records the resulting effect $\mathbf{e}$.

Next, the interaction experience, $\{\mathbf{o}^{(i)}, a^{(i)}, \mathbf{e}^{(i)}\}_{i=1}^{N}$, is used to form symbols. To this end, a deep neural network model with two parts is trained to predict $\mathbf{e}$ given $\mathbf{o}$ and a. The first part is the encoder network, $f(\mathbf{o})$, which creates a *binary* latent vector $\mathbf{z}$ given the depth image of the object, $\mathbf{o}$. The second part, the decoder network $g(\mathbf{z}, a)$, predicts the effect $\mathbf{e}$ when action a is executed on state $\mathbf{o}$ that has the latent representation $\mathbf{z}$. As the network tries to predict effects, symbolic representations are created by the encoder network that can be treated as object categories regulated by the corresponding action-effect experience.

The continuous interaction experience $\{\mathbf{o}^{(i)}, a^{(i)}, \mathbf{e}^{(i)}\}_{i=1}^{N}$ is transformed into the symbolic experience $\{\mathbf{z}^{(i)}, a^{(i)}, c^{(i)}\}_{i=1}^{N}$ using the discovered categories, and then the symbolic experience is used to distill a decision tree to predict effects given object categories and actions. The reason to use a decision tree is that we can represent any statement in propositional logic with decision trees [102], and we can convert rules of the environment into logical statements that encode pre- and post-conditions of actions on the objects.

Finally, these statements are represented in PPDDL, which allows one to make plans in a probabilistic environment. Lastly, plans are executed to validate the learned symbols and rules. In the following sections, we describe these parts in detail.

4.3.1. Exploration with the Environment

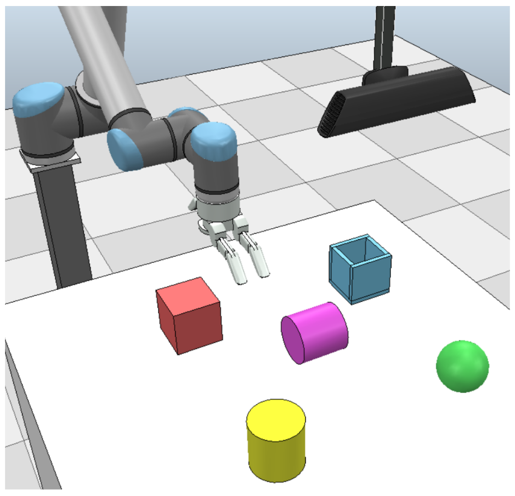
A manipulator robot with a gripper and a depth camera is used to explore the environment and monitor the changes (Figure 4.2a). The robot is initialized with a fixed set of actions $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$ through which it interacts with the objects in its workspace. Forward, side, and top poking actions are used to poke objects from different sides (Figure 4.2b, top). The stacking action is used to release one object on top of another object (Figure 4.2b, bottom). These actions are encoded with one-hot encoding. On the perception side, each detected object is represented with its top-down depth image. The generated change, on the other hand, is represented by the positional offset of the acted object in pixel coordinates together with the force change sensed at the wrist joint of the robot. In single-object interactions, the robot observes and stores the initial state as the object-centered, top-down depth image of the object and the effect as the change in object position and force sensor readings,

$$\mathbf{e}_{\text{single}} = (\mathit{\Delta} x, \mathit{\Delta} y, \mathit{\Delta} d, \mathit{\Delta} F), \tag{4.1}$$

where $\mathit{\Delta} x$ and $\mathit{\Delta} y$ are the changes in x-axis and y-axis in pixel coordinates, respectively, $\mathit{\Delta} d$ is the change in depth, and $\mathit{\Delta} F$ is the change in force. In paired-object interactions, the robot observes and stores the initial state as the combination of two object-centered depth images (o_1, o_2), and the effect as the change in position of both objects,

$$\mathbf{e}_{\text{paired}} = (\mathit{\Delta} x_1, \mathit{\Delta} y_1, \mathit{\Delta} d_1, \mathit{\Delta} x_2, \mathit{\Delta} y_2, \mathit{\Delta} d_2), \tag{4.2}$$

where $\mathit{\Delta} x_1$, $\mathit{\Delta} y_1$, $\mathit{\Delta} d_1$ refer to the displacement of the first object, and $\mathit{\Delta} x_2$, $\mathit{\Delta} y_2$, $\mathit{\Delta} d_2$ refer to the displacement of the second object.


(a)

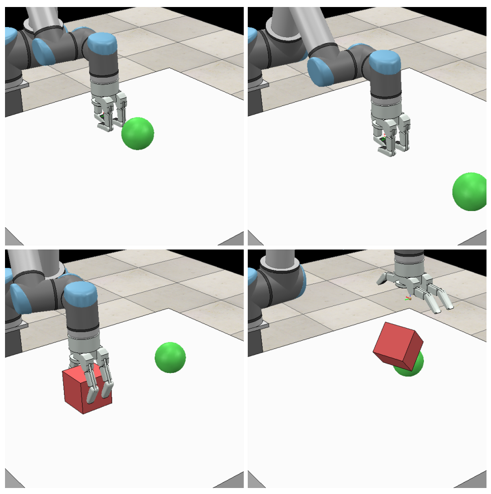

(b)

Figure 4.2. The tabletop experiment setup with UR10. A simulated UR10 robot arm and BarrettHand grasper are used for manipulation; a Kinect sensor is used for perception. Five types of objects are shown in the table. (a) The experiment setup. (b) Available actions: top—pushing an object; bottom—stacking an object.

4.3.2. Symbol Discovery with Deep Networks

The main objective of the network is to discover symbols, i.e., object and effect categories, that are effective in abstract reasoning about the consequences of robot actions. In other words, the object categories, together with robot actions, should give the ability to predict the effect categories. To achieve this, we propose a special neural network structure that is composed of two parts: an encoder $f(\mathbf{o})$ to predict $\mathbf{z}$ which is the object category, and a decoder $g(\mathbf{z}, a)$ to predict $\mathbf{e}$ (Figure 4.3, top). This is an encoder-decoder design that has been shown to be quite successful in many different applications [20,21], [103,104]. The binary bottleneck layer forces the network to learn low-dimensional symbolic representations that are useful for predicting the generated effect of actions. As the input is a top-down depth image, the encoder is a convolutional neural network with the Gumbel-Sigmoid (GS) function [51, 52] as the last-layer activation function (where the error back-propagation is handled with the reparameterization trick; [104]). We also experimented with the $sign(x)$ function using straight-through estimators (STE; [105]) and found that GS has a lower variance. Results with STE are given in Appendix A.2. Using GS activation of the bottleneck neurons, the continuous representation is directly transformed into a discrete category. The decoder part is realized as a multi-layer perceptron (MLP). The category $\mathbf{z}$ of the object $\mathbf{o}$ concatenated with the one-hot vector of action a is given to the decoder as input. The decoder predicts the effect $\mathbf{e}$ expected to be observed on object state $\mathbf{o}$ via action a. The network minimizes the following objective:

$$\mathcal{L} = \sum_{i=1}^{N} \frac{1}{2} \left(g(f(\mathbf{o}^{(i)}), a^{(i)}) - \mathbf{e}^{(i)} \right)^2 . \tag{4.3}$$

This architecture effectively creates high-level symbolic categories of objects that encapsulate the effects of executed actions. One important advantage is that the model does not need hand-engineered object features and object clusters for finding object symbols, contrary to previous studies, since the system learns *discrete* categories directly to optimize the effect prediction performance. Moreover, as the bottleneck layer is discrete, the possible decoder outputs $\mathbf{e} = g(\mathbf{z}, a)$ form a finite set $\mathcal{E} = \{\mathbf{e}_1, \mathbf{e}_2 \ldots\}$

which can be denoted by atoms $\mathcal{C} = \{c_1, c_2, \dots\}$. The learned object categories also serve as input for the discovery of new categories with new interaction experiences, such as stacking an object $\mathbf{o}_1$ on top of another object $\mathbf{o}_2$ (Figure 4.2b, bottom).

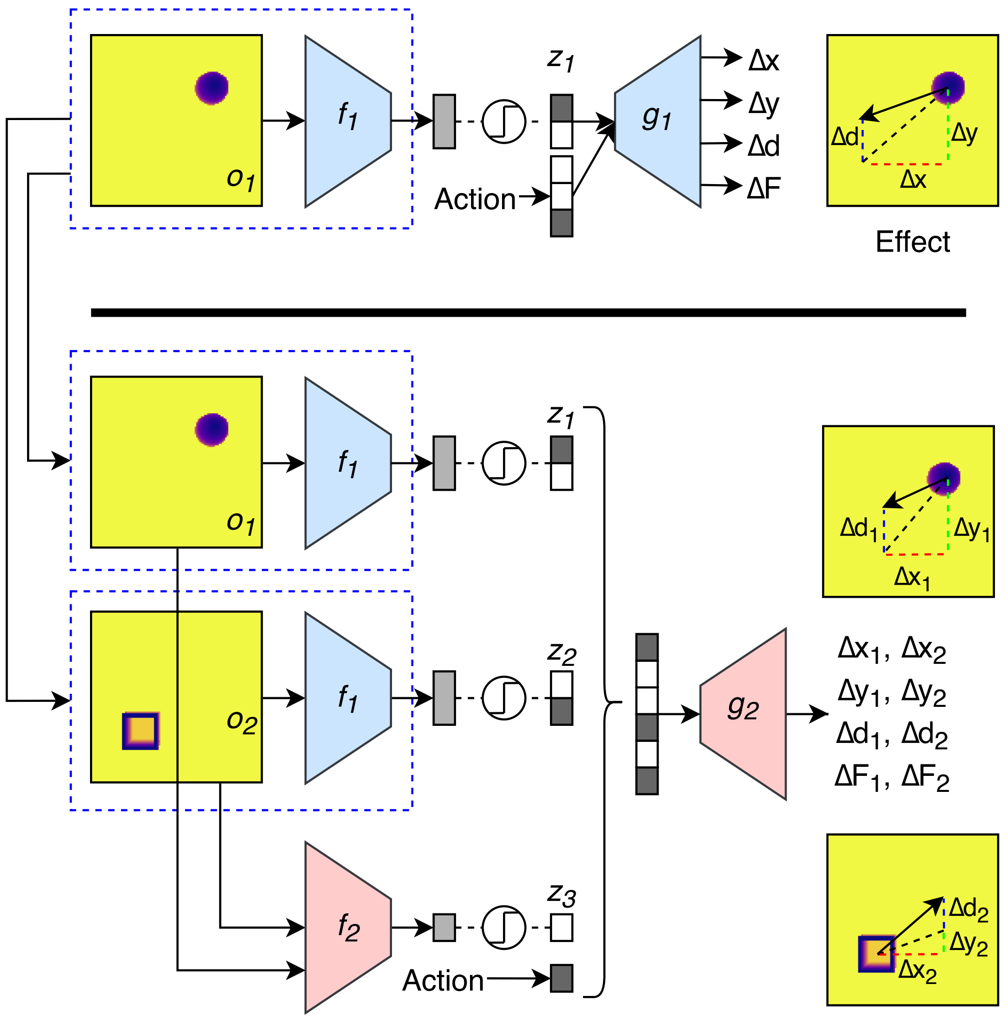


Figure 4.3. Network architectures for single-object interactions (top) and for paired-object interactions (bottom).

The same deep network structure is used to extract the corresponding symbols with a slight modification to incorporate previously learned knowledge (Figure 4.3, bottom). Here, an encoder f_2 takes the depth images of the objects and produces a binary latent vector $\mathbf{z}_3$. As the important point here, the single object symbols ($\mathbf{z}_1$ and $\mathbf{z}_2$) computed by the f_1 encoder are also added to the network as input together with the action information. The idea is that we can use previously acquired symbols to encode new information more compactly, thus allowing a progressive increment of symbols. Note that the encoder f_1 is frozen at this second stage of the training. The encoder f_1 provides some interaction-related information about objects and lets the encoder f_2 focus and learn properties and relations *between* the objects.

Number of symbols is automatically set by selecting the number of bottleneck neurons using a hyperparameter search procedure. To limit the number of rules and predicates, this procedure aims to find the minimum number of symbols that provide competitive performance in prediction. Starting from one unit, we record the mean and the standard deviation of mean square errors (MSE) of multiple runs. We increase the number of units until there is no significant drop in the prediction error. MSE curves are reported in Appendix A.1.

4.3.3. Extracting Symbolic Rules

In the third part of the pipeline, a decision tree is trained to predict the effect c of the stack action a given high-level single ($\mathbf{z}_1$ and $\mathbf{z}_2$) and paired ($\mathbf{z}_3$) object categories (i.e., the dataset is $\{[\mathbf{z}_1^{(i)}; \mathbf{z}_2^{(i)}; \mathbf{z}_3^{(i)}; a^{(i)}], c^{(i)}\}_{i=1}^{N}$). Here, the aim is to extract the probabilistic rules of the environment by converting the decision rules on the paths of the tree into logical statements, which ultimately enables probabilistic planning. Each path from the root node to a leaf node in the decision tree stores the required set of predicates $\{p_1 = (z_3 < 0.5), p_2 = (z_2 > 0.5), \dots\}$ represented by discovered single and paired-object categories (in the internal nodes) in order to achieve the effect category c (in the leaves). In other words, each path corresponds to a set of preconditions in order to reach a different effect. As the decision rules at each node in a path $\mathcal{P}$ are in conjunction ($p_1 \wedge p_2 \wedge \cdots \wedge p_k$), and these paths are in disjunction ($\mathcal{P}_1 \vee \mathcal{P}_2 \vee \cdots \vee \mathcal{P}_m$), the tree represents a statement in disjunctive normal form. Thus, any statement in propositional logic can be represented as a decision tree [102]. The class probabilities at a leaf (the fraction of samples) correspond to probabilities of observing different effects for the same set of preconditions. Therefore, each path is directly converted to a different rule in probabilistic PDDL. While training the decision tree, the minimum number of samples required for a node to be a leaf node is empirically set to 100 samples. The extracted rules are only limited to predicting the effects of an action. In this way, the agent is not expected to learn representations (and consequently rules) unrelated to its embodiment and actions. For example, in our tabletop environment, the robot cannot differentiate cubes from vertical cylinders as different categories since they respond similarly to similar actions, even though their visual appearances differ.

Our motivation for constructing PPDDL descriptions is to use probabilistic AI planners to make plans and execute them efficiently. PPDDL is composed of a domain description and a problem definition. In the domain description, there are predicates and actions. Predicates represent boolean values that can be activated or deactivated. Each action has a precondition, which is a set of predicates that needs to be satisfied, and an effect, which activates/deactivates other predicates. The domain description is generated from the list of rules. In the problem definition, the initial state of the world is encoded along with the goal to be satisfied. To encode the initial state, the robot perceives the current environment and sets the truth values of the predicates for the existing categories. The planner finds the sequence of actions to satisfy the predicates given in the goal description, starting from the initial state and using the actions defined in the domain description.

4.4. Robot Experiments

In the following experiments, we aim to answer the following questions to evaluate the proposed method: (i) Do the learned symbols hold any high-level meaning? (ii) Are the learned symbols effective for symbolic planning? We compare our method with two alternative baselines. The first one is an autoencoder with discrete activations where symbols are learned directly from passively observed states, independent from actions and effects. The second one is an encoder-decoder network with continuous activations, followed by clustering in the latent space. Regarding the first question, we evaluate the methods based on their performance in differentiating object categories. For the second question, we evaluate the planning performance of different methods. The planning performance is evaluated by the success rate of the plans generated by the learned rules.

4.4.1. Experiment Setup

4.4.1.1. Interactions. We adopted the robotic setup, including the action and object sets used, from Ugur and Piater [42], which showed effective skill transfer from the simulator to the real world, involving actions with 3-fingered prehension. The experiments are performed in CoppeliaSim VREP simulator [95] where a six-degrees-of-freedom UR10 [96] robot arm and a Barrett Hand system [106] interacts with the objects on the table, and a top-down facing Kinect sensor is used for environment perception (Figure 4.2). The objects used in the experiments include rectangular cups, horizontally and vertically placed cylinders, spheres, and cubes. For each object type, ten different objects with varying diameters/edge lengths in the range of 10 to 20 cm are included in the object dataset for interaction.

4.4.1.2. Perception. Before each action execution, a top-down depth image (128×128 pixels) of the scene is captured. Objects are placed at different reachable locations on the table during the interactions to ensure the network is invariant with the perspective. Pixels of the images are normalized globally to increase the convergence speed of stochastic gradient descent [94]. Objects in the image are detected with a simple procedure by finding the point with minimum depth and cropping the area of 42×42 pixels centered around it. This procedure yields object-centered representations for the objects used in the current study but preserves the perspective distortion due to the varying locations of the objects and fixed sensor position.

4.4.1.3. Encoder-decoder network. The encoder network (Figure 4.3) consists of four blocks, each containing two convolutional layers that are followed by batch normalization [8] and ReLU activation. The numbers of filters in these blocks are 32, 64, 128, and 256. The last layer consists of two hidden units with a GS activation. The decoder network is a two-layer MLP with 32 hidden units. Further details of these networks can be found in Appendix A.1.

4.4.2. Discovered Object Categories

Based on the hyperparameter optimization procedure, the number of binary activation neurons in the bottleneck layer is automatically set to 2; therefore, the system found $2^2 = 4$ object categories. How different object types (unknown to the robot) are represented by the discovered object categories is analyzed and provided in Table 4.1. In general, different types of objects were coded into different categories, except that cubes and vertical cylinders share the same category even though their depth images differ. This is due to our action and effect-regulated categorization: cubes and vertical cylinders behave the same under all available single-object actions of the robot. Although the depth images of the same type of objects with different sizes differ significantly, this information is not reflected in the categories because the size of the objects does not have a significant influence on the consequences of the current actions. The categories can be interpreted as ‘pushable’, ‘rollable in a single direction’, ‘pushable and insertable’, and ‘rollable in all directions’, respectively. Examples from each category are shown in Figure 4.4.

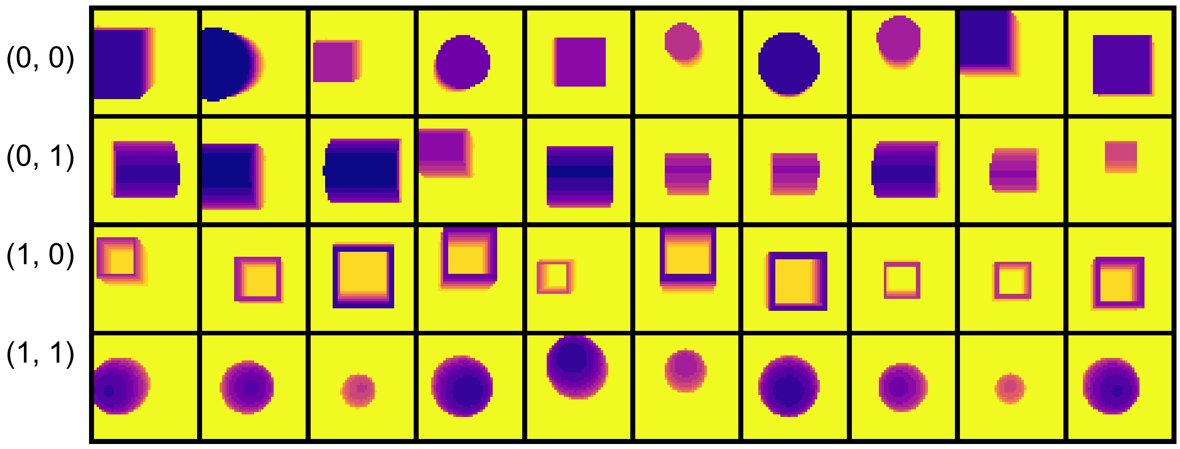


Figure 4.4. Example depth images as inputs to the encoder network f_1.

Table 4.1. The relative assignment frequencies of objects to different symbols.

DeepSym				
Category	(0, 0)	(0, 1)	(1, 0)	(1, 1)
Sphere	99.9 ± 0.2	0.1 ± 0.2	0.0 ± 0.0	0.0 ± 0.0
Cube	0.0 ± 0.0	99.9 ± 0.2	0.0 ± 0.0	0.1 ± 0.2
Vertical Cylinder	0.0 ± 0.0	99.9 ± 0.2	0.1 ± 0.2	0.0 ± 0.0
Horizontal Cylinder	0.4 ± 0.9	3.1 ± 5.5	93.0 ± 4.7	3.4 ± 3.8
Cup	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	100.0 ± 0.0
Autoencoder (OBO)				
Category	(0, 0)	(0, 1)	(1, 0)	(1, 1)
Sphere	60.6 ± 1.7	15.7 ± 5.0	15.0 ± 4.5	8.8 ± 3.2
Cube	37.4 ± 2.3	22.9 ± 3.1	19.3 ± 2.4	20.4 ± 4.3
Vertical Cylinder	44.0 ± 2.9	23.2 ± 5.7	19.4 ± 7.2	13.4 ± 3.1
Horizontal Cylinder	44.7 ± 2.9	20.0 ± 4.9	18.6 ± 3.7	16.7 ± 4.5
Cup	86.5 ± 2.0	4.3 ± 2.1	6.4 ± 2.3	2.8 ± 3.7
Continuous bottleneck + clustering (OCEC)				
Category	(0, 0)	(0, 1)	(1, 0)	(1, 1)
Sphere	94.2 ± 9.4	4.0 ± 8.4	1.8 ± 5.5	0.0 ± 0.0
Cube	0.0 ± 0.0	99.0 ± 3.2	0.0 ± 0.0	1.0 ± 3.2
Vertical Cylinder	0.0 ± 0.0	98.5 ± 4.7	0.0 ± 0.0	1.5 ± 4.7
Horizontal Cylinder	28.6 ± 29.1	0.0 ± 0.0	63.4 ± 26.1	8.0 ± 16.9
Cup	0.0 ± 0.0	13.8 ± 32.5	0.0 ± 0.0	86.2 ± 32.5

As a baseline for comparison, we trained an autoencoder with a binary hidden layer (similar to [76]) using Gumbel-Sigmoid to reconstruct the depth images of objects (inputs to f_1) instead of effects, dubbed as Object-Binary-Object (OBO). As a second baseline, we trained our proposed encoder-decoder architecture with the binary bottleneck layer replaced with a usual continuous layer that is applied k-means clustering ($k = 4$) after learning. Let us call this approach Object-Continuous-Effect followed by Clustering (OCEC).

The results are shown in Table 4.1. Here, objects vary in their sizes and initial positions. The mean and the standard deviation of 10 runs are reported. For ease of understanding, we name columns so that the category where spheres are mostly placed is renamed to (0, 0), the category where cubes are mostly placed is renamed to (0, 1), and so on. The naming convention also allows us to take an average across different runs. For the autoencoder network (i.e., OBO), we see that objects are collapsed primarily into one category. The robot is expected to predict the consequences of its actions using these categories, and as shown, these categories are not distinctive to help such prediction. With this, we verified the advantage of extracting the symbols from the interaction experience of the robot that includes object-action-effect information, i.e., from an object encoder-effect decoder network, rather than searching the symbols in *passively-observed* static features.

OCEC gave better results compared to OBO since the bottleneck layer in OCEC *does* include information from the effect space because of the predictive training similar to our proposed model. However, the latent codes in the bottleneck layer of OCEC might not be distributed locally, making clustering harder. When this is the case, we need more complicated clustering algorithms, such as spectral clustering, to cluster the latent space accurately. For example, in Table 4.1, we see that OCEC is more biased toward misclassifying cups as the stable category and the horizontal cylinders as spheres. When we take an average over all objects, our method predicts objects in the correct category with 98.5 $\pm$ 0.94 % accuracy compared to OCEC with 88.3 $\pm$ 8.62 % accuracy.

4.4.3. Discovered Relational Categories

In the next step, we train a new pair of encoder-decoder network with the stack interaction experience of the robot (Figure 4.3 bottom) while transferring the learned object categories. By using the previously learned object symbols, the system is expected to learn symbols that the previous symbols do not capture. The number of units is set to one using the parameter search defined previously.

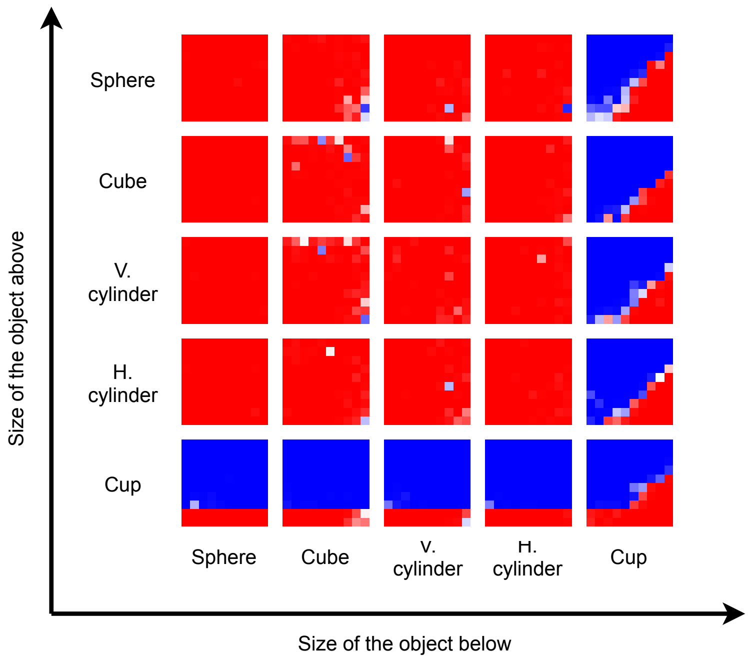


Figure 4.5. The encoder f_2 activations (blue for 0, red for 1) for paired objects. Here, x and y axes of each of the 5×5 plots represent the sizes of the objects below and above, respectively. Each square represents the relation for a given pair. Note that without any direct supervision, the system discovers approximately linear boundaries (e.g., the last column) for some object pairs that would help in effect prediction.

The response of the bottleneck neuron, i.e., how this neuron categorizes the input object pairs, is analyzed in Figure 4.5. Given different pairs of objects with different sizes, each image in this figure corresponds to a specific object pair, and each pixel provides the response of the bottleneck neuron (0 or 1) for specific object sizes. In our experiments, the effect of stack action depends on object categories and their relative size. For example, if an object is released on top of a larger cup, the released object drops into the cup. If the released object is larger than the cup, it is stacked on top of the cup's walls. The approximately linear boundaries for some object pairs in Figure 4.5 (for example, the last column) show that the bottleneck neuron captured these dynamics and found a symbol that roughly encodes the relative size; the output is one when the below cup is larger than the above object. In stacking interactions, the relative size relation only makes sense when the object below is a cup, and our system discovered this relational symbol. Another linear boundary found by the system is in the bottom row. The output of the encoder is one when the above object is a cup and below a specific size. We analyze the exploration data to understand why such a boundary emerges. We found out that if the above object is a small cup, the change in the position of the below object is very small.

The learned representations depend on the effect space and the action space of the agent. In our example, after the single-object training stage, the system differentiates different types of objects but does not differentiate different sizes of objects as they are not sufficiently important for the prediction of push actions. Only after it is trained with new data consisting of a new action, namely stacking, does the system start to differentiate between different sizes of cups. The agent only learns richer representations, and therefore better rules, when it has access to a richer action repertoire. This is a desired property of our system as it learns a minimal set of representations needed to predict the outcomes of its actions.

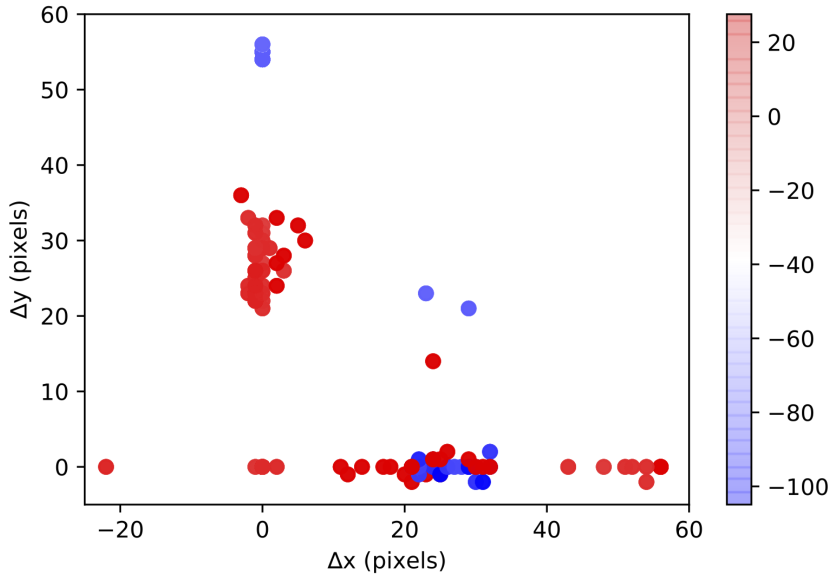
4.4.4. Discovered Effect Categories

After training, we pass the symbol space $\mathcal{Z}$ together with the action space $\mathcal{A}$ to the decoder to get the effect categories. More specifically,

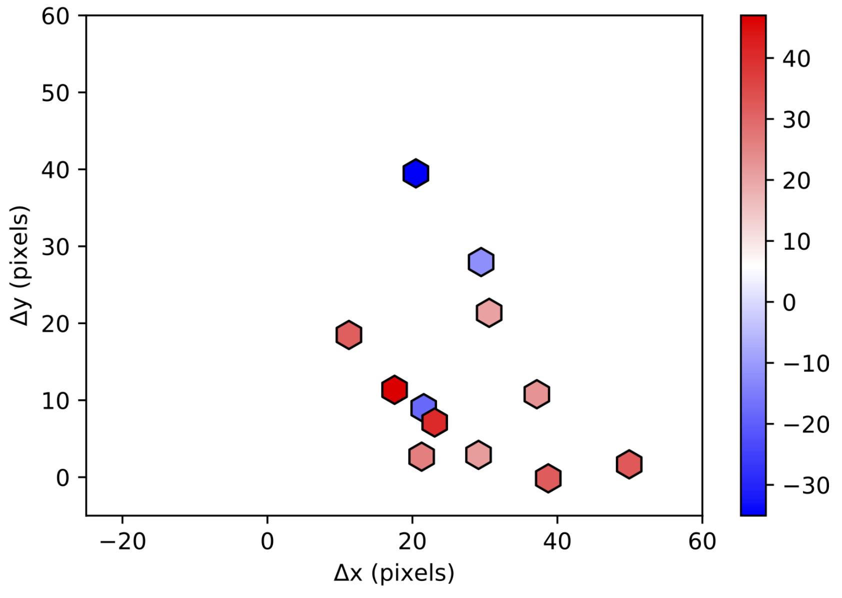
$$\mathcal{C}_{\text{single}} = g_1(\mathcal{Z}_{\text{single}}, \mathcal{A}_{\text{single}}) \tag{4.4}$$

$$\mathcal{C}_{\text{paired}} = g_2(\mathcal{Z}_{\text{paired}}, \mathcal{A}_{\text{paired}}, \tag{4.5}$$

where $\mathcal{Z}_{\text{paired}}$ is the Cartesian product of the object category space $\{0,1\}^2$ with the action space $\mathcal{A}_{\text{single}} = \{(0,0,1),(0,1,0),(1,0,0)\}$ resulting in 12 different effect categories for the single object effects. For the paired object effects, the input consists of two single object categories and one relational object category. Therefore, this number is $\{0,1\}^2 \times \{0,1\}^2 \times \{0,1\} \times \mathcal{A}_{\text{paired}} = 32$. Here, A_{paired} only contains the stack action, therefore $n(\mathcal{A}_{\text{paired}}) = 1$. These effect categories for the single and the paired interactions are shown in Figures 4.6 and 4.7, respectively. For visualization purposes, we use colors to represent the third dimension. In Figure 4.6, the low force values are in blue, and the high force values are in red. Likewise, in Figure 4.7, the low-depth values are in blue, and the high-depth values are in red. We omit effects of the below object $(\Delta x_2, \Delta y_2, \Delta d_2)$ in Figure 4.7 as they are almost zero. We see that the found effect categories faithfully represent the effect space without any clustering.

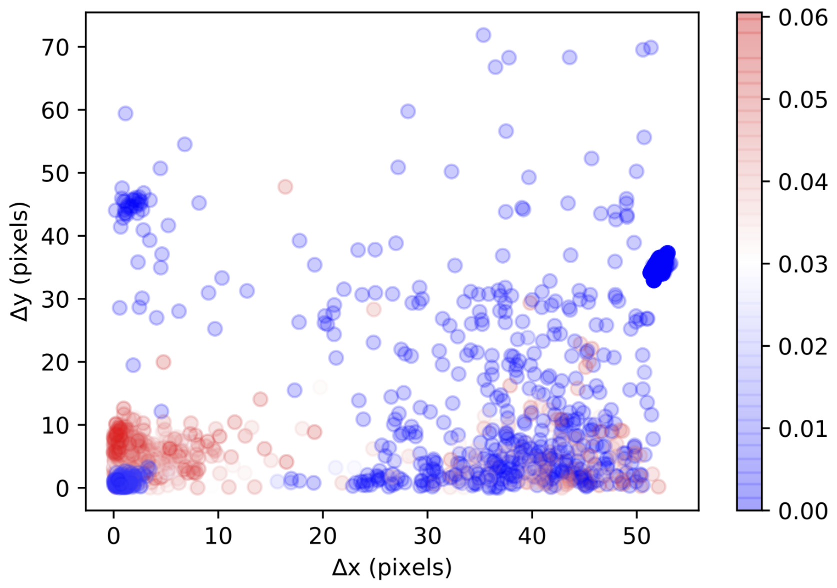


(a)

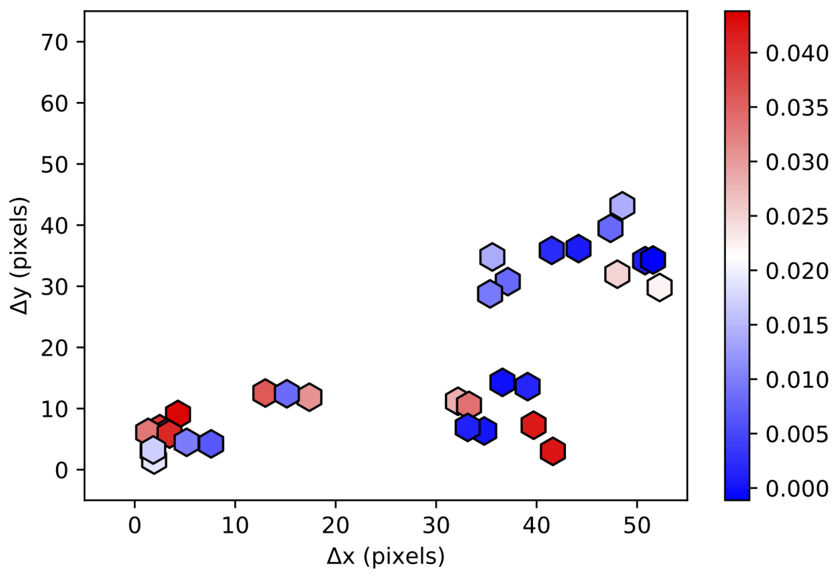


(b)

Figure 4.6. Effect space for the single object interactions. The low force values are in blue, and the high force values are in red. (a) Observed effects. (b) Found effects.



(a)



(b)

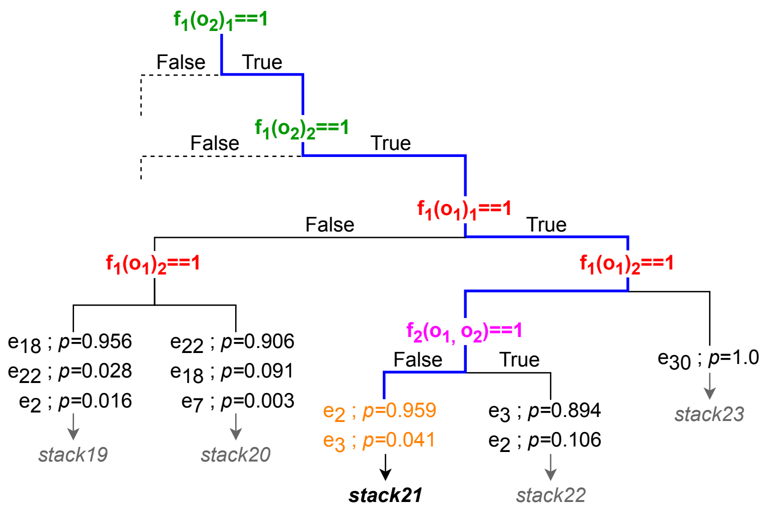
Figure 4.7. Effect space for the paired object interactions. The low-depth values are in blue, and the high-depth values are in red. (a) Observed effects. (b) Found effects.

4.4.5. Learned Rules and PPDDL Operators

The single- and paired-object categories (acquired from the output of the encoder), together with the action vector, are used as inputs to the decision tree in order to predict the effect categories (extracted from the output of the decoder). The learned tree is of depth 5, has 24 leaves, and its classification accuracy is 94.8%. The result of decision tree learning is shown in Figure 4.8a, where only a small number of decision paths out of 24 is explicitly shown because of the space constraints. Decision rules for the highlighted path is $(f_1(o_1)_1, f_1(o_1)_2, f_1(o_2)_1, f_1(o_2)_2, f_2(o_1, o_2)) = (1, 0, 1, 1, 0)$. Here, $(f_1(o_1)_1, f_1(o_1)_2)$ represents the category of the object above, $(f_1(o_2)_1, f_1(o_2)_2)$ represents the category of the object below, and $f_2(o_1, o_2)$ is the symbol for the paired-object relation. A natural-language translation of this path is as follows: 'If the above object is rollable in all directions (1, 1), and the below object is pushable and insertable (1, 1), and the below object is not larger than the above object, e_2 is observed (which is a stacking effect) with 0.959 probability'. PPDDL description corresponding to this decision path of the tree is shown in Figure 4.8b.

For our experiments in the tower building task, we manually introduced some auxiliary predicates, as well as special actions for the domain, to be able to chain multiple actions and count the number of objects in the tower. These are needed to set a goal of constructing a tower with multiple objects that are outside the robot's experience.

For effects with small Δx_1 and Δy_2, the `aux-instack` predicate is set to true if they satisfy $\Delta d_1 > \epsilon$ for some threshold ϵ, and otherwise, the `aux-height` predicate is set to true. For this specific application, these predicates allow us to differentiate stacking and inserting—our effects of interest—from other effects. Actions `increaseheight1` and `increasestack1` are treated as addition operators that increase the height of the tower (H) and the number of objects in the stack (S), respectively. There are multiple H and S predicates ranging from H1-H7 and S1-S7, and likewise multiple `increaseheight` actions. When the `aux-height` effect is observed, the planner must select the `increaseheight1` action to continue with the plan. Therefore, when a stack effect is observed, the height of the tower (which is represented by H) increases automatically. These would not be needed if we were to use the numeric values associated with effect clusters (i.e., functions in PDDL). We went on with the atomic effect representation (i.e., with no associated parameters) as they worked better with the mGPT implementation that we used.



(a)

```
(:action stack21
  :parameters (?below ?above)
  :precondition (and (not (aux-height) (not (aux-instack))
                  (pickloc ?above) (stackloc ?below)
                  (f10 ?below) (f11 ?above)
                  (relation0 ?below ?above))
  :effect (and (probabilistic
                0.959 (and (e2) (aux-height) (stackloc ?above)
                      (not (stackloc ?below)))
                0.041 (and (e3) (aux-instack) (aux-height)
                      (stackloc ?above) (not (stackloc ?below)))
                0.000 (e1)
                 ..
                0.000 (e30)
               (not (pickloc ?above))))
(:action increaseheight1
  :precondition (and (aux-height) (H0))
  :effect (and (not (H0)) (H1) (not (aux-height))))

(:action increasestack1
  :precondition (and (aux-instack) (S0))
  :effect (and (not (S0)) (S1) (not (aux-instack))))

(:action makebase
  :parameters (?obj)
  :precondition (not (base))
  :effect (and (base) (aux-height) (aux-instack) (not (pickloc ?obj))
            (stackloc ?obj)))
```

(b)

Figure 4.8. An example expansion of the decision tree. On the leaves, each effect e_i is observed with the corresponding probability. (a) An example of a decision path and (b) its translation to PPDDL description, color-coded to match the decision tree.

Lastly, the predicate `pickloc` is true for objects that are on the table and available for use for the tower construction; `stackloc` is true for the object that is at the top of the tower. These are shown in Figure 4.8b.

4.4.6. Performance of Planning

The PPDDL descriptions that are automatically constructed by the discovered symbols and rules were verified by generating plans given a set of goals, executing these plans in the simulator, and assessing the success of the executed action sequences in achieving these goals. To be concrete, we asked the system to generate plans to create towers of desired heights with a given fixed set of objects. The challenge of the task is to place objects on top of each other in the correct order. With five objects, there are 5! plans. For plan generation, the mGPT planner is used [107].

Since in our experiments, the system is asked to generate plans given a number of objects on the table, we encode the task of, say, "construct a tower with a height of three (H3) using four objects (S4)" as H3S4 (Figure 4.9).

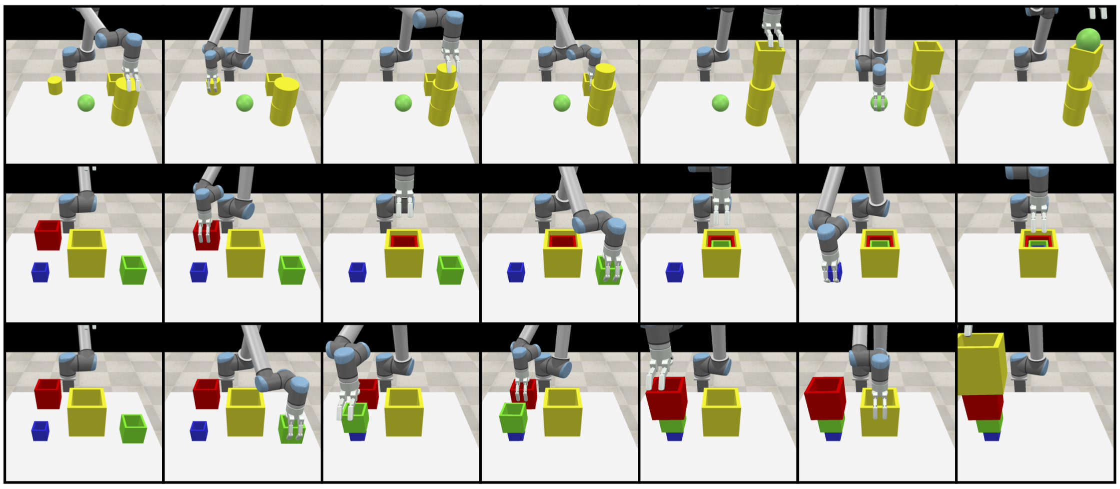


Figure 4.9. Top—The objective is to construct a tower of height five using five objects, H5S5. The system assesses the success probability to be 0.07. Middle—The objective is H1S4, and the system assesses the success probability to be p=0.76. Bottom—If we change the objective to H4S4, the success probability increases to 0.88.

4.4.6.1. DeepSym vs. OCEC. We first compare our system with the alternative OCEC system. We train both systems ten times and select the best-performing models (based on the decision tree accuracy). We initialized 20 random problems and asked the planner to solve the task using two different domain descriptions generated from different methods. We run the probabilistic planner 100 times for each problem and record the number of successes to estimate the success probability of the plans. The results are reported in Table 4.2. We report two different metrics: (1) planning success shows whether the system generated a feasible plan or not, and (2) execution success shows whether the execution is successful or not. The latter is concerned with the stochasticity of the environment, not with the feasibility of the plan. We see that the OCEC model performs considerably worse with 25% planning accuracy than our approach with 95% planning accuracy. This is mainly caused by the wrong classification of the cup object (see Table 4.1 in Section 4.4.2), which is an essential piece of information in this problem. When single- and paired-object categories are incorrectly classified, the system generates an invalid problem description, which results in infeasible plan outputs. For the same number of symbols, the learned symbols in the OCEC pipeline do not directly depend on the action and the generated effects, while symbols learned with our architecture directly depend on the action and its corresponding effect as they are directly used for effect prediction. This leads to the creation of symbols that are more appropriate for planning.

Table 4.2. Planning results from random 20 configurations.

Method	Estimated prob.	Planning success	Execution success
DeepSym	80.4	95.0	70.0
OCEC	12.1	25.0	15.0

4.4.6.2. DeepSym Performance. Now that we have shown the performance gap between the two methods, we want to analyze when our method fails and succeeds. We considered four different goals (towers of heights from one to four) and performed 25 different runs with random initial object configurations for each objective. We configured object types and sizes to have at least one feasible solution. For example, for the H1 objective, we make sure that there are at least three cups that can be physically stacked into each other. The initial state and the final state of each problem are provided in Appendix A.4. The plan execution performance is reported in Table 4.3. There are three different outcomes: (1) the plan is executed successfully, (2) the planner outputs an erroneous plan due to a recognition error in the encoder, (3) the generated plan is correct, but the plan fails at execution time due to the stochasticity of the environment.

We see that the robot constructs towers with a height of four successfully. As the height of the tower decreases, the robot needs to insert some of the objects inside other cups. The insertion task is harder than the stacking task due to the stochasticity of the environment, which is also reflected in the estimated probabilities in Figure 4.8b; even if the below cup is larger than the above sphere, the insertion probability is 0.894. For example, for the challenging objective of creating an H1 tower including all objects, the system estimates the success probability to be 0.68 and, therefore, the failure probability to be 0.32. Accordingly, 36% of plans fail at execution time. This shows that the system can partially model the probabilistic nature of the environment. The planning errors are mostly due to the incorrect recognition of the paired-object categories. Example executions are shown in Figure 4.9.

<u>4.4.6.3. Deterministic vs. Probabilistic Planning.</u> We also experimented with deterministic planning instead of a probabilistic one. To do so, while converting rules to PDDL, we take the maximum likely effect as the generated effect. For example, if a specific action schema produces effect e_2 with a probability of 0.91 and e_{17} with a probability of 0.09, we take the effect with the maximum probability as the generated effect for the action schema. Thus, deterministic planning eliminates the possibility of other effects and, therefore, effectively eliminates possible solutions. When the learned rules faithfully represent the action-effect relations in the environment, we observe no significant difference between probabilistic and deterministic planning in terms of the success of plans. However, when there is significant inaccuracy in the learned repre-

sentation (e.g., an incorrect comparison between a pair of objects), the probabilistic PDDL description can account for this inaccuracy in the probabilities of effects; the inaccuracies are reflected as the uncertainty of the environment.

Table 4.3. Planning results from 25 executions for each task. To satisfy the H1S4 objective, the robot needs to insert objects inside each other, which is more challenging compared to the other tower tasks. Thus, the success probability is lower.

Task	H4S4	H3S4	H2S4	H1S4
Estimated execution probability	0.91	0.93	0.86	0.68
Success	0.88	0.56	0.68	0.32
Planning fail	0.12	0.20	0.20	0.32
Execution fail	0.00	0.24	0.12	0.36

4.5. Experiments on 8-puzzle

In this section, we evaluate DeepSym on the adapted MNIST 8-puzzle [76]. In the original 8-puzzle, the aim is to have the tiles in a specific arrangement (considered the goal configuration) by moving tiles into the empty square. In the adapted MNIST 8-puzzle version, tiles do not have symbolic values such as digits but instead contain images of digits, and the 0-tile is treated as the empty tile. Given the domain definition, i.e., the knowledge of how the configuration changes in response to slide actions, the 8-puzzle game can be solved with AI planners. However, the problem becomes non-trivial when the states are represented with raw images of the board, and the state transitions are not known. In the adapted MNIST 8-puzzle, our system is given the raw image of the board with $(3 \times 28) \times (3 \times 28) = 7056$ pixels. Therefore, the state vector is 7056-dimensional. An instance of the MNIST 8-puzzle is shown in Figure 4.10.

A system that can solve the puzzle should recognize the following: (i) actions only modify some part of the image (i.e., there are tiles), (ii) there are specific symbolic representations in these tiles (i.e., recognize the image content of the tiles), and (iii) the goal is only valid when these tiles are arranged in a specific order (sorted from left to right and top to bottom).

As in our robot experiments, the general pipeline (Figure 4.1) consists of four stages: (1) exploration, (2) symbol learning, (3) rule learning, (4) and the translation of rules to PDDL.

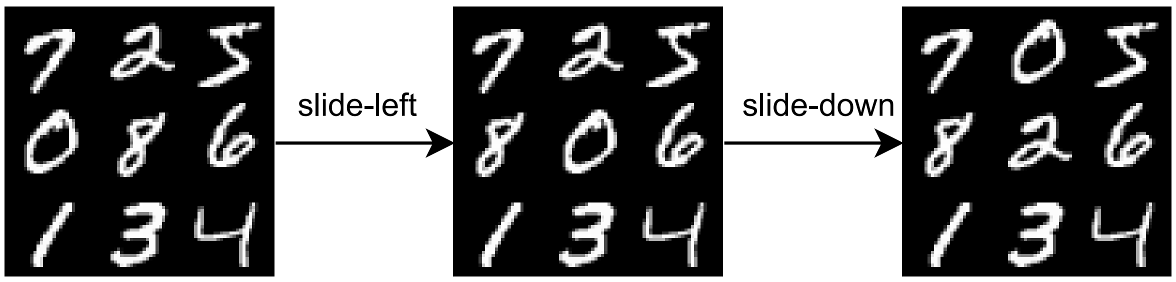


Figure 4.10. Two steps of the MNIST 8-puzzle. The 0-tile is treated as the empty tile. Each tile consists of a 28 × 28-pixel MNIST digit.

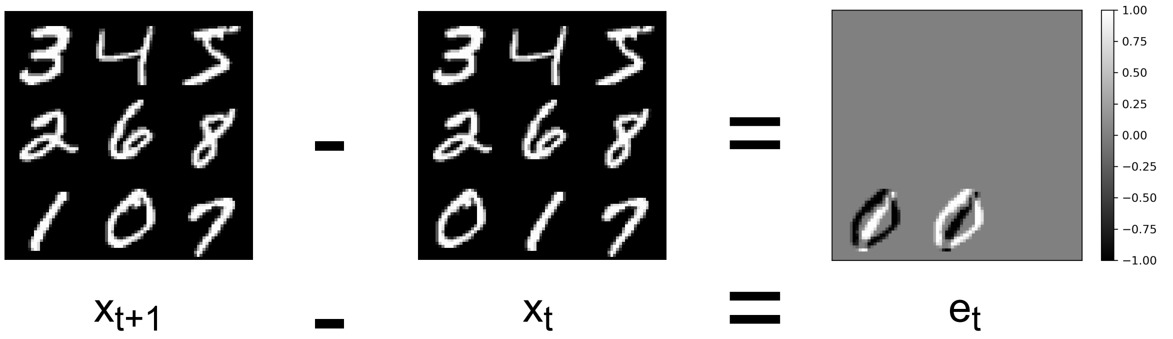


Figure 4.11. The effect is represented as the elementwise different of pixels between two consecutive timesteps.

In the exploration stage, the system initializes a random environment configuration, executes a random action (which is provided to the system), and records a 3-tuple $(\mathbf{x}_t, a_t, \mathbf{e}_t)$ where $\mathbf{x}_t$ is the current state, a_t is the executed action represented as a one-hot vector, and $\mathbf{e}_t$ is the generated effect represented as the pixel difference between the new state $\mathbf{x}_{t+1}$ and the current state $\mathbf{x}_t$ (Figure 4.11). We collect 100,000 such interactions from the environment.

In the symbol learning stage, we train an encoder-decoder network as in Section 4.3.2, where the encoder $f(\mathbf{x})$ takes the state vector $\mathbf{x}$ as an image of 84×84 pixels and outputs a binary vector $\mathbf{z}$, and the decoder $g(\mathbf{z}, a)$ takes the concatenation of $\mathbf{z}$ and the action vector a to produce the effect $\mathbf{e}$ which is also an image of 84×84 pixels. Both the encoder and the decoder are convolutional networks. We did not employ any hyperparameter search on the architectures but followed the building principles in DCGAN [108]. The details of the networks can be found in Appendix A.1. We train the model for 100 epochs with MSE loss in Equation (4.3).

After training, we distill the information in the decoder network into rules by training a decision tree using the predictions of the decoder. Lastly, we translate the rules represented by the decision tree into PDDL rules as in Section 4.3.3.

4.5.1. Learned Symbols

In the MNIST 8-puzzle environment, there is a finite set of possible effects that can be generated in a single action from any environment configuration. If we use the same image for a digit, then the encoder should represent 3248 different states (digits that are near the empty tile) in order for the decoder to produce the correct effect. Since we are using binary activations, $\log_2 3248 \approx 11.67$ units are necessary to avoid losing any information regarding effects. Therefore, we set the number of units to 13 (giving one more as a slack) in this experiment.

To understand the symbols that correspond to the low-level subsymbolic representations (i.e., images), we sample 100,000 random states from the environment and get their symbolic representations from the encoder. Then, we take the average of images that correspond to the same symbol. We show the average image that corresponds to the top 30 symbols sorted by their activation counts in Figure 4.12. We notice that the first nine symbols correspond to different locations of the empty tile (recall that the digit '0' is considered as the empty tile), which accounts for 41.5% percent of all activations (i.e., in 41.5% of the time, the encoder only outputs the position of the empty tile). Other symbols correspond to cases where the digit '3' or '5' is near the empty tile.

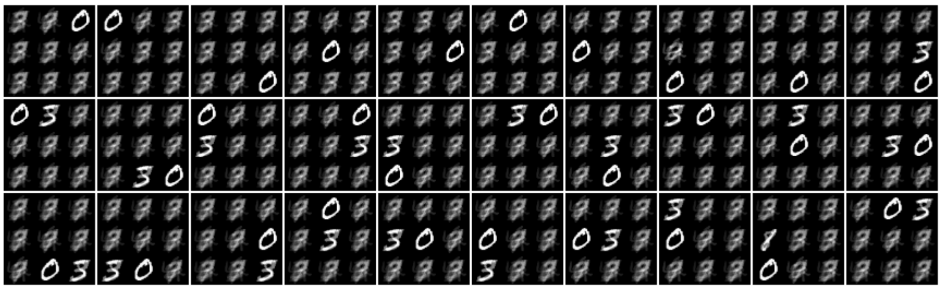


Figure 4.12. Average states that correspond to the top 30 symbols on MNIST 8-puzzle (sorted by their activation count from left to right and top to bottom).

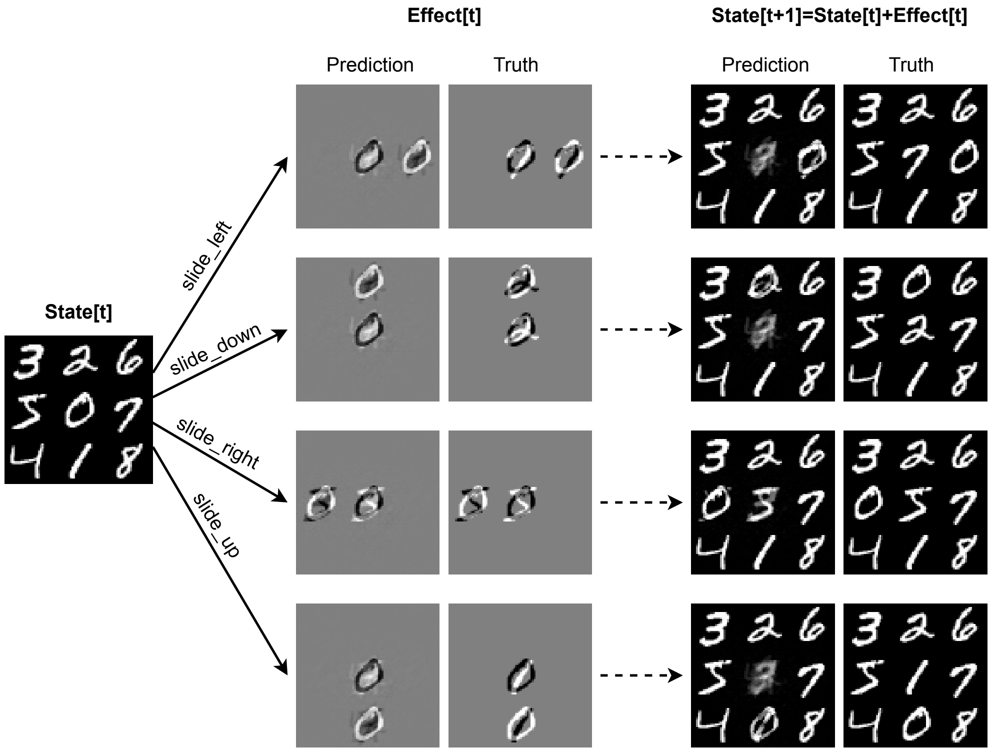


Figure 4.13. Four different effect predictions are shown together with their ground truths for different actions for a given state.

In Figure 4.13, some predicted effects are visualized for a given state and actions together with the ground truth effects. We see that the decoder successfully models the slide of the digit ‘0’. Combining the previous state with the predicted effect, we can have an estimate of the next state shown in the right column in Figure 4.13.

```
(:action slide_left5
  :precondition (and (not (z9)) (not (z5)) (z3))
  :effect (probabilistic
          0.16667 (and (z0) (z1) (not (z2)) (not (z3)) (z4)
                   (z5) (not (z6)) (not (z7)) (z8) (z9)
                   (z10) (not (z11)) (z12))
          0.00758 (and (z0) (z1) (not (z2)) (not (z3)) (z4)
                   (z5) (not (z6)) (not (z7)) (z8) (z9)
                   (z10) (z11) (z12))
          0.68939 (and (z0) (z1) (not (z2)) (z3) (z4) (z5)
                   (not (z6)) (not (z7)) (z8) (z9) (z10)
                   (not (z11)) (z12))
          0.13636 (and (z0) (z1) (not (z2)) (z3) (z4) (z5)
                   (not (z6)) (not (z7)) (z8) (z9) (z10)
                   (z11) (z12))))
```

Figure 4.14. An example PPDDL rule generated from a decision path.

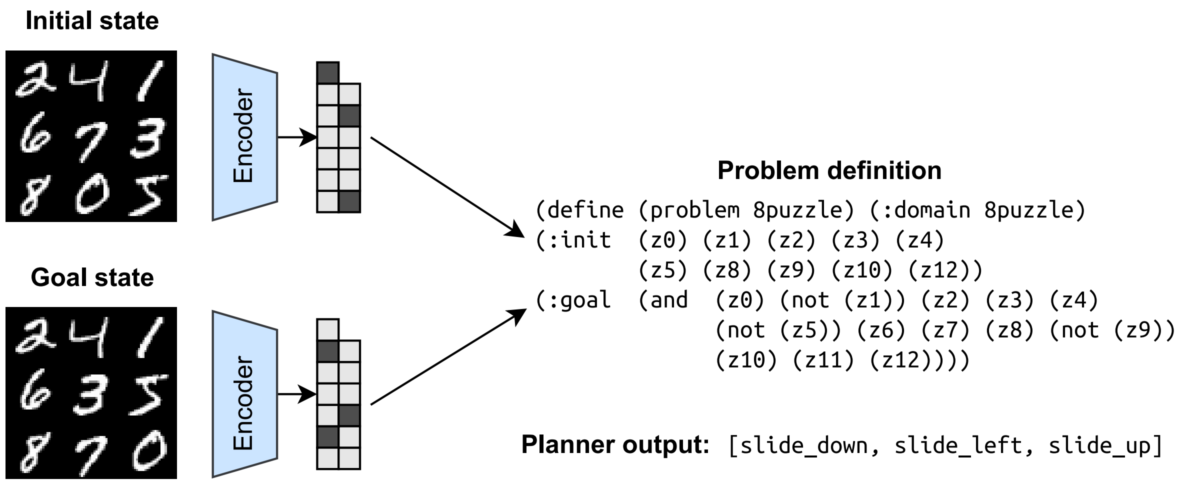
4.5.2. Learned Rules

To train a decision tree for the rule extraction, we collect the set of training examples as follows. Given the current state $\mathbf{x}_t$, the encoder generates the corresponding symbol $\mathbf{z}_t = f(\mathbf{x}_t)$ which is then used as input to the decoder together with the one-hot action vector a_t to predict the effect: $\bar{\mathbf{e}}_t = g(\mathbf{z}_t, a_t)$. Then, we predict the next state $\mathbf{x}_{t+1}$ by summing the predicted effect $\bar{\mathbf{e}}_t$ with the current state ($\bar{\mathbf{x}}_{t+1} = \mathbf{x}_t + \bar{\mathbf{e}}_t$) as in Figure 4.13. Lastly, we use the encoder to generate the symbol $\bar{\mathbf{z}}_{t+1}$ that corresponds to $\bar{\mathbf{x}}_{t+1}$: $\bar{\mathbf{z}}_{t+1} = f(\bar{\mathbf{x}}_{t+1})$. The decision tree is trained with $\{[\mathbf{z}_t; a_t], \bar{\mathbf{z}}_{t+1}\}$ input-output pairs. This is even more generic than robot experiments where we trained the tree with $\{[\mathbf{z}_t; a_t], c_t\}$ pairs (c_t is the effect category predicted by the decoder) since it allows us to express the goal using the image modality. In both cases, the idea is the same: train a decision tree with symbolic input-output pairs to learn probabilistic rules.

As the last step, we convert the decision paths of the tree into probabilistic PDDL rules as in Section 4.4.5. As an example, a translated rule from a decision path is shown in Figure 4.14 where predicates `(z0)` ...`z(12)` correspond to activations of each unit in z. There are no auxiliary predicates as there are in the tabletop environment; the PDDL file only consists of such translated rules given above.

4.5.3. Planning Examples

Using the generated PPDDL description, our system is requested to output a plan for the goal state from a random initial state. For this, the problem definition (where the current state and the goal state are indicated) is created in PPDDL using the activations of the encoder (see Figure 4.15).

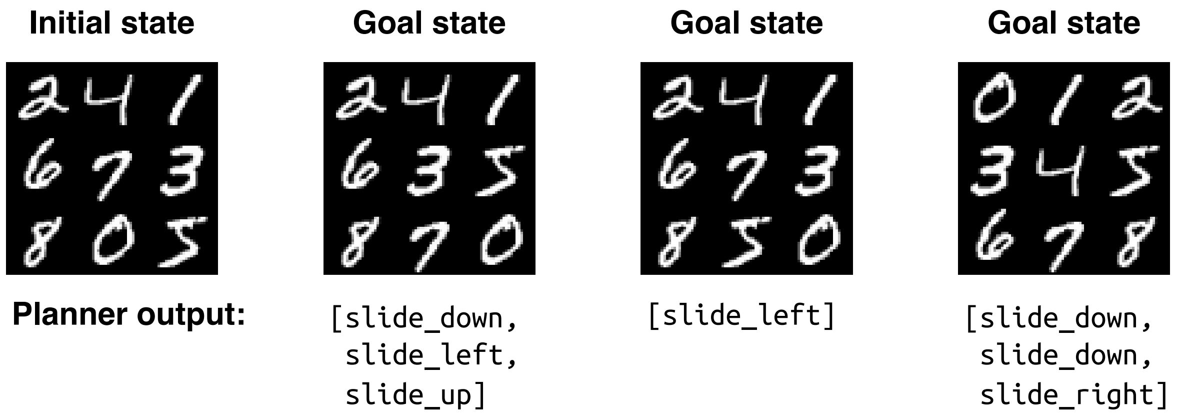


```
(define (problem 8puzzle) (:domain 8puzzle)
(:init  (z0) (z1) (z2) (z3) (z4)
        (z5) (z8) (z9) (z10) (z12))
(:goal  (and  (z0) (not (z1)) (z2) (z3) (z4)
              (not (z5)) (z6) (z7) (z8) (not (z9))
              (z10) (z11) (z12))))
```

```
[slide_down, slide_left, slide_up]
```

Figure 4.15. The generated plan for the goal state.

As shown in the figure, our system was able to find the correct action sequences in order to reach the given goal configuration. Note that we observed that the output plan only slides the tiles so as to move the empty tile into the correct position. This is a consequence of the system because the encoded activations do not represent the global state but a local state: the position of the empty tile and its neighbors. One can extend the locality by incorporating multiple timestep effects of actions [109].



```
[slide_down,
 slide_left,
 slide_up]
```

```
[slide_left]
```

```
[slide_down,
 slide_down,
 slide_right]
```

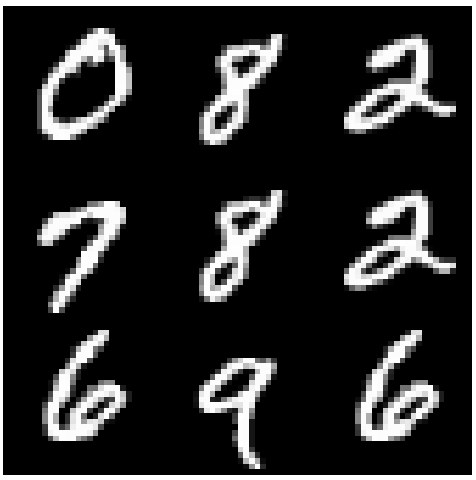
Figure 4.16. Three different goal positions and with planner outputs.

As the current formulation cannot capture the global state, we experimented with local state representations. For example, in Figure 4.16, we set two different arbitrary goals that are one step and three steps away from the initial state (the first and the second goal in Figure 4.16). The planner outputs the correct plan since it can capture the nearby tile information. However, when asked for the third goal in Figure 4.16, the generated plan only moves the empty tile to the correct position while disregarding other tiles. We argue that the global state might be captured by considering relations between the local states of the tiles.

Table 4.4. The average percentage of successful plans that move the empty tile for different plan lengths over five runs.

	1-step	2-step	3-step	4-step
Random plans	24.4 ± 4.2	9.8 ± 2.3	11.4 ± 2.1	10.6 ± 4.0
DeepSym	92.6 ± 5.8	88.0 ± 8.6	88.8 ± 7.4	89.0 ± 7.9

We generated 100 random goal states that are n-step away from the corresponding initial state and reported the planning results to quantitatively assess the performance of the method. We also add the results for executing random actions to assess the performance increment. We report the percentage of plans that successfully move the empty tile to the correct position in Table 4.4. From the results, we see that DeepSym can successfully move the empty tile into the correct position for different plan lengths.


(a)

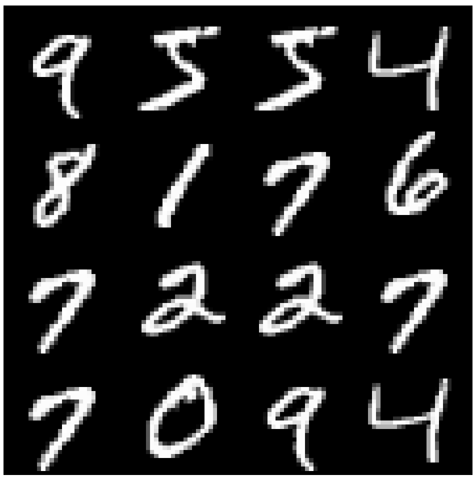

(b)

Figure 4.17. Example configurations for (a) 8-puzzle w/replacement and (b) 15-puzzle w/replacement. In these environments, each digit except ‘0’ may appear more than once.

4.5.4. Scaling-up to 15-puzzle

This section aims to analyze the performance of the system when we scale up the dimensionality of the environment. Examples in the previous section suggest that our system can correctly identify the empty tile, learn the transition based on the empty tile, and make plans to move the empty tile into different positions. We would like to test whether this is the case for larger environments. Therefore, we scale up the 8-puzzle in two different ways: (1) 8-puzzle with replacement (will be denoted as w/r) and (2) 15-puzzle with replacement. Example configurations for each is shown in Figure 4.17. Each digit except ‘0’ (the empty tile) may appear more than once in these versions. We train DeepSym with 14 units for 8-puzzle w/r and with 15 units for 15-puzzle w/r (see Appendix A.3 for details).

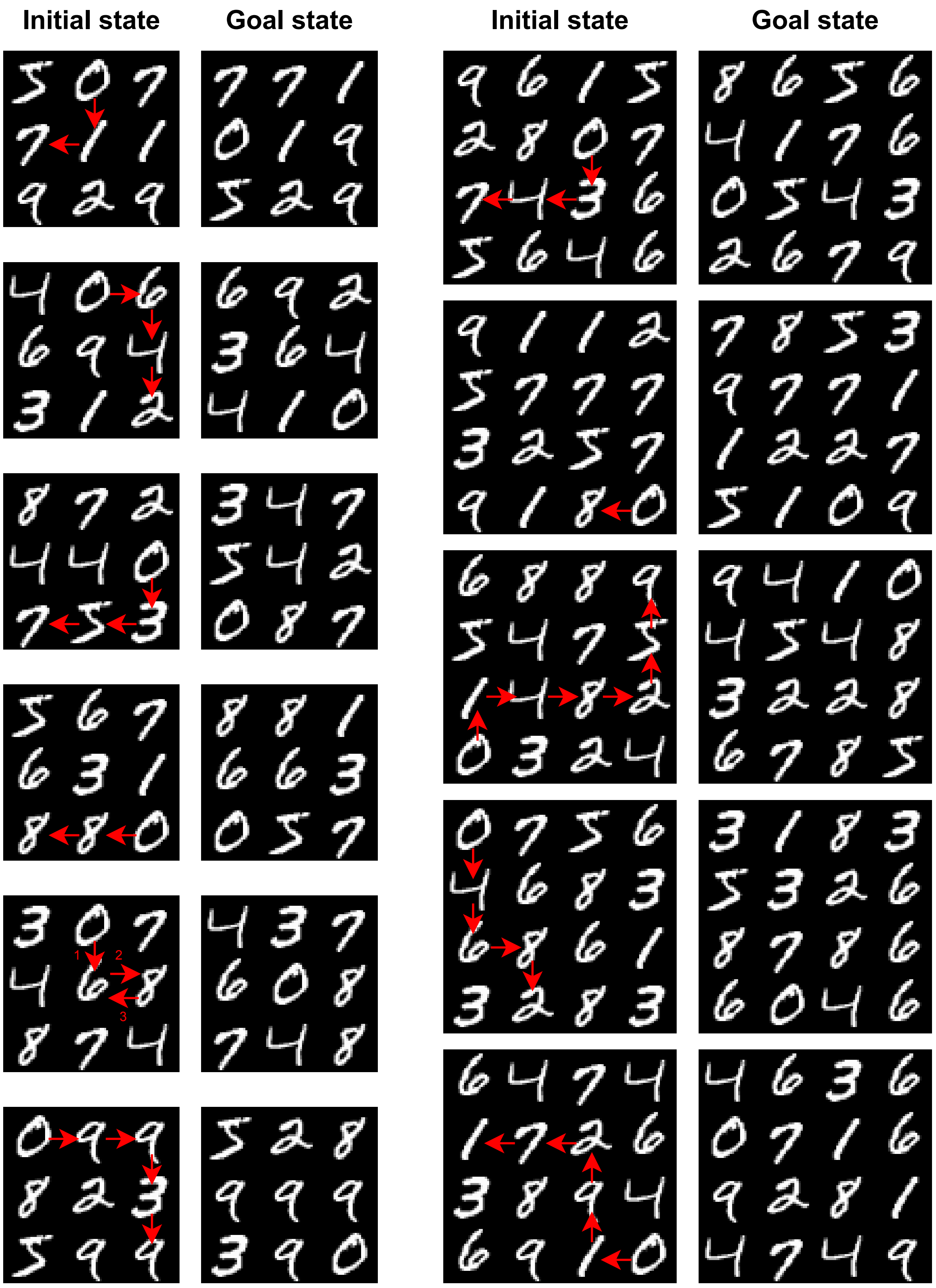


Figure 4.18. Planning results for 8-puzzle w/r and 15-puzzle w/r. The arrow denotes the movement of the empty tile at each step.

The low-level state-space and effect-space are $112 \times 112 = 12544$ dimensional for 15-puzzle w/r while it stays the same for 8-puzzle w/r. We used the same convolutional architecture with different paddings to ensure the same output size. The most frequently activated symbols are shown in Figures A.3 and A.4. We give the planning results for these environments in Figure 4.18. We see that the system moves the empty tile (by sliding other tiles) to the correct position but disregards other tiles.

4.5.5. Comparison with Autoencoder

This section aims to compare DeepSym with an autoencoder baseline. We train an autoencoder (as in [76]) in these three MNIST n-puzzle environments with the same architecture and the same number of bottleneck units as in DeepSym. Given the bottleneck size, it would be impossible for the autoencoder to encode all state space. The most frequently activated symbols for 8-puzzle and 8-puzzle w/r are shown in Figures A.5 and A.6, respectively. We train a decision tree for rule learning using the encoder activations to compare the planning performance. Namely, the decision tree is trained with $(f(\mathbf{x}_t), f(\mathbf{x}_{t+1}))$ input-output pairs where f is the encoder network. After the training, we extracted rules from all paths of the tree and constructed a PPDDL description. The planner failed to produce any plan output for random initial and goal states. This is expected since all the state space cannot be encoded, and therefore, some states are not represented correctly in the PPDDL description. One would need to increase the bottleneck size in order to convert all the state space into PPDDL descriptions. In [76], the bottleneck size is set to 25 units (instead of 13 in our experiments) to cover the state space.

4.6. Discussion

A plan corresponds to a sequence of actions to move from an initial state to a goal state. One must chain the effects of actions to predict a future state. Thus, the effects of actions should be known to generate a plan. Therefore, the capability of knowing the preconditions of actions and predicting the effects of actions is a requirement for generating a successful plan [39]. Our system realizes this requirement with a deep neural network trained to predict effects of actions.

The main difference between DeepSym and approaches that focus on compressing the state representation (e.g., with an autoencoder, [76], [78], or with a world model, [41]) is that the learned representations in DeepSym are only due to actions and effects of the agent [71]. Learning symbols based on the capabilities of the agent allows one to filter out details of the environment not related to the agent. On the other hand, the approach of compressing the state representation brings its own advantages. One can use a large dataset of states to pre-train an unsupervised model to learn a compact model of the environment and then use the learned model to train a supervised model for planning or policy learning.

Finding action-independent discrete representations is non-trivial in a large state space, even for the toy examples given in Section 4.5. In our robot experiments, the autoencoder with discrete units [76] was shown not to generate a useful representation with a low bottleneck dimension. On the other hand, DeepSym can learn useful and compact representations for planning as it considers actions and effects. For environments that are more realistic for lifelong learning, such as Minecraft [110], the raw state-space is virtually infinite, making it difficult to find a minimal set of meaningful discrete representations without taking actions and action effects into account. On the other hand, action- and effect-based learning allows for an efficient representation of the state space by filtering out the aspects of the environment not relevant to the actions of the agent. For example, in the 8-puzzle environment, the encoder disregards the tiles not near the empty tile since the generated effect does not depend on them. The learned representation allows for generating plans to move the empty tile to different positions. DeepSym learns the minimal set of representations that are needed for the effect prediction of action. Therefore, our system learns action-centric representations, i.e., representations that involve the empty tile. State- and action-based methods are two different (possibly complementary) approaches with advantages and disadvantages. For example, if the problem domain is small, or there exists a large-scale pre-trained model of the environment, encoding all the state space will allow one to solve any encountered problem. However, this approach might be infeasible for larger domains. On the other hand, action-based encoding learns the minimal set of symbols to predict effects at the cost of missing possibly global task requirements (e.g., a specific arrangement in 8-puzzle).

It is theoretically possible to learn a simpler feature-based representation that will be more computationally efficient when compared with deep networks when state, action, and observed effects are all known [42], [73]. However, this approach would need manual feature extraction for newly encountered domains, while a differentiable network that can be automatically tuned offers a more uniform and extendible approach.

One thing we observed is that with the narrow bottleneck size (i.e., 13 units for 3248 configurations), the encoder does not represent all the neighbor configurations that are needed for effect prediction. However, when we increase the bottleneck size, the encoder indeed learns all the necessary configurations. Even if the system successfully encodes all the local states, it would still need to symbolically encode the global state to solve the task globally. One approach to encoding the global state might be extending the locality by considering the effects of multiple timesteps [109].

4.7. Conclusion

In this chapter, we introduced a method that discovers effect- and action-guided object categories, encodes them as discrete symbols, and learns rules that predict action effects. It sustains a general cognitive development progression where symbols are formed, rules are learned, planning is achieved, and verified in execution. Our system contributes to the state-of-the-art by showing the following desirable properties that have not been achieved/shown simultaneously elsewhere. We proposed a generic, single pipeline neural solution for mapping raw sensorimotor experience into the symbolic domain. The proposed network allows progressive learning of increasingly complex abstractions, exploiting previously learned abstractions as inputs. It is gradient-friendly, so it can be incorporated into any gradient-based machine learning system for more complex processing. When compared with the continuous bottleneck layer version of our system, i.e., OCEC, our system performs better in effect category formation, leading to more successful action planning. This suggests that instead of post-training clustering of the continuous unit outputs, employing discrete units from the beginning is beneficial.

In this chapter, networks take a fixed number of objects as input (i.e., a single object for push and two objects for stack) and predict their effects when an action is applied. However, in the general case, we might not know in advance the number of objects that an action is going to affect. In the next chapter, we extend the architecture to handle the effects of a varying number of objects by taking into account their relations in the environment using self-attention layers [26].

5. LEARNING MULTI-OBJECT SYMBOLS WITH ATTENTIVE DEEP EFFECT PREDICTORS

5.1. Introduction

In the previous chapter, we have shown that the symbols formed with single-object interactions can be used to bootstrap new symbols or rule formation while interacting with two objects. However, the transition from a single object to two objects required the construction of a new neural network. Furthermore, it was not possible to learn symbols from interactions that involve a varying number of objects, some of which may affect the action execution and others not.

In this chapter, we aim to remove these limitations by acquiring a single neural system in order to discover symbols based on the sensorimotor data generated by the robot when it interacts with multiple objects with arbitrary multiplicity. To this end, we propose a deep neural architecture that includes self-attentive layers [26] with binary latent representations. We show the validity of the proposed architecture in a simulated manipulation scenario where a robotic arm interacts with object(s) while building symbolic representations. Most importantly, our system not only learns object-specific symbols but also learns multi-object symbols that are formed on the fly by automatically processing the related object symbols through the self-attention mechanism. After learning, we investigated the formed symbols and observed that they are effective in making effect predictions. We showed that the learned symbols enable reasoning capabilities with multiple objects that may influence the interaction dynamics in various ways.

5.2. Methods

The proposed architecture is shown in Figure 5.1. This architecture predicts the effects of an executed action at a given state. The architecture consists of an encoder

$f(x)$ with a binarized bottleneck layer, a decoder $g(x)$, and an attentive module $a(x)$. We assume that we have an image segmentation module for cropping and tracking objects through several timesteps. While this is a strong assumption, state-of-the-art segmentation methods in computer vision can segment and track objects in complex environments [111, 112].

In the first part, the high-dimensional continuous state x is first partitioned into segments $(x_1, x_2, \ldots, x_k)$ using the segmentation module. As we expect to obtain relational symbols that potentially encode information related to several objects, the relative positions of the objects are important and need to be preserved while processing image segments. In order not to lose the location information, here we concatenate the location of each pixel in the cropped image as two additional channels. Then, each segmented image (x_i) is given as input to the encoder. The encoder consists of several convolutional layers for processing high-dimensional visual input. For binarization at the last layer of the encoder, we use the Gumbel-sigmoid function [51, 52] for gradient backpropagation. In the end, the encoder outputs a binary vector z_i for each segment x_i. These binary vectors (z_i) are expected to encode symbols that can encode one or more objects and that are effective in predicting the effects of actions and, therefore, multi-step planning.

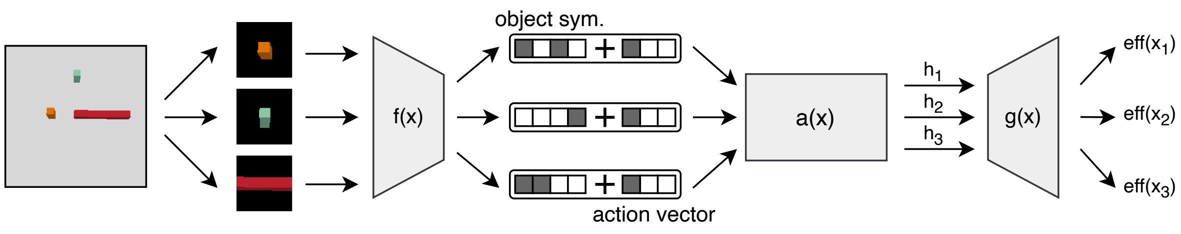


Figure 5.1. Attentive DeepSym architecture.

Each action is represented with a one-hot binary vector. Actions are assumed to be high-level parameterized motion primitives. The action vector a is concatenated with each z_i separately. Concatenated vectors $((z_1, a), (z_2, a), \ldots, (z_k, a))$ are given as input to the attentive module. The attentive module consists of several self-attentive layers [26] which allows the generated symbols $(z_1, z_2, \ldots, z_k)$ to interact with each other to generate $(h_1, h_2, \ldots, h_k)$ that should hold relational information for accurate effect prediction. Note that we assumed the existence of action primitives in this chapter. These primitives can be learned with different motion primitive methods [100], [113,114] and transferred from previous stages of development as we previously showed [43].

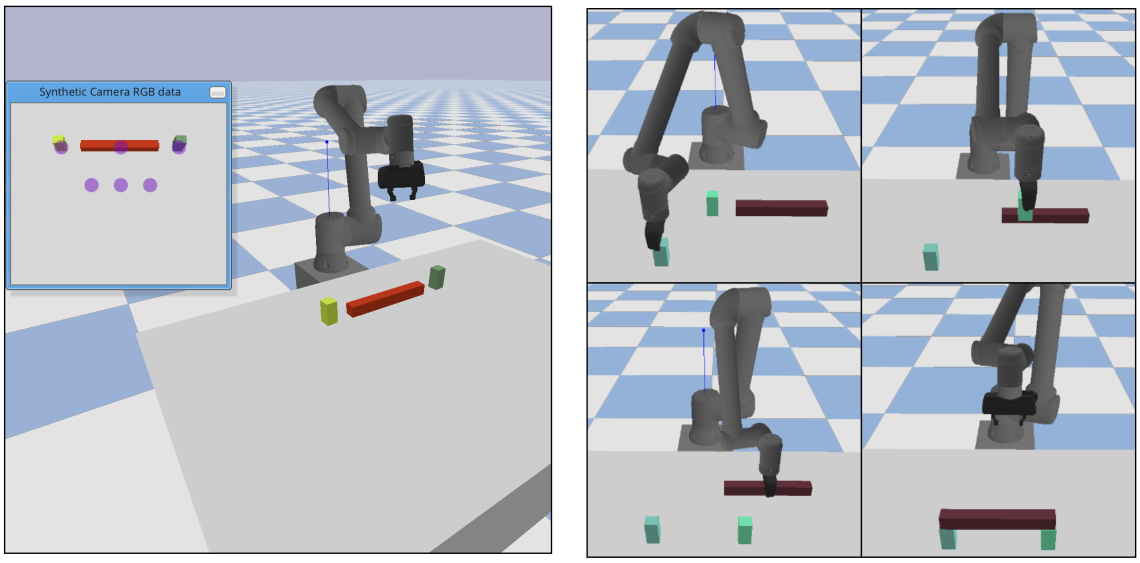


Figure 5.2. The experiment setup. Six possible pick and release locations are shown in purple in the synthetic camera image in (a). An example exploration with three objects is shown in (b).

The self-attention operation allows each symbol to attend to other symbols to form a new representation. Since the whole model is trained in an end-to-end fashion, symbols are formed in such a way that they not only define the characteristics of the cropped object but also contain information about relations with other objects. For example, if a long stick is picked and released to a different location, and if there is a small cube on top of the stick, then the position of the cube changes as well. To accurately predict such relational effects, the model needs to encode information regarding relations between objects. Self-attention seamlessly combines this information using query-key-based attention operation [26].

As the last step, the decoder function predicts the generated effect for each segment. In our case, we represent the effect $\mathrm{eff}(x_i)$ as the position displacement of the ith object after the action. In our experiments, we directly retrieve the position displacement from the simulator. However, one can use a more generic effect, such as the change in pixel values [46]. The learned symbols heavily depend on the effect representation, and we treat effect representation as a separate problem. This chapter, instead, focuses on creating an architecture that can learn symbols for a varying number of inputs and predict action effects that would require processing relational information between objects.

5.3. Experiments

5.3.1. Experiment Setup

The experiment setup is shown in Figure 5.2. This is a tabletop environment where a UR10 robot arm picks up and releases objects at six pre-defined locations shown in Figure 5.2a. Given six pick-up and six release locations, the total number of high-level actions is 36. An example sequence of action executions is shown in Figure 5.2b. The environment is initialized with one to three objects initially located at the row closer to the robot body (see the top three purple dots in Figure 5.2a). The robot perceives its environment with a depth camera located on top of the table. The camera takes 256×256 pixels depth image. We assume that the robot has a segmentation module that can crop and track the movement of objects. This can be achieved with state-of-the-art slot-based models [111, 112]. In this chapter, we use the segments provided by the simulator. An example crop is shown in Figure 5.1–left. All crops are 64×64 pixels. Finally, in order to preserve object locations, the x and y locations of each pixel in the cropped image are concatenated as two additional channels.

The robot collects the interaction data set as follows. The environment is initiated with $k \in \{1, 2, 3\}$ objects where k is set randomly. The initial depth image x_i of the environment, together with its segmentation s_i, is recorded before taking any action. Then, a random action a_i is executed and position displacements of k objects $e_i = ([\Delta x_1, \Delta y_1, \Delta z_1], \ldots, [\Delta x_k, \Delta y_k, \Delta z_k])_i$ are recorded as the generated effects. Here, x and y represent the pixel coordinates of the object's center of mass, and z represents the depth value obtained from the depth camera for the corresponding object. Note that the length of e_i depends on the number of objects. In total, 12,000 (x_i, s_i, a_i, e_i) tuples are recorded as the interaction data set. We use 10,000 samples for training, 1,000 samples for validation, and the remaining 1,000 samples for testing.

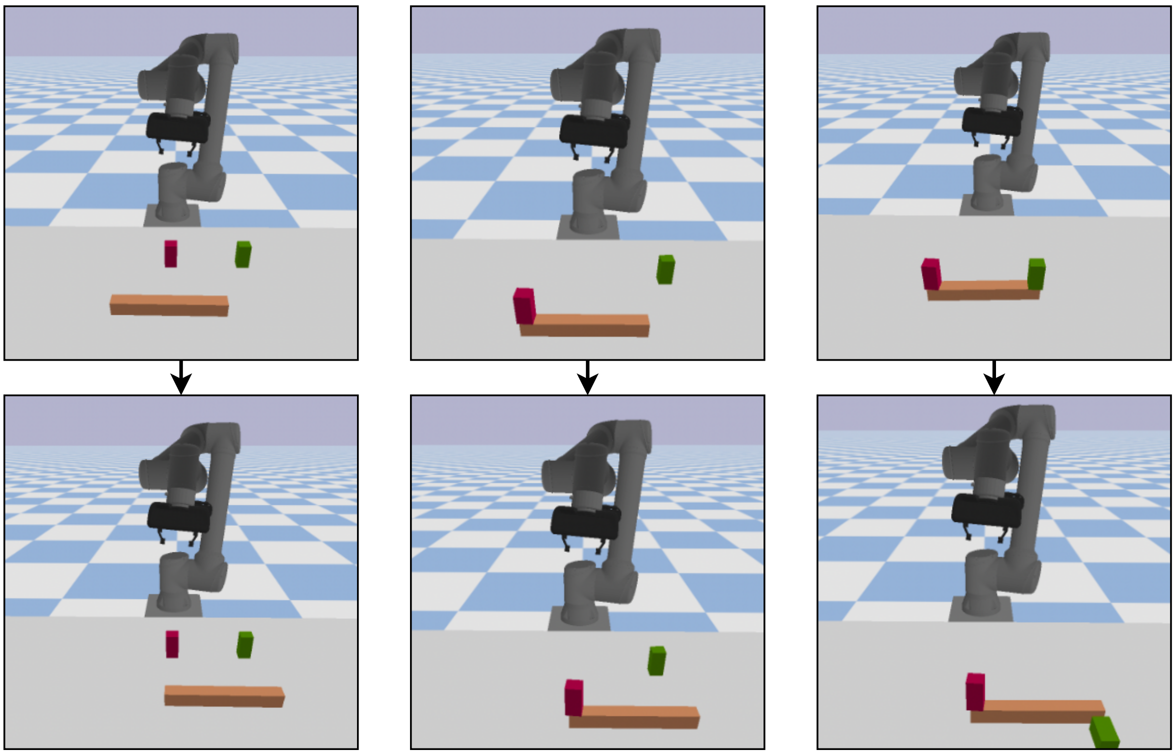


Figure 5.3. Effect prediction for different states.

We train the architecture in Figure 5.1 to predict the generated effects e_i of an action a_i. The encoder $f(x)$ consists of four convolutional layers with 64, 128, 256, and 512 filters. Convolutional layers are similar to that of DCGAN [108] with a kernel size of four, a stride of two, and a padding of one. After convolutions, we take an average across height and width dimensions and project this fixed-size vector into an 8-dimensional vector. Lastly, we binarize the activations using the Gumbel-sigmoid function for backpropagation [51, 52]. The self-attention module $a(x)$ is a transformer with four transformer encoder layers [26]. We use the default settings in PyTorch for transformer layers [115]. The decoder $g(x)$ is a multi-layer perceptron (MLP) with three hidden layers, each containing 256 units. We use batch-normalization [8] in the encoder and the decoder to increase the convergence speed. We train all modules in an end-to-end fashion with Adam optimizer [7].

5.3.2. Effect Prediction

Our system predicts the continuous effect, i.e., continuous pixel position and depth changes of objects, from learned discrete symbolic activations. It is neither our aim nor possible to minimize the prediction error to zero, yet we need to make sure that our system can make discover symbols that can make fairly good effect predictions. Therefore, we analyzed the prediction error. After training, the mean effect prediction errors on the test set are 8.6mm, 15mm, and 6mm for x, y, and z dimensions, respectively. For comparison, the average changes of the corresponding dimensions of objects in the effect set are 50mm, 68mm, and 10mm. As a result, we can conclude that our system discovered symbolic representations that are effective in predicting the effects of actions on single or multiple objects.

Next, we investigate whether our system discovered symbols that automatically include information from action-relevant objects in multi-object settings and model interaction dynamics of these objects. For this, we created a scenario where the action applied to an object is the same, but the other objects in the environment are arranged in different ways such that different effects are expected to be obtained and are expected to be correctly predicted by our system. The action is to pick up a long stick from one position and release it to another position, and two other objects are placed in different configurations, as shown in Figure 5.3. In Figure 5.3, three different example interactions experienced by the robot were shown in three columns. The initial snapshot of each interaction is shown in the upper row, and the final snapshot after action execution is shown in the bottom row. In all three cases, the robot picks up the orange stick, moves it to one position right, and releases it (see the bottom-right purple position in Figure 5.2a). Due to different initial configurations of the red and green blocks, different effects are (expected to be) observed.

Table 5.1. Effect predictions for example cases in Figure 5.3. Units are in millimeters.

	Green Cube	Orange Stick	Red Cube
Predictions			
Case 1	(3, 6, -2)	(-2, 192, -2)	(4, -9, -1)
Case 2	(3, 8, 0)	(-17, 196, 1)	(-10, 180, -5)
Case 3	(-3, 210, -45)	(4, 159, 8)	(-30, 131, -20)
Ground truth			
Case 1	(0, 0, 0)	(0, 180, 0)	(0, 0, 0)
Case 2	(0, 0, 0)	(0, 180, 0)	(-1, 176, 0)
Case 3	(102, 177, -75)	(3, 180, 0)	(2, 176, 0)

The effects predicted by our system, along with ground-truth effect values, are provided in Table 5.1 for each interaction (case). The results show us that our system was able to model relational information between objects and, therefore, made correct predictions. For example, in Case 1, the system correctly predicted that only the position of the stick changes. In Case 2, the position of the red cube also changes along with the stick. And finally, in Case 3, the positions of both cubes change along with the stick (in the same direction). This shows that our system learned symbols that enable it to make high-level reasoning that involves multiple objects, such as “objects on top of another object will move together with the object below”. We conclude that self-attention in transformer layers indeed helps the binary symbols interact with each other to predict the correct effect. In the previous chapter, DeepSym, the modeling of such interactions was only possible at the input level by manually concatenating the necessary object crops with a pre-defined number of objects. Here, due to the self-attention layers, the model discovers symbols that automatically use the binary activations of the related objects and disregards the activations of unrelated objects.

5.3.3. Learned symbols

The number of binary units in the post-encode bottleneck layer is set to eight units; therefore, there can be at most 256 unique symbols. As we analyzed the activation frequency of these symbols, we observed that the most frequently activated 35 symbols cover 95% of the training set. The prototypical values, i.e., the average states from samples that activate these 35 specific object symbols, are shown in Figure 5.4. Only the symbols that are activated more than 100 times out of 28,607 samples (the number of objects in 10,000 samples) are shown in the figure. As shown, these object symbols encode the location of the object, the depth of the object, and the occlusion information.

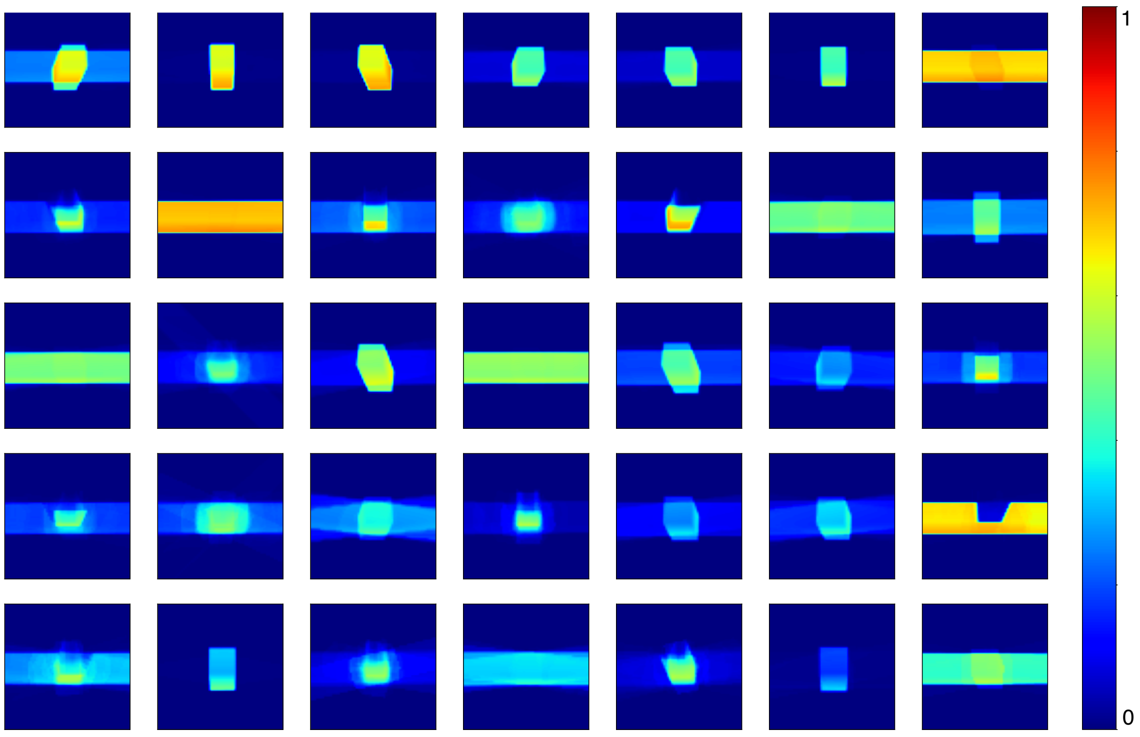


Figure 5.4. The average depth images for each symbol are sorted by their activation count. The color bar shows the normalized depth value.

While a number of symbols are activated only for sticks or only for cubes, others are activated for both cubes and sticks. For example, the symbol in the second row and the first column is activated for both cubes and sticks (see samples in Figure 5.5). This makes sense since the generated effect is the same in both cases; the robot cannot grasp the occluded object and, therefore, cannot change its position. We can interpret this as a 'not graspable' symbol. However, note that there is not a specific symbol for 'not graspable stick' or 'not graspable cube' since it does not bring any additional advantage in terms of effect prediction accuracy; the model can already predict the generated effect (which is 0mm, 0mm, 0mm on average) without differentiating these two objects. We can conclude that our system learned a minimal set of symbols that is required to make predictions and reasoning, and these symbols were not only determined by the available objects in the environment but also by the action capabilities of the robot.

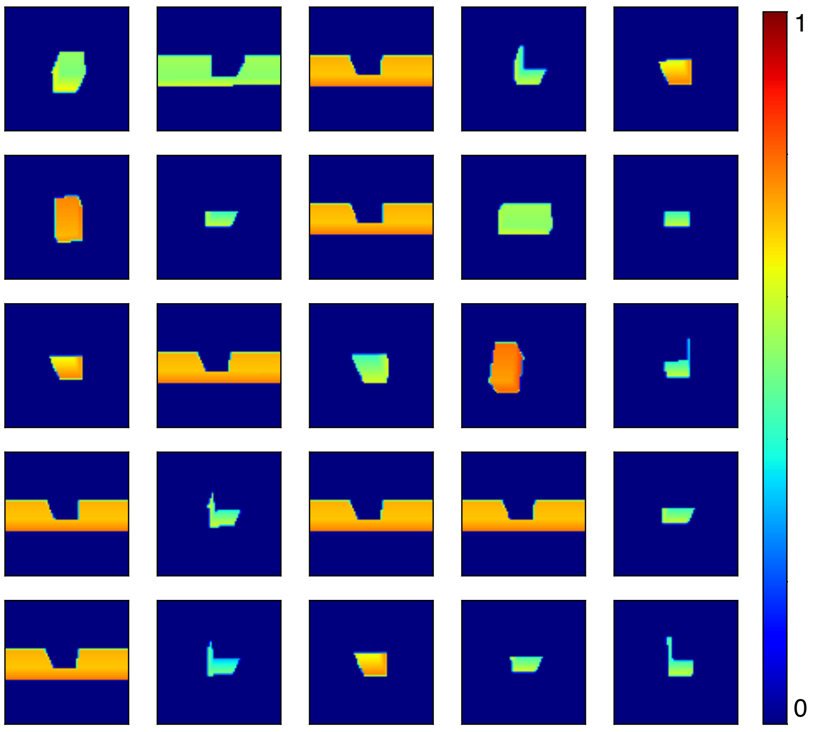


Figure 5.5. Example images that activate the symbol '00011011'.

5.3.4. Planning with Discovered Symbols

In this section, we investigate the suitability of symbolic representations in making multi-step plans. Although our system can predict the effects on objects given the tabletop image and the action, here, we aim to acquire a full symbolic reasoning capability by learning and exploiting a transition model that predicts the next symbolic state given the current symbolic state and the action. For this, we train a separate network $m(z)$ that predicts the next symbolic state given the current symbolic state and the action (see Figure 5.6) using the discovered symbolic representations that were presented in the previous sections. This network consists of two dense layers followed by a single self-attention layer and another two dense layers. We train the network with binary cross-entropy loss for each dimension.

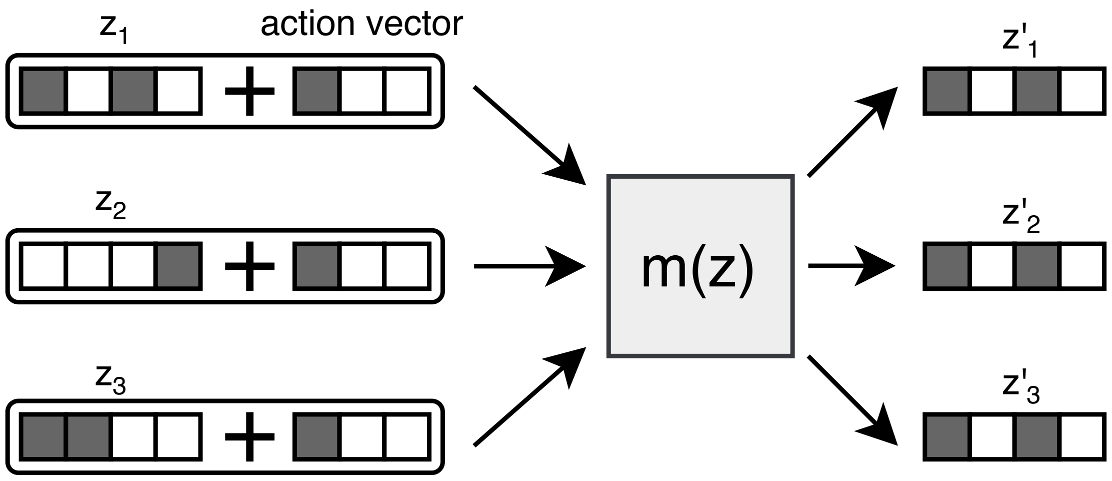


Figure 5.6. Symbolic forward module. Given the symbolic representation of the current state, this module directly predicts the symbolic representation of the next state.

Such a network, after training, allows us to search for a goal symbolic state using any tree search algorithm [102]. Figure 5.7 shows a plan generated and executed in order to achieve a composite structure given as a goal. As shown, the learned symbolic transition model could be used to generate feasible plans and, when executed, could achieve the desired goal, indicating that multi-step predictions in the generated long-horizon plan were correct. This example shows that the discovered symbols can be used for training a forward symbol prediction model that allows for achieving goals via tree search.

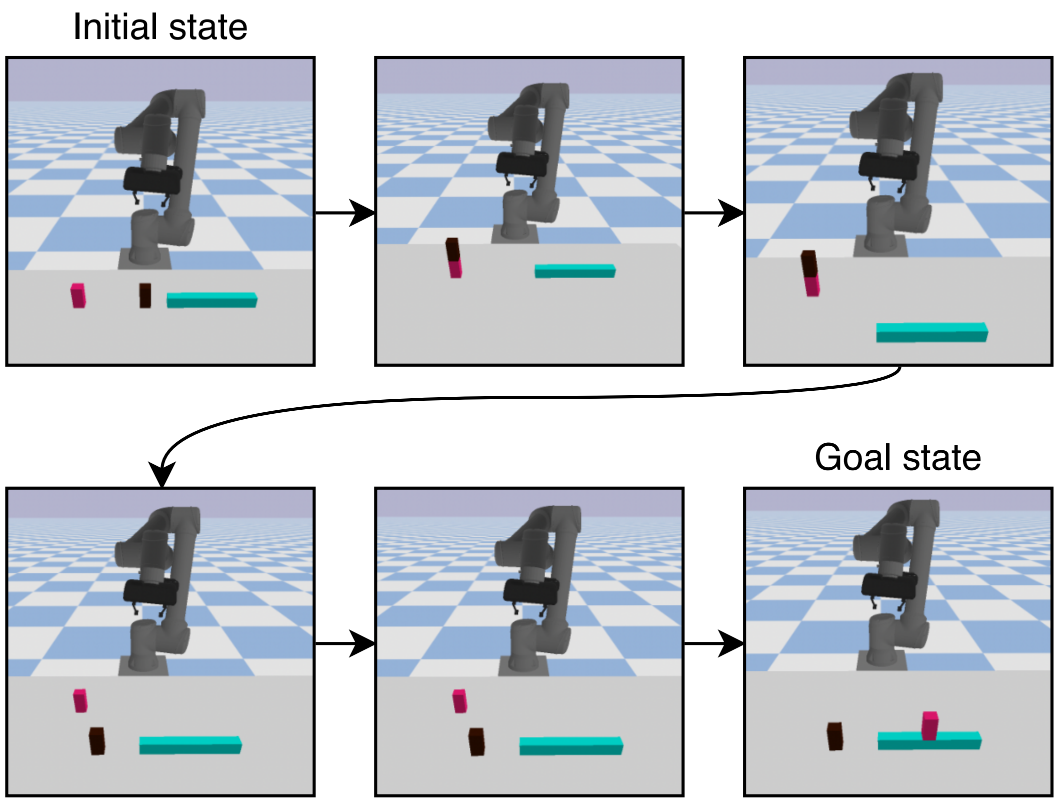


Figure 5.7. An example plan for an arbitrary goal state. In the fourth action, the robot tries to pick and release from the top-right position to the bottom-left position, which does not interact with any object.

5.4. Conclusion

In this chapter, we proposed and implemented a predictive encoder-decoder network that utilized a binary bottleneck layer and, importantly, a self-attention mechanism in order to discover symbols relevant for interacting with the environment of the robot that hosts a varying number of objects. Through experiments with a simulated manipulator robot, we showed that the robot acquired reasoning capabilities to encode interaction dynamics of a varying number of multiple objects in different configurations using the discovered symbols. For example, when queried, the robot could reason that (possibly multiple numbers of) objects that are on top of another object would move together if the object below is picked up, and the objects around would not move. We showed that these reasoning capabilities were acquired by learning a minimal set of symbols that are optimized for effect prediction in the ecological niche of the robot. We also showed that the discovered symbols can be used for planning by training a higher-level neural network that makes pure symbolic reasoning.

Even though the learned symbols allow for planning with a neural network, our initial aim was to build rules from symbolic transitions that can be used for domain-independent planning. Here, we cannot easily build rules as we do not have direct access to relations between objects or symbols. For example, after training, we can transform a set of pre- and post-conditions into symbolic transitions. However, these transitions only make sense in the presence of relations between symbols; we would need symbols that interact with each other so that we can successfully build rules without overfitting them to data. One possible solution might be to deduce relations based on the attention weights for each object. Yet, our preliminary experiments showed that these values are generally distributed over multiple objects, and when we combine attention weights for multiple layers, values are generally uniformly distributed over all objects.

As we realized the main block prevented us from building rules defined over the learned symbols, which is the lack of knowledge about relations between symbols, in the next chapter, we explicitly build the architecture to output attention weights as binary values, which are used as relational symbols between objects.

6. DISCOVERING RELATIONAL OBJECT SYMBOLS WITH SYMBOLIC ATTENTIVE LAYERS

6.1. Introduction

In Chapter 4, DeepSym [46], we combined the two motivations: learning preconditions and effects of actions with deep neural networks. In DeepSym, an encoder-decoder network with a discrete bottleneck layer is trained to predict the effect of actions (Figure 6.1 – bottom left). However, the network can only handle a fixed number of object interactions, restricting the types of relations that can be learned. This restriction is lifted in Chapter 5, Attentive DeepSym [53] by introducing a self-attention mechanism [26] to the architecture (Figure 6.1 – bottom right). As symbols interact with each other using self-attention, the network can make accurate predictions for related objects (e.g., on top of each other). Although this architecture is effective in making accurate predictions for related objects through the learned multi-object symbols, it does not reveal the explicit relations between objects. Furthermore, as the self-attention layer is applied after discretization, the relational representational capacity of the model is limited by the learned symbols.

In this chapter, we explicitly compute discrete self-attention weights from object features and treat them as relational symbols between objects. Using these discrete relations, we fuse object symbols in an aggregation function to produce a single representation for each object, which is then used to predict the observed, potentially multi-object effect. This results in a more powerful architecture, which we named Relational DeepSym, that can explicitly output the relations between a varying number of objects while enjoying other properties of DeepSym. Our experiments in a simulated tabletop scenario show that (1) Relational DeepSym achieves lower errors than DeepSym [46] and Attentive DeepSym [53] for different numbers of objects and actions, (2) learns not only object symbols but also relational symbols. The rest of the chapter is organized as follows: the problem definition and our assumptions are given in Section

6.2, the proposed model is explained in Section 6.3.1, and the differences with previous DeepSym architectures are discussed in Section 6.3.2.

6.2. Problem Definition

From a developmental learning perspective, this chapter starts off with a basic sensorimotor system [43], [116], where the robot can locate objects and pick and place them on top of each other.

Consider an environment represented by a set of object features $O = \{o_1, \dots, o_n\}$ where $o_i \in \mathbb{R}^{d_o}$ is a d_o-dimensional continuous-valued vector for the ith object's features and n is the number of objects which can vary through multiple environment instances. Without any loss of generality, we define object features as the combination of (1) the object type (e.g., short block, long block), (2) the position (x, y, z) and the orientation (x, y, z, w) of the object, resulting in a total of eight dimensions for each object. However, the method can be applied to any other modality type (e.g., images, point clouds) by modifying the networks accordingly as long as the environment state can be partitioned into a set of objects.

In our experiments, the robot has a single type of high-level action with different parameterizations: `pick-place`$(o_i, \mathit{\Delta} y_i, o_j, \mathit{\Delta} y_j)$ where o_i and o_j are the objects to be picked up and the target object, respectively, and $\mathit{\Delta} y_i$ and $\mathit{\Delta} y_j$ are the y-axis pick and release positions relative to the center of the object (Figure 6.2a). $\mathit{\Delta} y_i$ and $\mathit{\Delta} y_j$ can take discrete values of $\{-1, 0, 1\}$, which correspond to 7.5cm left, center, and 7.5cm right of the object center, respectively. This results in a total of $9n^2$ grounded actions (i.e., actions with parameters) for n objects. The robot randomly picks a grounded action, executes it in the environment, and observes the new environment state as $O' = \{o'_1, \dots, o'_n\}$ where $o'_i \in \mathbb{R}^{d_o}$ is the new object features for the ith object.

The goal is to transform the state vector $O = \{o_1, \dots, o_n\}$ where each $o_i \in \mathbb{R}^{d_o}$ is a feature vector describing object i into a set of object symbols $Z = \{z_1, \dots, z_n\}$ and relational symbols $R_k = \{r_{11}^{(k)}, \ldots, r_{nn}^{(k)}\}$ where $z_i \in \{0,1\}^{d_z}$ is a d_z-dimensional binary vector (i.e., an object symbol) for the ith object and $r_{ij}^{(k)} \in \{0,1\}$ is a binary value for the kth relation between the ith and jth objects. Once we have a symbolic representation Z, R of a given state O, we can transform the continuously represented interaction data $\{O^{(i)}, a^{(i)}, O'^{(i)}\}_{i=1}^{N}$ into its symbolic counterpart, $\{(Z^{(i)}, R^{(i)}), a^{(i)}, (Z'^{(i)}, R'^{(i)})\}_{i=1}^{N}$, and learn a set of symbolic transition rules $(Z, R) \xrightarrow{a} (Z', R')$ enabling domain-independent planning with AI planners to achieve a goal state [39, 42], [46], [81], [117].

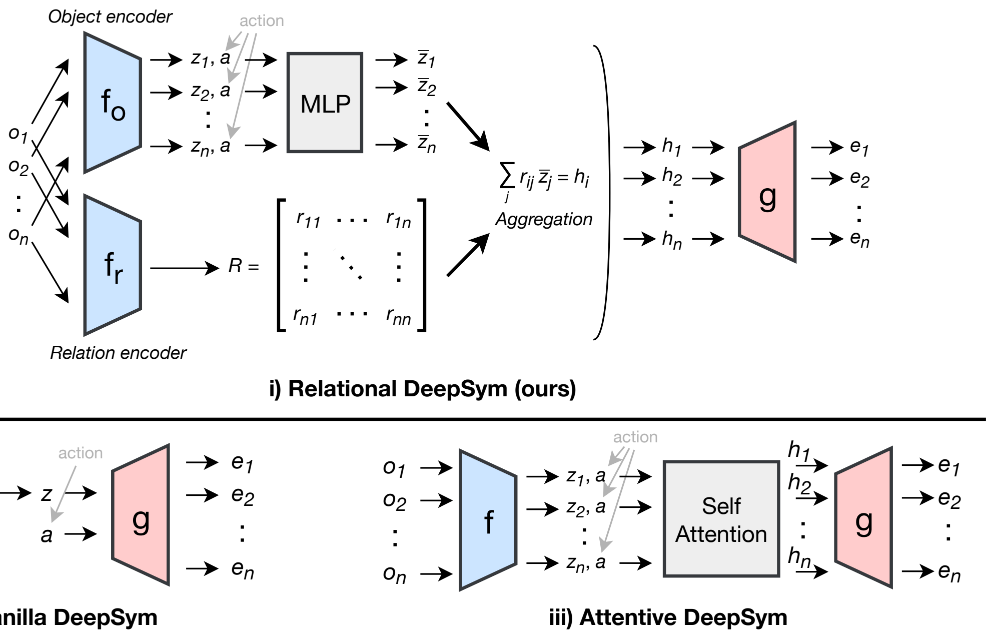


Figure 6.1. The proposed model is shown in the top panel. For comparison, we also provide high-level outlines of DeepSym [46] and Attentive DeepSym [53] in the bottom panel in (ii) and (iii), respectively.

To learn object symbols and relations between objects, we follow an objective similar to previous studies [42], [46], [53] and train a model to predict the effect $E = \{e_1, \ldots, e_n\}$ of the executed action a where $e_i = \delta(o'_i, o_i)$ is defined as the cartesian position difference between o'_i and o_i before and after the execution of the action a. In our experiments, we compare the effect prediction performance of the proposed model with two related models, DeepSym [46] and Attentive DeepSym [53], in a simulated tabletop environment.

6.3. Methods

6.3.1. Relational DeepSym

The top panel in Figure 6.1 shows a high-level overview of the proposed model. The model consists of four main components: (1) an object encoder f_o that learns object symbols, (2) a relational encoder f_r that learns relational symbols between objects, (3) an aggregation function that combines information from multiple objects by multiplying object symbols with relational symbols, and (4) a decoder g that predicts effect e_i of the executed action a for each object i. As the whole architecture is differentiable and trained in an end-to-end fashion to minimize the effect prediction error, we expect the object and the relational encoders to learn to predict symbols and relations, which is useful for the decoder to predict the effect.

To output a discrete vector without removing the differentiability, the activation of the last layer is set to the Gumbel-sigmoid function [51, 52]. The Gumbel-sigmoid function approximates a Bernoulli distribution by injecting noise drawn from the Gumbel distribution to the logits, which forces the model to output in the extremities (i.e., either very low or very high values) to send a signal in the presence of noise. The object encoder f_o outputs a binary vector z_i for the ith object given its features o_i:

$$z_i = f_o(o_i). \tag{6.1}$$

In our experiments, f_o is a multi-layer perceptron; however, other differentiable architectures can be used for different modalities (e.g., convolutional layers to process images)

The relational encoder takes object features $\{o_1, \dots, o_n\}$ as input and processes them independently to output query and key vectors $\{(q_1, k_1), \dots, (q_n, k_n)\}$ for each object. As in the object encoder, the relational encoder is a multi-layer perceptron with two different outputs for the query and the key. Note that multiple heads $R_1, R_2, \dots, R_k$ can be used to model different relations between objects. Let Q and K be $n \times d$ matrices, each row containing a query vector q_i and a key vector k_i, respectively. Then, the attention weight matrix R with size $n \times n$ where r_{ij} is the (directed) relation between objects i and j—treated as relational symbols in this chapter—are computed as follows

$$q_i, k_i = f_r(o_i) \quad \forall i \in \{1, 2, \ldots, n\} \tag{6.2}$$

$$R = \text{GumbelSigmoid}\left(\frac{QK^T}{\sqrt{d}}\right), \tag{6.3}$$

where d is the dimensionality of the query and key vectors. This is slightly different from the regular self-attention function [26] in which the softmax function is used instead of the Gumbel-sigmoid function. This modification creates two different behaviors: (1) the use of a sigmoid function instead of a softmax function allows multiple attention weights to different objects to be active at the same time (whereas in softmax, attentions compete with each other), and (2) the use of the Gumbel-sigmoid function discretizes the attention weights while preserving differentiability, allowing us to treat the weights as *relational symbols* between objects.

In the third step, the aggregation function combines object symbols $\{z_1, \ldots, z_n\}$, relational symbols $\{R_1, \ldots, R_k\}$, and the executed action a to produce a single representation for each object. The aggregation function has the following steps:

$$a_i = [(1, \Delta y_i)_{o_i=\text{pick}}, (1, \Delta y_i)_{o_i=\text{place}}] \tag{6.4}$$

$$\bar{z}_i = \text{MLP}(z_i, a_i) \tag{6.5}$$

$$H^j = R_j \bar{Z} \tag{6.6}$$

$$H = (H^1, H^2, \ldots, H^k) \tag{6.7}$$

for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, k\}$, where object symbols and the action vector are concatenated in Equation (6.5), and the aggregation occurs in Equation (6.6). To concatenate action-specific information to each object symbol z_i in a permutation-invariant way, we define a 4-dimensional vector a_i in Equation (6.4) for each object i in which the first and the third dimensions are set to one if object i is the picked or placed object, respectively. For example, consider the action in Figure 6.2a, $\texttt{pick-place}(o_3, -1, o_0, 0)$, which translates as "pick up o_3 from its left and place it on top of o_0". Here, a_3 and a_0 are set to $[1, -1, 0, 0]$ and $[0, 0, 1, 0]$, respectively, and a_1 and a_2 are set to zero vectors. One can possibly aggregate the input multiple times by applying Equation (6.6) more than once to model longer effect chains. In our experiments, we use a single aggregation step. Multiple combinations from multiple attention heads are concatenated in Equation (6.7) to produce a single representation h_i for each object.

As the final step, the decoder takes the aggregated representation h_i as input and predicts the effect $\hat{e}$ for each object for the executed action a. The decoder is a multi-layer perceptron. The predicted effect is then compared with the ground truth effect e to compute the mean squared error:

$$\mathcal{L} = \frac{1}{M} \sum_{j=1}^{M} \sum_{i=1}^{N} (\hat{e}_i^{(j)} - e_i^{(j)})^2, \tag{6.8}$$

where M is the batch size, and N is the number of objects.

6.3.2. Comparison with Related Models

As in DeepSym (Figure 6.1 – bottom left), this architecture is also an encoder-decoder architecture with discrete bottleneck layers. The difference is that the information between objects is shared in the aggregation function using the learned attention weights for a more accurate effect prediction for actions involving several objects. In DeepSym, this can only be achieved by fixing the number of input objects, whereas there is no such limitation in the proposed model.

Regarding the Attentive DeepSym architecture [53] (Figure 6.1 – bottom right), the most significant difference is the placement of the self-attention module. In the previous study [53], the self-attention module takes object symbols (the encoder's output) as its input and directly outputs the aggregated representation. This restricts the model from learning attention weights only from the learned symbols. In this proposal, attention weights are learned from object features, making relations more general.

The second significant difference is the explicit use of attention weights. In the previous study [53], attention weights are used within the self-attention layer as in the original Transformer architecture [26]. However, since attention weights are continuous, they cannot be easily expressed as relational symbols between objects.

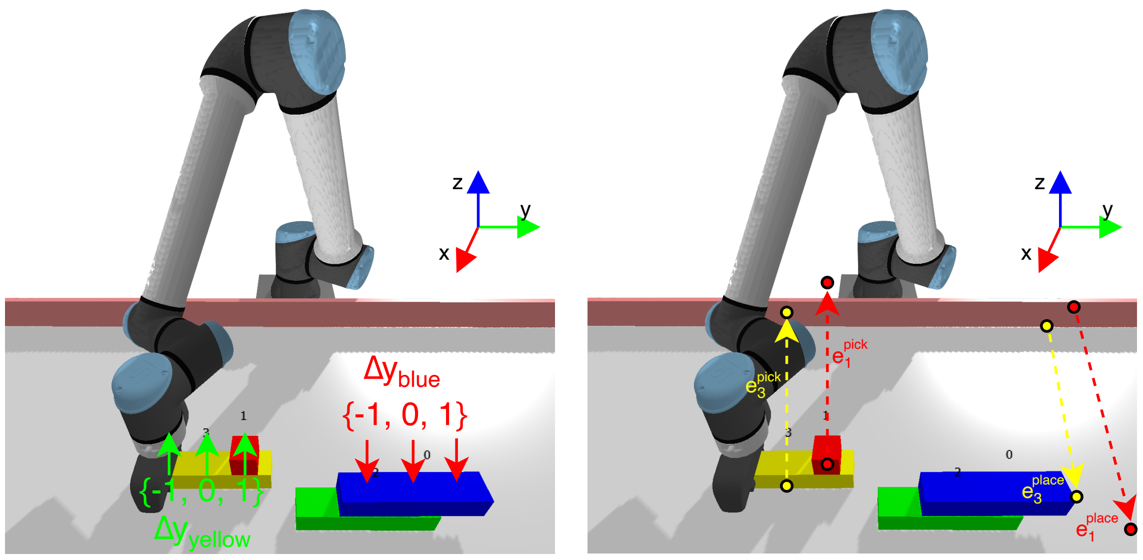
6.4. Experiments

6.4.1. Experiment Setup

6.4.1.1. Environment. We created a tabletop object manipulation environment for our experiments (Figure 6.2). The environment consists of a UR10 robot and two to four objects. These objects are either short blocks or long blocks with their physical properties (e.g., size, mass, friction coefficient) fixed through the interaction phase. The robot has a single type of high-level action: grasping and releasing an object on top of or near another object. We assume that object positions can be recognized by a separate module, and the robot can track the cartesian position change of these objects. What is to be learned is the effect of the executed action on each object in different configurations.

In our experiments, we first collect a fixed-size dataset required for the training by interacting with the environment and then train the model. Note that this procedure can be turned into a buffer-based training where the model training and the data collection are done in parallel, similar to many reinforcement learning setups.

6.4.1.2. Data Collection. At each iteration of the exploration process, the robot picks a grounded action a, i.e., a specific parameterization of the action such as `pick-place`(2, -7.5, 1, 0.0), and executes it in the environment to observe effect e_i of action a on each object i. Object features before the execution of the action are recorded as the state vector. Here, object features are object types and poses with respect to the object frame that is going to be picked, which allows models to generalize to different object positions. Effects are the concatenation of (1) the position change of objects after the pick-up action, and (2) the position change of objects after the release action: $e_i = [\delta(o'^{\text{pick}}_i, o^{\text{pick}}_i), \delta(o'^{\text{place}}_i, o^{\text{place}}_i)]$ resulting in a 6-dimensional vector for each object (Figure 6.2b). Such an effect representation filters out the movement effect of the object from the source location to the target location. In this way, the effect representation models what happens 'immediately after the pick-up' and 'immediately after the release' actions. Object and effect representations might have been selected as raw images as in Chapters 4 and 5; however, we opt for a simpler setup to compare different architectures in a controlled environment.



(a) (b)

Figure 6.2. An example interaction with the environment. Two objects are selected as pick and place targets. Green and red arrows show possible grasp and release locations, respectively. (a) `pick-place`$(o_3, -1, o_0, 0)$. (b) An example effect.

To compare different architectures in different settings, we collected three datasets that contain exactly two, three, or four objects. We combine these datasets to create a fourth dataset that contains a varying number of objects. Each dataset contains $(\{o_1, o_2, \ldots, o_n\}, a, \{e_1, e_2, \ldots, e_n\})$ triplets where n is the number of objects. We collect 120K samples for two objects, 180K for three objects, and 240K for four objects. We use 80% of samples for training, 10% for validation, and 10% for testing.

6.4.1.3. Baselines. We compare our method with DeepSym [46] and Attentive DeepSym [53]. As the vanilla DeepSym architecture requires a fixed-size input and output, we modified it to make it suitable for our experiments. Namely, a maximum number of objects is determined for a given training session. Then, the input (and the output) vector is reshaped into $[o_{\text{grasped}}, o_{\text{released}}, o_{\text{rest}}]$ where o_{grasped} and o_{released} are the object features of the grasped and released objects, respectively, and o_{rest} is the object features of the remaining objects.

6.4.1.4. Training Details. All architectures are trained with the same hyperparameters throughout the text unless mentioned otherwise. We train models for 4000 epochs with five repetitions with different seeds. Adam optimizer [7] is used with a batch size of 128 and a learning rate of 0.0001. All network components (e.g., encoder, decoder) consist of two hidden layers with 128 hidden units. The number of attention heads for attentive models is set to four. We clip gradients by their norm to 10.

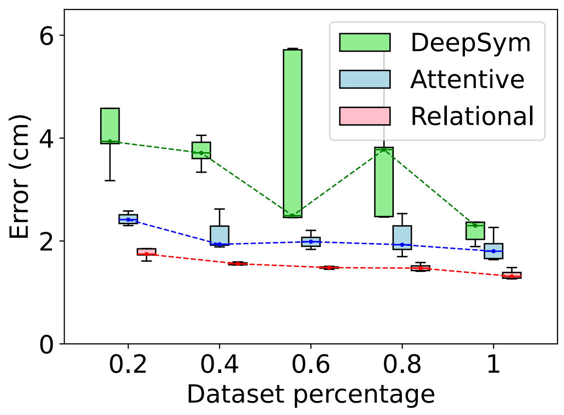
6.4.2. Effect Prediction Results

Firstly, we compare effect prediction results for different datasets in Table 6.1. The reported results are absolute errors summed over all dimensions (three dimensions for the pick-up effect and three dimensions for the release effect). The results show that the proposed method achieves significantly lower errors than others. Moreover, the variance is lower than others, indicating that Relational DeepSym is more robust to different seeds.

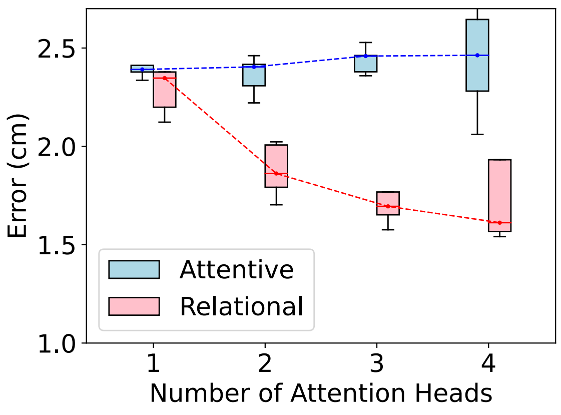
Table 6.1. Effect prediction results averaged over five runs. Units are in centimeters.

Dataset	DeepSym	Attentive	Relational
Two objects	2.22 ± 0.56	0.89 ± 0.10	0.50 ± 0.03
Three objects	3.06 ± 0.16	2.55 ± 0.09	1.67 ± 0.02
Four objects	4.26 ± 0.68	2.75 ± 0.12	2.00 ± 0.04
Two to four objects	2.38 ± 0.25	1.86 ± 0.12	1.35 ± 0.04

Errors increase as the number of objects increases. This is expected since the number of unique effects increases with the number of objects as the robot creates more complex structures in random exploration. Choosing the exploration schedule in a guided way, similar to experience replay in reinforcement learning [118], would be a promising future direction.



(a)



(b)

Figure 6.3. Prediction errors for different models as the number of samples (a) and attention heads (b) increase.

To compare the sample efficiency, we train each model on a subset of the full train set that is composed of interactions with two to four objects (432K samples in total) and evaluate the effect prediction performance on the full test set. Figure 6.3a shows the prediction errors over five runs for different models as the number of samples increases. Each model is trained for 1000 epochs except for the full train set, where we train for 4000 epochs. The results show that the performances of Attentive DeepSym and Relational DeepSym increase in a similar fashion with the increasing sample size. However, the overall performance of Relational DeepSym is better than Attentive DeepSym on all sample sizes. This suggests that a bottleneck (in this case, the discrete representation) with a set of object symbols and relational symbols is more sample-efficient than one with only object symbols. In Attentive DeepSym, the bottleneck is essentially a set of object symbols and relations are implicitly learned from these symbols (see Figure 6.1 – bottom right). On the other hand, in Relational DeepSym, the object and the relational symbols are processed independently from each other (f_o and f_r in Figure 6.1 – top) and combined in the aggregation function. This allows the model to represent the environment with more symbols.

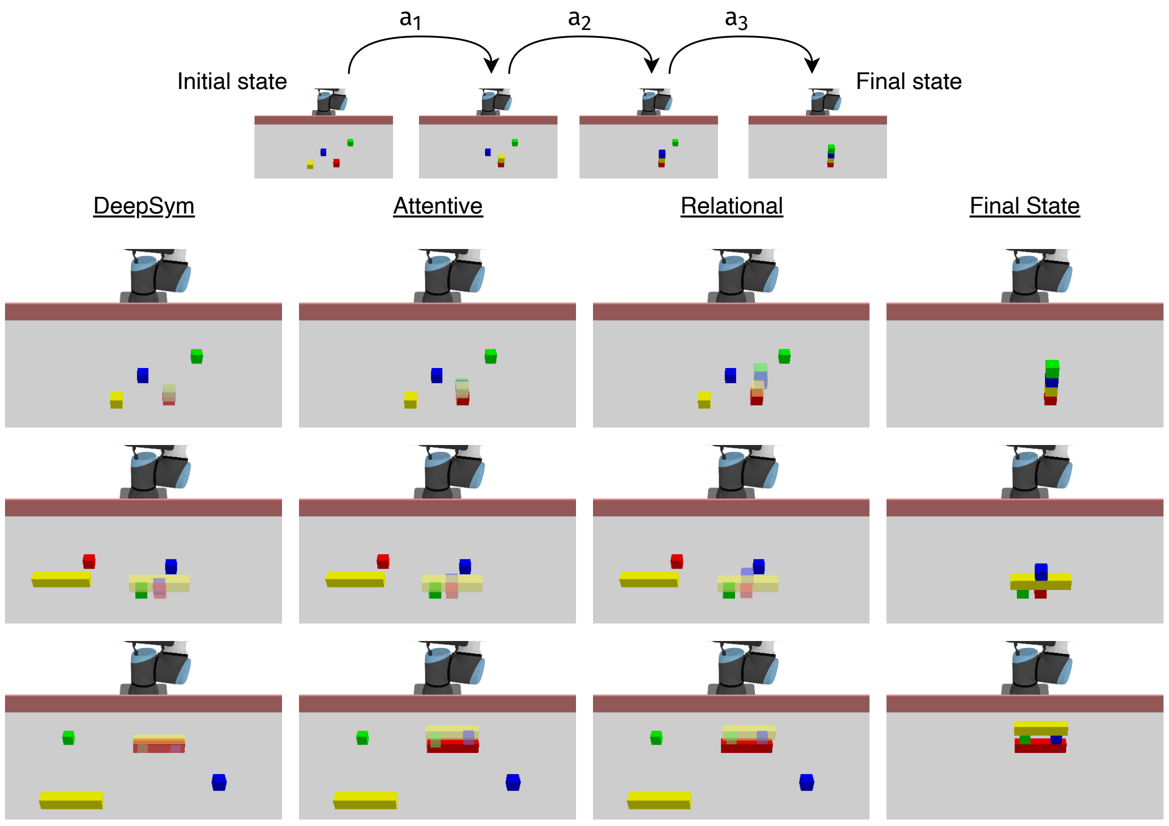
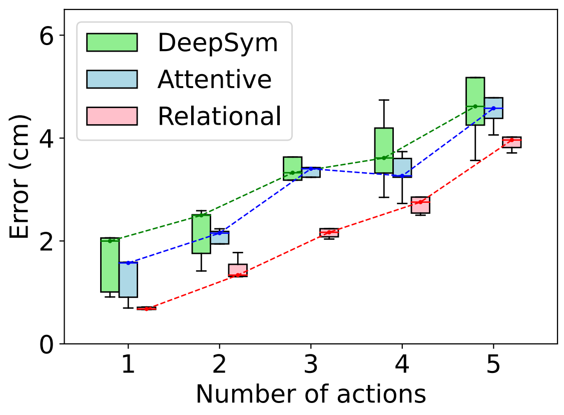
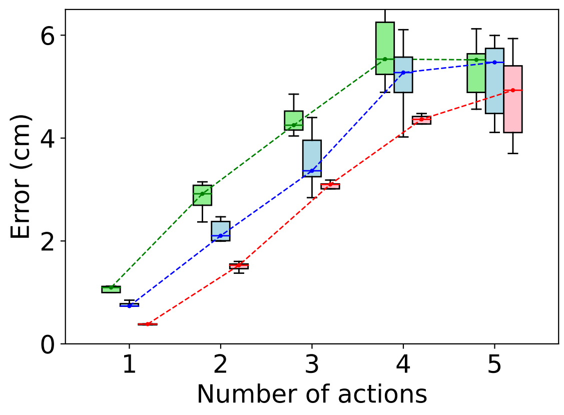


Figure 6.4. Action sequence prediction results for different models. Top – An example action sequence is shown with the initial state in the first image and the ground truth final state after executing the action sequence in the fourth image. Rows 2-4 – The final object positions predicted by each network are shown with a transparent color.

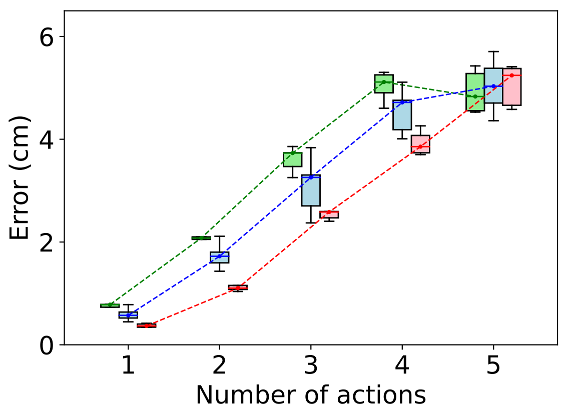
Next, we analyze the effect of the number of attention heads on the prediction performance. Figure 6.3b shows the prediction errors for Attentive and Relational DeepSym for one to four attention heads. We see that the performance of Attentive DeepSym remains the same for different numbers of heads. As the discrete bottleneck in Attentive DeepSym is not relations but object symbols, the number of attention heads does not affect the effect prediction accuracy. However, the performance of Relational DeepSym increases with the increasing number of attention heads and thus the number of relations since the model capacity increases with multiple relations in the aggregation step. The number of attention heads is a hyperparameter that needs to be tuned for each problem by finding the plateau in the performance, as done in DeepSym [46] for the dimensionality of object symbols.



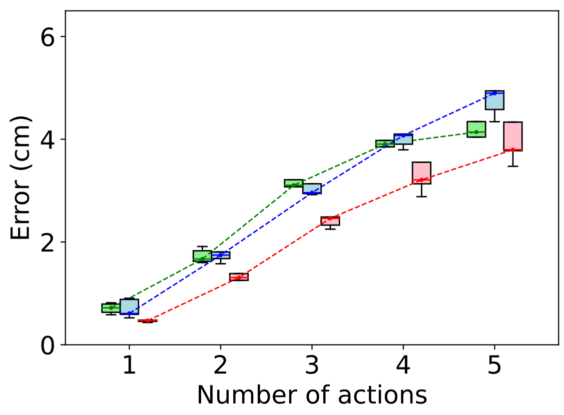
(a)



(b)



(c)



(d)

Figure 6.5. Prediction errors for different models as the number of actions increases. (a) Two objects, (b) three objects, (c) four objects, (d) two to four objects.

6.4.3. Action Sequence Prediction

In this section, using the effect predictions $\{\hat{e}_1, \ldots, \hat{e}_n\}$ of models, we predict the next state $\{\hat{o}'_1, \ldots, \hat{o}'_n\}$ by adding the prediction back into the position part of the state vector. Firstly, the predicted pick-up and release effects are summed with the state vector. Then, the movement from the pick-up position to the release position is added for objects that are predicted to be picked up. This way, given an initial state, we can predict the final state the robot reaches after executing a sequence of actions. Here, the challenge is to understand what happens when an object is lifted and released on top of another object in the presence of multiple objects.

Figure 6.4 shows action sequence prediction examples. Relational DeepSym's predictions are more accurate than others, especially in the z-axis, the most significant axis in these experiments. This shows that the proposed model understands that the presence of an object on top of another object will change the action results.

In Figure 6.5, we analyze how models perform as the number of actions increases. We see that Relational DeepSym shows a slightly lower error than others. Errors increase for all models when the number of actions increases. This is an expected result since we add the effect prediction back into the state vector, effectively cascading the error over multiple steps.

6.4.4. Comparing Different Activations for Relations

In this section, we compare the performance of the Gumbel-sigmoid activation used for learning relational symbols with sigmoid, softmax, and Gumbel-softmax functions. Although Gumbel-sigmoid is also used for learning object symbols as well, we rather focus and ablate on the relational part. We train a Relational DeepSym model with the same hyperparameters as in Section 6.4.1 except for the activation function used in Equation (6.3) to compute object-object relations.

Table 6.2. Effect prediction results with different activation functions for relations. Units are in centimeters.

	w/o rounding	w/ rounding
Gumbel-sigmoid	1.34 ± 0.09	1.83 ± 0.41
Gumbel-softmax	1.66 ± 0.16	2.70 ± 0.61
Sigmoid	1.32 ± 0.04	24.03 ± 3.71
Softmax	1.35 ± 0.03	23.26 ± 2.83

We report errors on the test set for different activations in Table 6.2. Since learning a symbolic definition of the environment is a requirement that we want to satisfy to enable domain-independent planning with off-the-shelf AI planners [39, 42] [46], [81], [117], we can only use discrete outputs that are rounded to either zero or one at inference time. As such, we report two different results: (1) without rounding and (2) with rounding. The results show that using Gumbel-sigmoid for learning relational symbols achieves lower errors in terms of effect prediction. This is expected since Gumbel-sigmoid is designed to approximate a Bernoulli distribution, which is in accordance with the distribution of pairwise relations; an object-object relation is either active or inactive.

6.5. Conclusion

In this chapter, we proposed a new method to simultaneously learn object symbols and relations between objects in a single architecture. Namely, *discrete* attention weights are computed from object features to model relations between objects. As these weights are discrete, they can be regarded as relational symbols between objects. Such a feature is desirable because it allows us to model the environment with object symbols and relations between objects, which was not available previously [46], [53]. We showed that the proposed model achieves significantly lower errors than others in predicting the effects of (possibly a sequence of) actions on a varying number of objects and produces meaningful symbols that allow us to model the relations between objects for settings where the number of objects can vary. In Chapter 8, we will show how to convert the learned symbols into PDDL operators [119] for domain-agnostic planning with off-the-shelf planners. But first, in the next chapter, we will focus on how to learn high-level skills from parameterized actions, an assumption we took for granted up to these chapters.

7. LEARNING SYMBOLIC SKILLS FROM PARAMETERIZED ACTIONS

7.1. Introduction

In the previous chapter, we focused on learning object symbols and relations from the continuous interaction experience of the robot. The robot interacted with its environment using pre-defined high-level movement primitives, such as picking and placing an object. It was assumed that such primitives could be bootstrapped from motor babbling data [43] or can be learned from expert demonstrations [100, 101].

In this chapter, we show how Relational DeepSym architecture can be extended to bootstrap the robot with high-level skills that can be used in further stages of learning to interact with the environment. This will ultimately allow the system to start from very little knowledge and engineering (only the assumption of recognizing object positions), develop high-level representations that are in line with its embodiment and environment, and use them for domain-independent planning. Effectively, the system converts its continuously represented sensorimotor information into domain-specific elements.

7.2. Problem Definition

We start off with the assumption that the robot is equipped with a set of continuously parameterized object-oriented actions $\mathcal{A} = \{a_1(\theta, o), \dots, a_k(\theta, o)\}$ where o is the object defining the center frame of the action and $\theta \in \mathbb{R}^d$ is the parameter vector of the action defining the trajectory in cartesian space that the robot follows:

$$a(\theta_i, o_j) = (p_1, \dots, p_n)_{\theta_i, o_j}. \tag{7.1}$$

Here, p_i is a three-dimensional point in cartesian space. We can sample different parameterizations of the action by sampling θ from a uniform distribution. There may be more than one type of parameterized action. In this chapter, we consider a single type of parameterized action defined by three via points, an anchor position (in our case, object position), and a grasp state. We are interested in learning a set of parameterizations $\{\theta_1, \ldots, \theta_m\}$ such that these parameterizations maximally cover the effects in the data set constructed by executing actions with random parameterizations. Movements defined by these parameterizations are treated as skills learned by the robot.

For instance, consider a robot pushing an object. Once the position of the object and the desired angle for push is determined, it is quite trivial to generate the trajectory that the robot follows to push the object. Parameters such as object positions, push angles, grasp points, via points, and so on are continuously defined values that allow us to bridge the low-level motor commands with the task space.

Note that here, we have a more generic action definition than pre-defined actions used in previous chapters; the current formulation defines a continuous action space, which prevents us from searching since branches of a node would not be finite.

7.3. Methods

7.3.1. Learning Object and Action Symbols

Figure 7.1 shows the outline of the proposed architecture for learning skills together with object symbols. The architecture is an extension of Relational DeepSym [55] (compare with Figure 6.1) where there is an action encoder f_a (in addition to object encoder f_o and relation encoder f_r) for transforming action parameters into symbolic outputs with Gumbel-sigmoid function for preserving differentiability [51,52].

As in Chapter 6, we use object and relation encoders, f_o and f_r, for learning symbols required for predicting the effect of the action. Previously, as the robot had a finite set of actions that it could apply on objects, we directly concatenated the one-hot enumeration of the action to object symbols $\{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$. Since the action space is now continuous, we cannot directly concatenate its parameters since doing so would prevent us from converting the collected data into a symbolic data set. Therefore, as object symbols and relations are discrete (which constitute nodes of the search tree), we need a symbolic counterpart of the action (which would be the edges of the search tree). The action encoder f_a serves this purpose by transforming action parameters into a discrete output. The action encoder takes the acted object o_j as input, and its output $\mathbf{p}_j$ is concatenated to object o_j's object symbol $\mathbf{z}_j$. For other object symbols, we concatenate a zero vector with the same length.

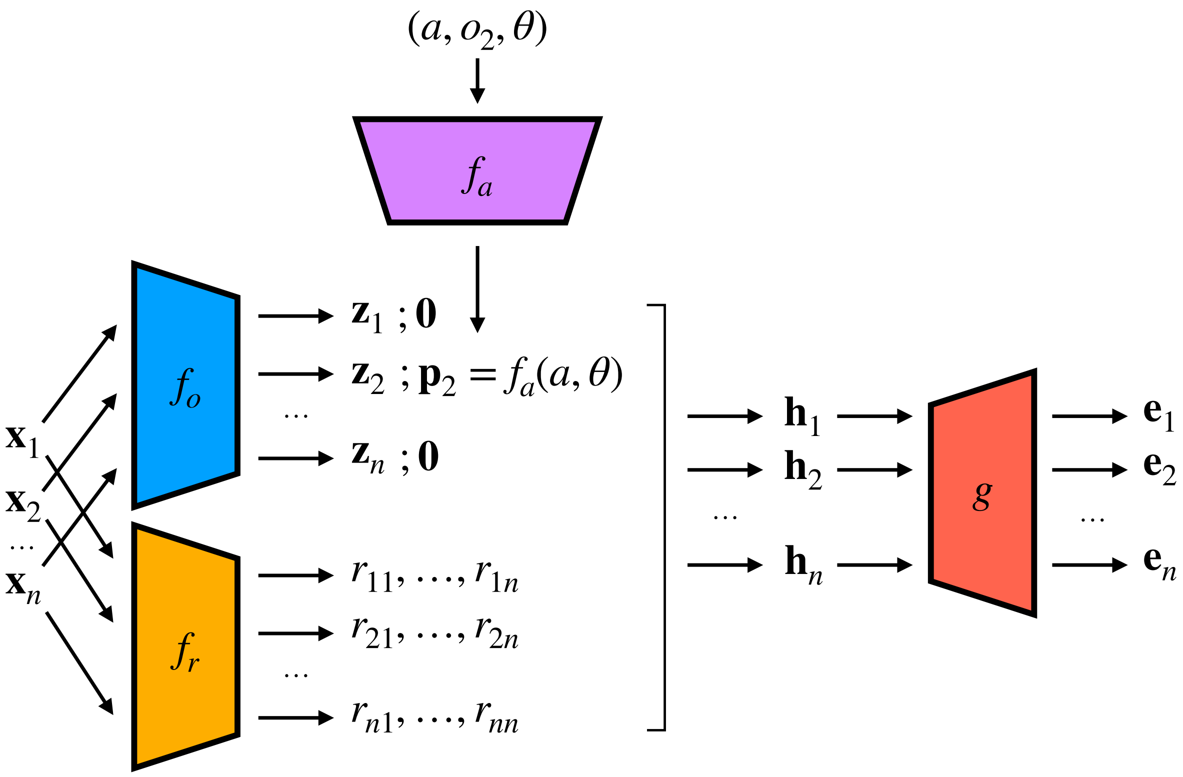


Figure 7.1. The outline of the architecture for learning skills.

Object symbols $\{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$, relations $\{\ldots, r_{ij}, \ldots\}$, and the action symbol $\mathbf{p}$ is aggregated as in Section 6.3.1 which results in a fixed-size vector $\mathbf{h}_i$ for each object i. The decoder g predicts effect $\hat{\mathbf{e}}_i$ for each object, and the whole architecture is trained to minimize the following objective:

$$\mathcal{L} = \sum_{i=1}^{n} \|\hat{\mathbf{e}}_i - \mathbf{e}_i\|^2. \tag{7.2}$$

```
Require: $f_a$, $n_{\text{sample}}$, $n_{\text{iter}}$, $n_{\text{min}}$ $\eta$, $\mathcal{O}$, $\tau$
  $\theta^* \leftarrow \{\}$
  for each $\underline{\mathbf{p}}$ in $\mathcal{O}$ do
    $\theta \leftarrow \{\}$
    for $i$ in $n_{\text{sample}}$ do
      $\theta[i] \leftarrow$ get_random_parameterization()
    end for
    for $i$ in $n_{\text{iter}}$ do
      $\mathcal{L} \leftarrow \|f_a(\theta[i]) - \underline{\mathbf{p}}\|^2$
      $\theta[i] \leftarrow \theta[i] - \eta\nabla_{\theta[i]}\mathcal{L}$
    end for
    $\theta^*[\underline{\mathbf{p}}] \leftarrow \{\}$
    for $i$ in $n_{\text{sample}}$ do
      if $\|f_a(\theta[i]) - \underline{\mathbf{p}}\| \leq \tau$ then
        $\theta^*[\underline{\mathbf{p}}]$.append($\theta[i]$)
      end if
    end for
    if length($\theta^*[\underline{\mathbf{p}}]$) $> n_{\text{min}}$ then
      $\theta^*[\underline{\mathbf{p}}] \leftarrow$ mean($\theta^*[\underline{\mathbf{p}}]$)
    else
      $\theta^*[\underline{\mathbf{p}}] \leftarrow$ null
    end if
  end for
  return $\theta^*$
```

Figure 7.2. Extract parameterizations from the action encoder.

The action encoder should produce such action symbols that when used together with object symbols and relations, the effect space should be maximally covered. Intuitively, the action encoder should cluster the action space in a way that actions bound with the same symbol should produce similar outputs when given the same object symbols and relations.

7.3.2. Extracting Skills from Action Symbols

After training the architecture, we can convert state $\mathbf{X}$ and action (a, o, θ) into their symbolic counterparts $Z = \{\mathbf{z}_1, \dots, \mathbf{z}_n\}$, $R = \{\dots, r_{ij}, \dots\}$, and $\mathbf{p}$. Once we convert and build rules defined over these symbols, we can search for a plan (an action symbol sequence) $\mathcal{P} = (\mathbf{p}^{(1)}, \dots, \mathbf{p}^{(k)}\}$ that would reach a goal state $\mathbf{X}_g$ from an initial state $\mathbf{X}_i$. However, even if we find the correct action symbol sequence, we cannot execute them in the environment as their parameterizations are not known. In Algorithm 7.2, we show a procedure to extract parameterizations for each action symbol that can be later used for action execution.

We start by enumerating all possible outputs that f_a generate—all d_a-dimensional binary vectors where d_a is the output dimensionality of f_a. For each enumeration $\underline{\mathbf{p}}$ (a grounded symbol), we optimize randomly sampled parameterizations in the direction that would drive them towards $\underline{\mathbf{p}}$. This can be thought of as moving parameterizations to each grounded symbol's prototype vector. Then, we take a mean of samples that are below an error threshold τ and record them as the learned parameterizations.

7.4. Experiments

7.4.1. Experiment Setup

Our experiments are done in a tabletop setup with four types of objects: small cubes, large cubes, long cubes, and boxes. Each environment instance starts with four to six random objects. A UR10 robot is equipped with a single action parameterized by (1) an anchor position and (2) three via points. We assume that we can detect objects' positions and set the center of an object as the anchor position. Via points are four-dimensional vectors where the first three dimensions define the relative cartesian distance from the anchor, and the fourth dimension defines the open/close state of the gripper before moving to the via point. Via points are sampled from three different boxes with 15x15x15cm sizes shown in Figure 7.3. This results in a quite large action space.

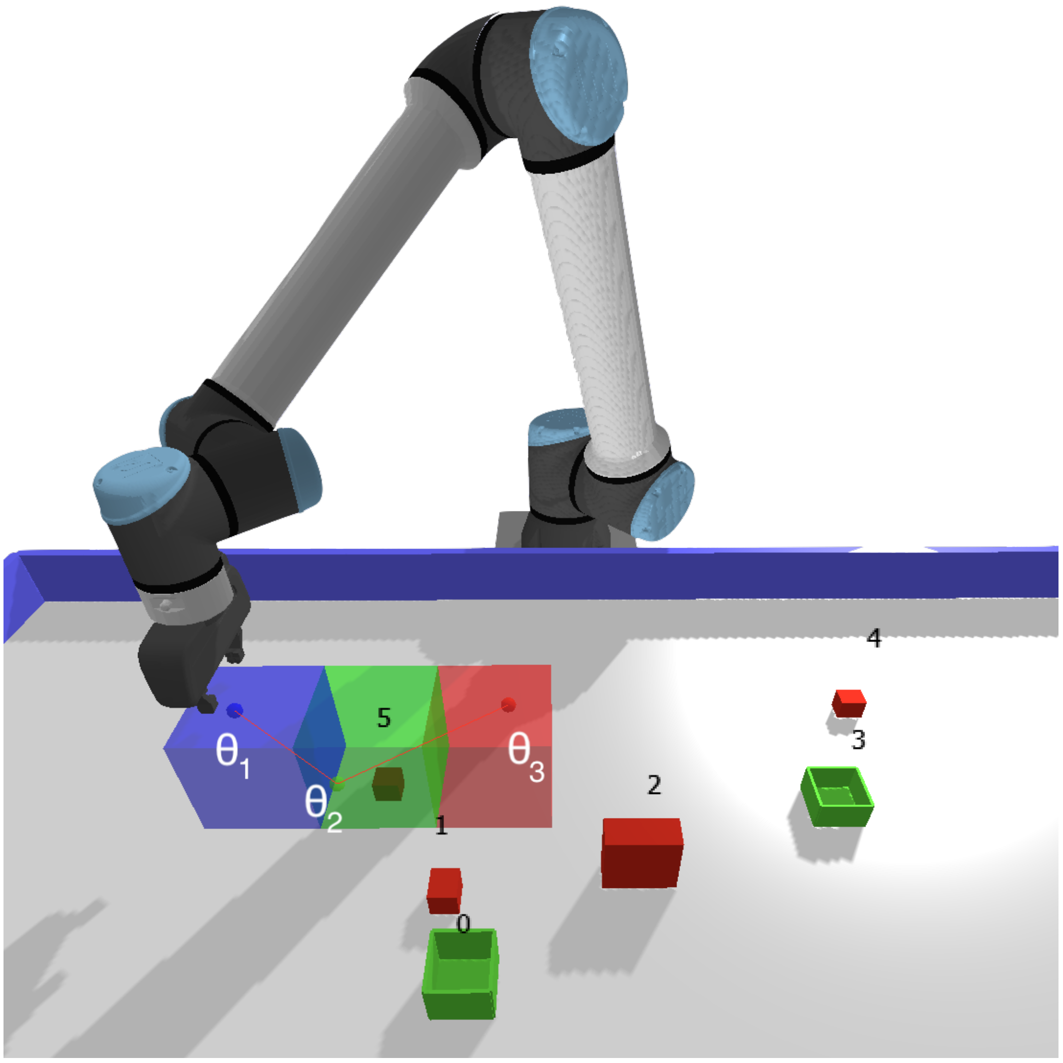


Figure 7.3. An example parameterized action from the exploration process. Blue, green, and red shaded areas show the sample space of via points when o_5 is taken to be the anchor object. θ_1, θ_2, and θ_3 are the sampled parameters.

In the data collection process, the robot randomly picks a target object and a random parameterization of the action. The state information, which consists of each object's position, orientation, size, and whether the object is touched by the gripper or not (12 dimensions in total), is saved before the action as $\mathbf{X}$. Then, the robot executes the action and saves the state after the execution as $\mathbf{X}'$. This results in the following data set: $\mathcal{D} = \{\mathbf{X}^{(i)}, (a, o, \theta)^{(i)}, \mathbf{X}'^{(i)}\}_{i=1}^{N}$.

We collect 200K samples and use 80% for training the method. The model is trained for 1000 epochs with a batch size of 200 and a learning rate of 1e-5 using Adam optimizer [7]. Each MLP block consists of four layers with 128 hidden units. The output dimensionality of the object and the action encoders, as well as the number of attention heads, are set to four. In order to prevent large steps in the training, we clip the global norm of all gradients by 100.

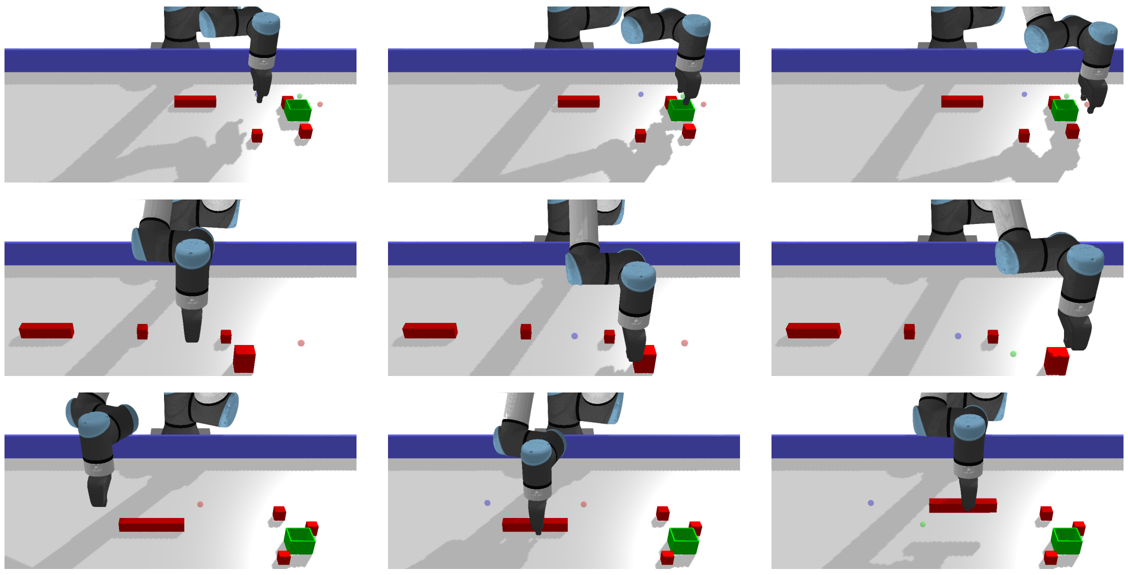


Figure 7.4. Three example parameterized actions found with Algorithm 7.2 after training the network. Top – empty action, middle – push, bottom – grasp.

7.4.2. Learned Action Parameterizations

After training the network, we extract parameterizations from the action encoder using Algorithm 7.2. As we set the action encoder's output dimensionality to four, there can be at most 16 unique symbols and, thus, 16 distinct parameterizations. In Figure 7.4, we show example parameterized actions in each row. Note that these parameterizations are not optimized based on a specific objective but rather exemplars—prototypes—of the exploration data. In other words, we do not force the system to find parameters that would increase a specific value, a reward, or the interaction with objects. Nevertheless, we see that some of the found parameterizations are useful for interacting with objects, which is expected since the randomly collected dataset also contains interactions with objects.

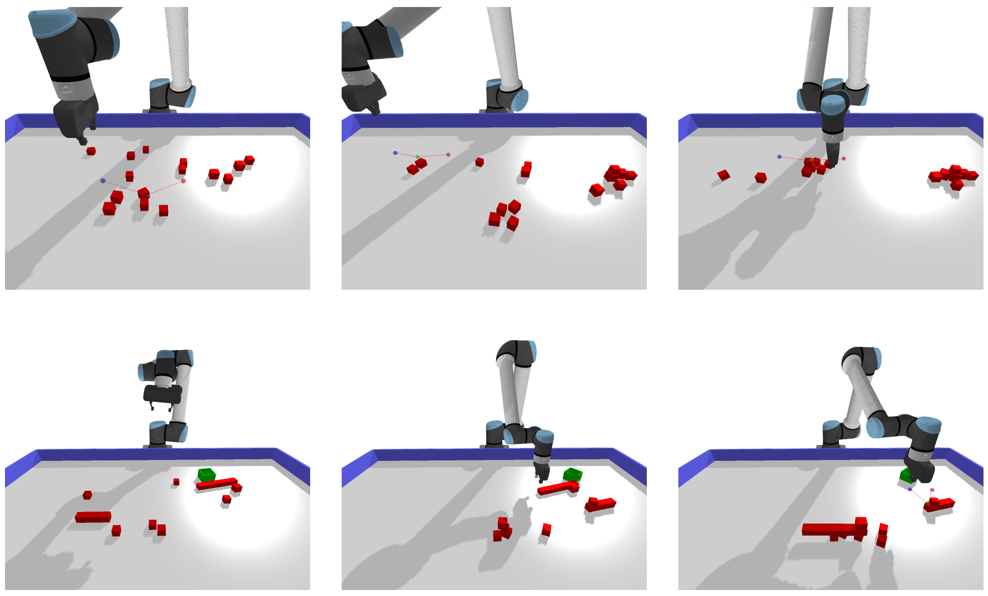


Figure 7.5. The evolution of the environment through time.

To show the usefulness of the found parameterizations, in Figure 7.5, we show an example exploration process by executing only parameterizations that mimic ‘grasp’ and ‘release’. Each row shows the change in the environment through time while the robot interacts with objects. This shows that the found parameters can be used to bootstrap the agent with useful skills, enabling the robot to interact with its environment to collect new experiences for further stages of learning.

7.5. Conclusion

In this chapter, we showed how Relational DeepSym architecture can be extended to learn high-level skills from continuously parameterized object-oriented actions. This essentially shows that the original ‘the robot has pre-defined actions’ assumption can be addressed as some of the found skills (push, grasp, release) are used in the previous chapters to learn object symbols.

Nevertheless, we would like to mention that finding useful skills that interact with objects while keeping the sample complexity at a reasonable level is only possible if we keep the search space small. For this reason, object-oriented parameterizations are strong candidates as they only require locating interactable entities in the environment, which might be achieved with depth cameras. With similar motivation, in a related work [120], the learning part of the manipulation is kept at minimum by starting from (1) good pre-grasp poses found by a task-general grasp detector [121], (2) moving into configurations with high manipulability [122] while using (3) variable impedance control in end-effector space [123]. These factors combined together drastically narrow down the search space.

Currently, the robot draws uniformly random action parameterizations while interacting with the environment. However, this might not be very sample-efficient if a large portion of the parameter space leads to null effects. As a future direction, we plan to search the parameter space with methods that are better suited for efficiently exploring continuous spaces, such as curiosity-based methods [124], evolutionary algorithms [125–127], or leverage LLMs by constructing a natural language interface with the domain [35].

As we showed that it is possible to learn high-level skills that would interact with objects, in the next chapter, we take it for granted in order to focus on another aspect of the overall pipeline: building rules defined over the learned symbols with Relational DeepSym.

8. SYMBOLIC MANIPULATION PLANNING WITH DISCOVERED OBJECT AND RELATIONAL PREDICATES

8.1. Introduction

In Chapter 6, we proposed an architecture that can learn not only unary object symbols but explicitly encodes relations between objects using binary attention weights. However, symbolic-level transitions from the learned symbols that enable domain-independent planning were not considered.

In this chapter, we first learn a set of unary and relational symbols between objects using the Relational DeepSym architecture from the previous chapter [55], then we propose a method to build abstract operators defined over these unary and relational symbols that describe the symbolic transition of a state when an action is executed. We translate these operators into PDDL descriptions and show that they can be used for planning with off-the-shelf AI planners in a tabletop object stacking task. We compare our method with DeepSym [46] and Attentive DeepSym [53] in terms of effect prediction accuracy and planning performance. Our results show that planning with the operators defined over relational symbols performs better than the baselines.

8.2. Problem Definition

An environment is characterized by a tuple $(\mathcal{X}, \mathcal{A}, P)$ where $\mathcal{X}$ denotes the continuous sensory state of the environment (which is called state-space from now on), $\mathcal{A}$ is a finite set of actions that the agent can execute, and $P(\mathbf{X}' \mid \mathbf{X}, \mathbf{a})$ is the probability that taking action $\mathbf{a} \in \mathcal{A}$ at state $\mathbf{X}$ results in state $\mathbf{X}'$. An environment instance consists of a set of objects $\{o_1, \ldots, o_n\} \in \mathcal{O}$ each having a d_o-dimensional continuous-valued feature vector $\mathbf{x}_i \in \mathbb{R}^{d_o}$ defining the state of the environment as an unordered set of feature vectors $\mathbf{X} := \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \in \mathcal{X}$. $\mathbf{X}$ is not a fixed-size vector but a variable size depending on the number of objects n in the environment. An action $\mathbf{a}$ is a fixed-size vector representing a high-level parameterized movement primitive that the agent can execute, such as picking an object. Given an initial state $\mathbf{X}_0$ and a goal state $\mathbf{X}_g$, the objective is to find a sequence of actions $(\mathbf{a}_1, \ldots, \mathbf{a}_k)$ that maximizes the probability of reaching the goal state.

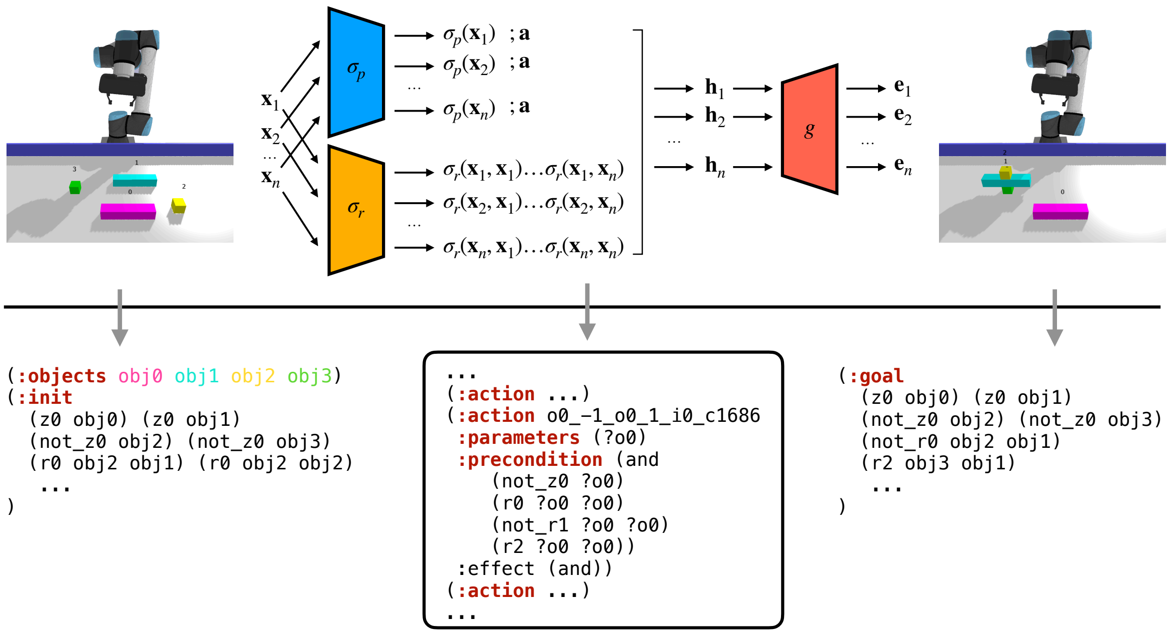
We are interested in learning the mapping $f : \mathcal{X} \to \mathcal{P}$ that transforms a state vector $\mathbf{X}$ into a set of predicates (or symbols) $\varSigma := \{\sigma_1(\mathbf{X}), \ldots, \sigma_m(\mathbf{X})\} \in \mathcal{P}$, where $\sigma_i : \mathcal{X} \to \{0, 1\}^{d_k}$ is a binary function where d_k is an environment dependent fixed dimension. After symbols are learned, we can find a set of operators (i.e., lifted actions in the symbolic space) $\{\phi_1, \ldots, \phi_k\} \in \varPhi$ in which each operator $\phi_i : \varSigma \to \varSigma$ transforms the current symbols into a new set of symbols. Once symbols and operators are learned, we can transform the initial state $\mathbf{X}_0$ and the goal state $\mathbf{X}_g$ into symbolic representations $\varSigma_0$ and $\varSigma_g$, respectively, and then find a sequence of operators $(\phi_1, \ldots, \phi_k)$ that transforms $\varSigma_0$ into $\varSigma_g$, and then execute the corresponding sequence of actions $(\mathbf{a}_1, \ldots, \mathbf{a}_k)$ to reach the goal state.

Note that the proposed operator learning system is built on top of our symbol learning network architecture [55]. Thus, to assess the added value of learning explicit symbolic transitions, we compare the performance of the proposed model in effect prediction and planning with DeepSym [46] and Attentive DeepSym [53] in a tabletop object manipulation setup.

8.3. Methods

8.3.1. Learning Unary and Relational Symbols

Figure 8.1 shows an outline of the method. The architecture is composed of four main blocks. The encoder network σ_p learns unary symbols over object features while the self-attention network σ_r learns relational symbols. The aggregation in the middle fuses unary and relational symbols with action into a vector representation for each object, which is given as input to the decoder network g to predict the effect of the executed action. After learning operators defined over the learned symbols, we can find a sequence of actions that reach the goal state from the initial state with AI planners.



σ_p
σ_r
$\mathbf{x}_1$
$\mathbf{x}_2$
...
$\mathbf{x}_n$
$\sigma_p(\mathbf{x}_1)$;a
$\sigma_p(\mathbf{x}_2)$;a
...
$\sigma_p(\mathbf{x}_n)$;a
$\sigma_r(\mathbf{x}_1,\mathbf{x}_1)\ldots\sigma_r(\mathbf{x}_1,\mathbf{x}_n)$
$\sigma_r(\mathbf{x}_2,\mathbf{x}_1)\ldots\sigma_r(\mathbf{x}_2,\mathbf{x}_n)$
...
$\sigma_r(\mathbf{x}_n,\mathbf{x}_1)\ldots\sigma_r(\mathbf{x}_n,\mathbf{x}_n)$
$\mathbf{h}_1$
$\mathbf{h}_2$
...
$\mathbf{h}_n$
g
$\mathbf{e}_1$
$\mathbf{e}_2$
...
$\mathbf{e}_n$

```
(:objects obj0 obj1 obj2 obj3)
(:init
  (z0 obj0) (z0 obj1)
  (not_z0 obj2) (not_z0 obj3)
  (r0 obj2 obj1) (r0 obj2 obj2)
   ...
)
```

```
...
(:action ...)
(:action o0_-1_o0_1_i0_c1686
 :parameters (?o0)
 :precondition (and
    (not_z0 ?o0)
    (r0 ?o0 ?o0)
    (not_r1 ?o0 ?o0)
    (r2 ?o0 ?o0))
 :effect (and))
(:action ...)
...
```

```
(:goal
  (z0 obj0) (z0 obj1)
  (not_z0 obj2) (not_z0 obj3)
  (not_r0 obj2 obj1)
  (r2 obj3 obj1)
   ...
)
```

Figure 8.1. An overview of the proposed method.

8.3.1.1. Encoder Network. $\sigma_p : \mathcal{X} \rightarrow \mathcal{P}$ is a multi-layer perceptron (MLP) with Gumbel-Sigmoid (GS) [51] activation that outputs a binary number for each object in the environment. We treat this as a unary predicate $\sigma_{\text{p}}(\mathbf{x}_i)$ that encodes the property of an object. The number of properties that can be encoded is bounded by the output dimensionality of the MLP—at most 2^{d} properties can be encoded with d-dimensional outputs.

8.3.1.2. Self-Attention Network. $\sigma_r : \mathcal{X} \rightarrow \mathcal{P}$ is a multi-layer perceptron combined with a modified version [55] of the original self-attention layer [26]. Given a state vector $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, the MLP part outputs a set of d-dimensional vectors $\mathbf{Q} = \{\mathbf{q}_1, \ldots, \mathbf{q}_n\}$ and $\mathbf{K} = \{\mathbf{k}_1, \ldots, \mathbf{k}_n\}$ where $\mathbf{q}_i$ and $\mathbf{k}_i$ are the query and key vectors for object o_i, respectively. Unlike the original self-attention layer, we do not define value vectors as we are interested in the attention values—object symbols will be treated as value vectors. The second important difference is the computation of the attention values:

$$\alpha_{ij} = \text{GumbelSigmoid}(\mathbf{q}_i \cdot \mathbf{k}_j). \tag{8.1}$$

Firstly, using the sigmoid function instead of softmax allows attention values to focus on multiple tokens independently. Secondly, the binarization (due to GS) of attention values allows us to treat them as binary relations between objects while preserving differentiability.

We define the output of the whole block as a relational predicate $\sigma_{\text{r}}(\mathbf{x}_i, \mathbf{x}_j)$ that encodes the relation between objects o_i and o_j. Note that the order of arguments is important since $\mathbf{q}_i \cdot \mathbf{k}_j$ and $\mathbf{q}_j \cdot \mathbf{k}_i$ can have different values. Note that, without loss of generality, we described the operation with a single attention head; however, in the implementation, we used three attention heads.

<u>8.3.1.3. Aggregation Function.</u> In this step, unary predicates $\sigma_{\text{p}}(\mathbf{x}_i)$, relational predicates $\sigma_{\text{r}}(\mathbf{x}_i, \mathbf{x}_j)$, and the action vector $\mathbf{a}$ are fused in a single representation $\mathbf{h}_i$ for each object that is fed into the decoder network for predicting the effect of the executed action. The aggregation function is defined as

$$\begin{aligned} \mathbf{z}_i &= \text{MLP}([\sigma_{\text{p}}(\mathbf{x}_i); \mathbf{a}]) \quad \forall i \in \{1, \ldots, n\} && (8.2) \\ \mathbf{h}_i^k &= \sum_{j=1}^{n} \sigma_{\text{r}_k}(\mathbf{x}_i, \mathbf{x}_j)\mathbf{z}_i \quad \forall i \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, K\} && (8.3) \\ \mathbf{h}_i &= [\mathbf{h}_i^1; \ldots; \mathbf{h}_i^K], && (8.4) \end{aligned}$$

where K is the number of relation types. The action vector $\mathbf{a}$ is concatenated with the unary predicate $\sigma_{\text{p}}(\mathbf{x}_i)$, and then fed into an MLP to obtain a representation $\mathbf{z}_i$ that holds action information. Then, for each relation k and object i, intermediate representations $\mathbf{z}_j$ are summed up for indices that satisfy the relation $\sigma_{\text{r}_k}(\mathbf{x}_i, \mathbf{x}_j)$, resulting in a fixed-size vector $\mathbf{h}_i^k$ containing information regarding objects that have a relation r_k with object o_i. Lastly, $\{h_i^1, \ldots, h_i^k\}$ are concatenated to obtain the aggregated representation $\mathbf{h}_i$ that can hold any necessary information about the object o_i, the action $\mathbf{a}$, and the relations between o_i and other objects. Equation (8.3) essentially enables message passing between different object symbols based on the learned relational symbols between objects, giving us a unified representation.

8.3.1.4. Decoder Network. g is an MLP that takes the aggregated representation $\mathbf{h}_i$ as input and outputs the effect $\hat{\mathbf{e}}_i$ of action $\mathbf{a}$ on the object o_i. The predicted effect is then compared with the ground truth $\mathbf{e}_i$ to compute the mean squared error that is backpropagated through the whole network

$$\mathcal{L} = \sum_{i=1}^{n} \|\hat{\mathbf{e}}_i - \mathbf{e}_i\|^2, \tag{8.5}$$

where n is the number of objects and the effect vector $\mathbf{e}_i$ is defined as the difference between the current state $\mathbf{x}_i$ and the next state $\mathbf{x}'_i$.

These four blocks create a single differentiable module that can learn unary and relational predicate symbols over object features to minimize the effect prediction error in an end-to-end fashion. To train the network, we execute random actions in the environment and collect a dataset of $(\mathbf{X}, \mathbf{a}, \mathbf{X}')$ tuples. Then, we train the network to minimize the effect prediction error $\mathcal{L}$ defined in Equation (8.5).

8.3.2. Learning Operators

In this section, we describe how to learn operators that can be used for planning. Throughout this section, we will denote a ground symbol as $\sigma_p(\mathbf{x}_i)$, a lifted symbol as $\sigma_p(?x)$, and a substitution as $\theta = \{?x/\mathbf{x}_i\}$ where $?x$ indicates a free variable that is not bound to any object.

After we train the network, we can transform the dataset $\{(\mathbf{X}^{(i)}, \mathbf{a}^{(i)}, \mathbf{X}'^{(i)})\}_{i=1}^{N}$ into a dataset of propositional symbols $\{(\Sigma_p^{(i)}, \Sigma_r^{(i)}, \mathbf{a}^{(i)}, \Sigma_p'^{(i)}, \Sigma_r'^{(i)})\}_{i=1}^{N}$ where

$$\begin{aligned}
\Sigma_p^{(i)} &= \{\sigma_p(\mathbf{x}_1^{(i)}), \ldots, \sigma_p(\mathbf{x}_n^{(i)})\} & (8.6)\\
\Sigma_r^{(i)} &= \{\sigma_r(\mathbf{x}_1, \mathbf{x}_1), \ldots, \sigma_r(\mathbf{x}_i, \mathbf{x}_j), \ldots, \sigma_r(\mathbf{x}_n, \mathbf{x}_n)\}, & (8.7)
\end{aligned}$$

and $\Sigma_p'^{(i)}$, $\Sigma_r'^{(i)}$ are defined similarly. For notational simplicity, we consider only a single relation type σ_r while the same procedure can be applied to multiple relation types.

Our main goal is to find a set of operators $\varPhi = \{\phi_1, \ldots, \phi_m\}$ parameterized by lifted actions α that are in the form,

$$\phi_i(\varSigma_p, \varSigma_r; \alpha_i) = (\varSigma'_p, \varSigma'_r), \tag{8.8}$$

modeling the symbolic transition between states. We start by partitioning samples by their actions $\mathbf{a}^{(i)}$ and preconditions $\varSigma_p^{(i)}$, $\varSigma_r^{(i)}$: samples are grouped if their lifted actions and preconditions can be represented by the same substitution θ. For example, consider the following samples:

$$\begin{aligned} \varSigma_p^{(1)} &= \{\sigma_p(\mathbf{x}_1) = 0, \sigma_p(\mathbf{x}_2) = 0, \sigma_p(\mathbf{x}_3) = 1\} && (8.9) \\ \mathbf{a}^{(1)} &= \text{pick-place}(\mathbf{x}_3, \mathbf{x}_1) && (8.10) \\ \varSigma_p^{(2)} &= \{\sigma_p(\mathbf{x}_1) = 0, \sigma_p(\mathbf{x}_2) = 1, \sigma_p(\mathbf{x}_3) = 0\} && (8.11) \\ \mathbf{a}^{(1)} &= \text{pick-place}(\mathbf{x}_2, \mathbf{x}_3). && (8.12) \end{aligned}$$

These samples can be grouped into the same category C_1 with substitutions $\theta_1 = \{?a/\mathbf{x}_3, ?b/\mathbf{x}_1, ?c/\mathbf{x}_2\}$ and $\theta_2 = \{?a/\mathbf{x}_2, ?b/\mathbf{x}_3, ?c/\mathbf{x}_1\}$. This procedure gives us a set of groups $\{C_1, \ldots, C_k\}$ where each group is defined by lifted preconditions and actions. Next, we compute the lifted effects for each group:

$$\begin{aligned} \mathcal{E}_p^+ &= \{\sigma \mid \sigma \in \varSigma_p'^{(i)}, \sigma \notin \varSigma_p^{(i)}\} && (8.13) \\ \mathcal{E}_p^- &= \{\sigma \mid \sigma \in \varSigma_p^{(i)}, \sigma \notin \varSigma_p'^{(i)}\}. && (8.14) \end{aligned}$$

Relational effects $\mathcal{E}_r^+$ and $\mathcal{E}_r^-$ are computed similarly. If lifted effects are not the same for all samples in a group (i.e., a stochastic environment setting), we select the most frequent lifted effect for each group. This completes our operator definition:

$$\begin{aligned} \phi_i(\varSigma_p, \varSigma_r; \alpha_i) &= (\varSigma'_p, \varSigma'_r) && (8.15) \\ \varSigma'_p &= \varSigma_p \cup \mathcal{E}_p^+ \setminus \mathcal{E}_p^- && (8.16) \\ \varSigma'_r &= \varSigma_r \cup \mathcal{E}_r^+ \setminus \mathcal{E}_r^-. && (8.17) \end{aligned}$$

However, with this strategy, the number of groups increases with the number of objects. On the other hand, most of the time, only a subset of precondition symbols are relevant for a given action. For instance, if the action is to pick and place an object on top of another, then precondition symbols of other objects are irrelevant.

Therefore, we only consider a subset of precondition symbols relevant to the action. Although determining which symbols are relevant is difficult to answer in a general sense, a practical and generally valid heuristic is to consider topological neighborhood or contact relations. In our experiments, we define this relevance as objects that are in the action arguments and objects that are in contact with these argument objects. A broader topological relevance alternative can also be to consider objects in the vicinity of action arguments.

```
(:action o1_0_o0_1_i19_c736
 :parameters (?o0 ?o1)
 :precondition (and
    (not (= ?o0 ?o1))
    (not_p0 ?o0) (not_p0 ?o1)
    (r0 ?o0 ?o0) (r0 ?o0 ?o1)
    (r0 ?o1 ?o0) (r0 ?o1 ?o1)
    (not_r1 ?o0 ?o0) (not_r1 ?o1 ?o1)
    (r2 ?o0 ?o0) (r2 ?o1 ?o1))
 :effect (and
    (not_r1 ?o0 ?o1)
    (r2 ?o1 ?o0)))
```

Figure 8.2. An example generated PDDL action schema. This action encodes pick-place($?o1$, ‘center’, $?o0$, ‘right’) where $?o0$ and $?o1$ are free variables. The action schema is the 20th most frequent action in the dataset, with 736 occurrences out of 160K samples.

8.3.3. Translating Operators to PDDL

Each operator ϕ_i is translated into a PDDL action schema where Σ_p and Σ_r are used as preconditions, $\mathcal{E}_p^+$ and $\mathcal{E}_r^+$ are used as effects, free variables that appear in the precondition and/or action are used as parameters, and the action name is defined by the action arguments. In the action schema, each $\sigma_{p_i}(?x)$ appear as `(p`i `?x)` or `(not_p`i `?x)`, depending on the value of the predicate. We filter out action schemas that are used less than a threshold, which we set to 50 in our experiments. An example action schema is shown in Figure 8.2. We observed that the most frequently used action schemas are empty actions, such as picking a short cube from the left, which results in no effect. This is due to the exploration process where we execute random actions in the environment that frequently result in no effect. Even though these definitions would not help in the planning process, we choose to keep them as they can be used in later exploration stages to avoid actions that do not have any effect and, thus, are not interesting for the agent.

8.4. Experiments

8.4.1. Experiment Setup

We conducted our experiments in a tabletop object stacking environment (see top left in Figure 8.1). There are two object types: 5x5x5 sized short block and 5x25x5cm sized long block. An environment instance contains two to four objects represented by pose and type. A UR10 robot arm has a single high-level action with four discrete parameters: picking an object from the left, right, or center of the object and releasing it on top of (or left of, right of) another object.

We followed the data collection procedure in the previous study [55], where the robot executes random actions in the environment. The difference is that we only record objects that are either action arguments or in contact with them. These object features $\{\mathbf{x}_i\}_{i=1}^{k}$ are used as the state vector. The effect vector $\mathbf{e}_i$ for object o_i is the difference between the next state $\mathbf{x}'_i$ and the current state $\mathbf{x}_i$. We subtract the lateral movement of the arm from the effect vector to remove the effect of the carry action. Otherwise, symbols need to encode global position information of objects, which is not necessary for planning since actions are already parameterized over objects. Note that state vectors can also be selected as raw pixels [46], [53]. We collect 200K samples and split them into 160K training, 20K validation, and 20K test samples.

We compare our method with DeepSym [46] and Attentive DeepSym [53] in terms of effect prediction accuracy, and only with Attentive DeepSym for planning performance. The original DeepSym formulation uses a fixed-sized vector to represent the environment's state, making it hard for us to compare the performance of the two methods. However, the comparison with the attentive formulation already serves as a valuable measure to assess whether explicitly learning relational symbols helps the planning performance.

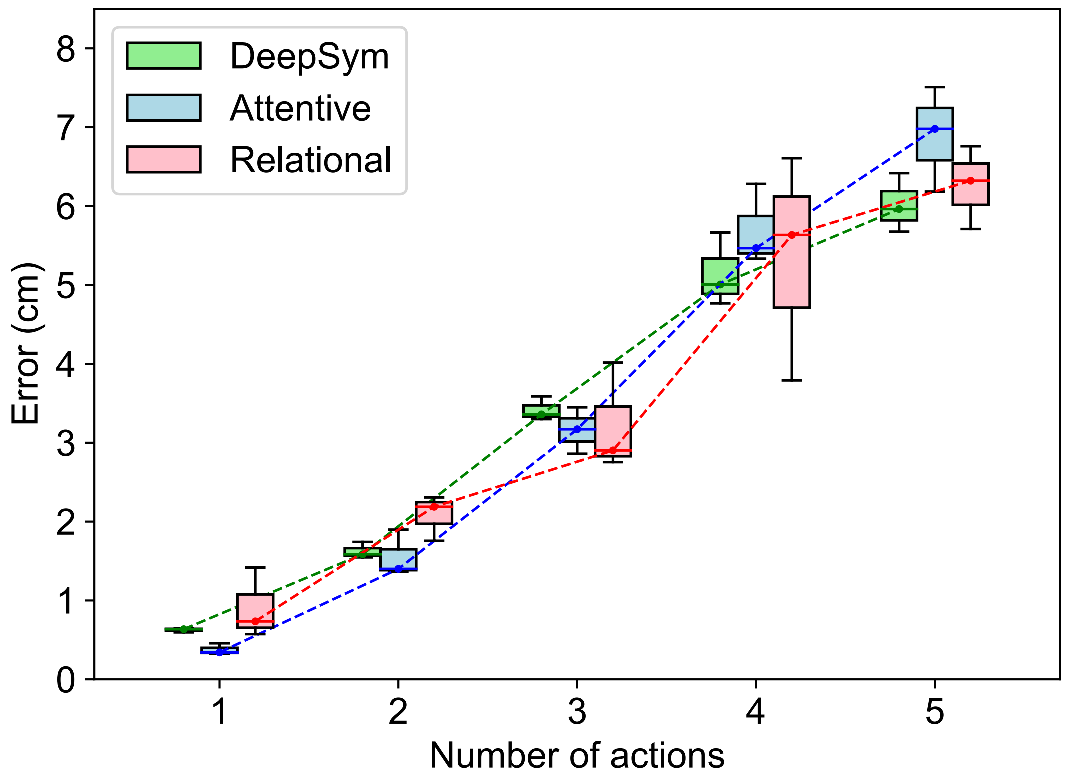


Figure 8.3. Effect prediction errors for different numbers of actions.

We train all methods for 4000 epochs with a batch size of 128 and a learning rate of 0.0001 using Adam optimizer [7]. MLP blocks consist of two layers with 128 hidden units. In Relational DeepSym (ours), we set the number of relation types to three and the object symbol dimension to one (i.e., the encoder's output dimensionality). However, for other baselines, we set the object symbol dimension to four since these methods need more representational capacity to encode the state of the environment without relational symbols. Also, the number of attention heads for the attentive formulation [53] is set to four. For layers before GS activation, we normalize both the input and the weight vectors [128] to have a norm of three. This prevents the vanishing gradients in the GS function. Lastly, we clip gradients by their norm to 10.

8.4.2. Effect Prediction Results

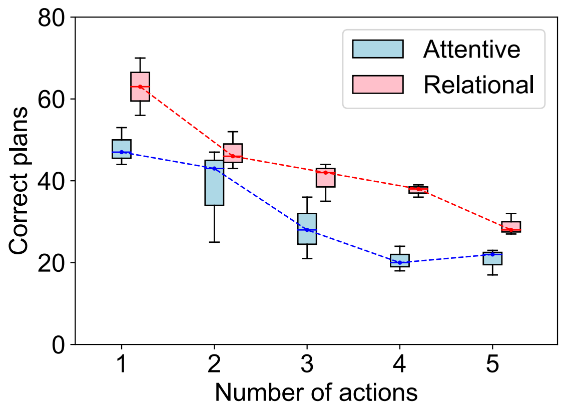
We report the test set effect prediction results in Table 8.1. Relational DeepSym performs better than other methods by a small margin, which aligns with the results in the previous study [55]. In Figure 8.3, we compare these methods by their cumulative effect prediction error when predicting a sequence of actions by feeding the prediction back into the state in an autoregressive fashion. Even though we do not see a significant difference between different methods, this does not directly translate to planning performance. Attentive DeepSym uses transformer layers to pass information between object symbols, and there is no straightforward way of translating the relational knowledge embedded in the transformer weights into lifted operators. This is the key advantage of the Relational DeepSym as it directly encodes relations symbols together with object symbols.

Table 8.1. Effect prediction results averaged over three runs. Units are in centimeters.

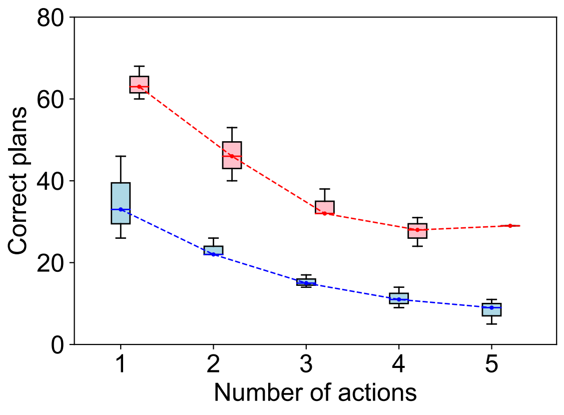
DeepSym	Attentive	Relational
4.79 ± 0.12	4.47 ± 0.10	3.21 ± 0.30

8.4.3. Planning Performance

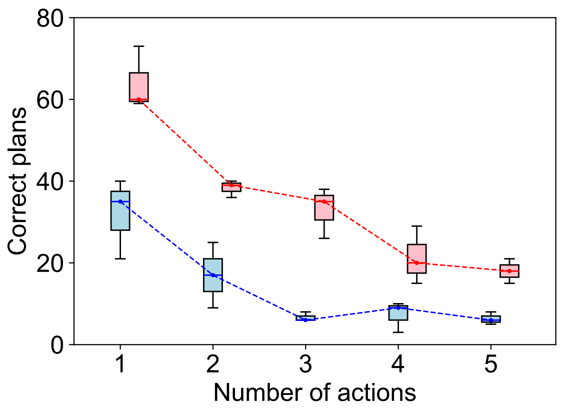
In this section, we compare the planning performance of Relational DeepSym with the attentive formulation [53]. We generate a random set of problem pairs $\{(\mathbf{X}_0^{(i)}, \mathbf{X}_g^{(i)})\}_{i=1}^{N}$ by executing random actions on the environment. For each problem pair, we convert $\mathbf{X}_0$ and $\mathbf{X}_g$ into their symbolic counterparts Σ_{p_0}, Σ_{r_0} and Σ_{p_g}, Σ_{r_g}, and produce PDDL problem statements. We filter out relations for object pairs that are not in contact in the goal state; otherwise, the planner might fail to find a plan due to spurious relations. We use the Fast Downward planning system [129] and set a timeout limit to 10 seconds. We automatically check whether a plan is correct by computing cartesian distances of objects from the goal state and accept it as correct if the difference is less than 5cm for all objects.



(a)



(b)



(c)

Figure 8.4. The planning performance for different numbers of objects over three runs with 100 random problem pairs. (a) Two objects, (b) three objects, (c) four objects.

Figure 8.4 shows the planning performance for different numbers of objects and actions. Planning with the domain defined over unary and relational symbols generated by Relational DeepSym performs significantly better than the implicit attentive version in which all relational information remains hidden in the network's weights.

We also test our method in a real-world experiment with the same environment definition (Figure 8.5). We detect the locations of objects with an Intel Realsense depth camera by clustering pixels and transforming them into the robot frame. We manually define a goal configuration with its corresponding contact graph. The rest of the algorithm works the same as in simulation experiments.

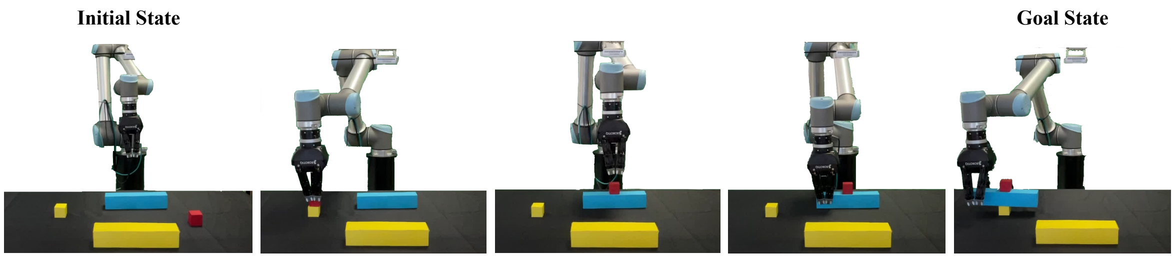


Figure 8.5. Given an initial environment configuration in the first column, our model can generate an action sequence reaching the goal state.

Another advantage of using relational symbols in addition to unary symbols is that we can represent an environment configuration that has many more objects than the ones in the training set since the number of objects does not affect the encoded representation since relational symbols are computed independently due to Gumbel-sigmoid instead of a softmax. This is not the case for the attentive formulation [53] as the model is biased towards the number of tokens in the training set, similar to other transformer-based architectures. In Figure 8.6, we initialize the scene with eight objects and create a goal configuration that was not in the train set. The planner successfully finds an action sequence that reaches the goal state.

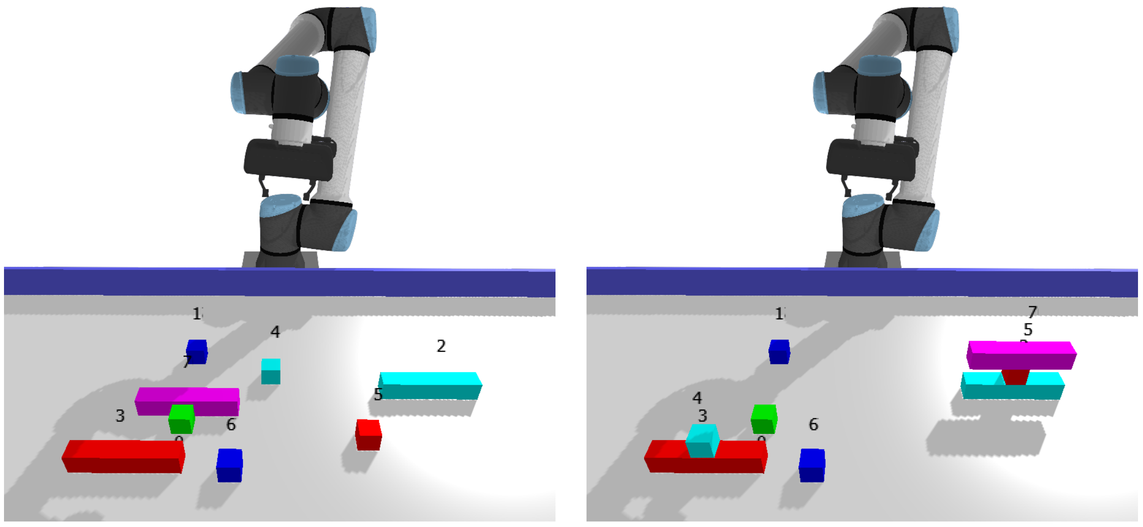


Figure 8.6. Given an initial (left) and a goal state (right), Relational DeepSym can find a plan to achieve the goal even though it is only trained with two to four objects.

8.5. Discussion

In our experiments, we define *relevance* as objects that are in the action arguments and objects that are in contact with these argument objects. An alternative relevance might be to consider objects in the vicinity of action arguments. Although this creates a limitation, most of our daily activities can be represented with this relevance. An example exception is controlling a race car with a joystick; learning the relevance between such objects should be treated as a separate problem.

8.6. Conclusion

In this chapter, we proposed and implemented a framework where object and relational predicates are discovered from the continuous sensory experience of the robot, symbolic rules that encode the transition dynamics that involve pick and place actions on arbitrary numbered compound objects are extracted, these rules are automatically transferred to PDDL and symbolic plans are generated and executed to achieve various goals. We showed that the planning performance of our framework significantly outperforms the baselines. The key factor for the performance increase is representing the environment with relational symbols in addition to object symbols.

9. DISCUSSION

The motivation of this thesis is to combine the advantages of neural networks in function approximation with classical search techniques where one can search for long-horizon plans once the domain is represented with appropriate symbols. In this thesis, we choose to learn object and action symbols that are helpful in making accurate effect predictions. We start by assuming that (1) the robot already has some basic movement primitives oriented toward objects, such as picking an object; (2) it can recognize the positions of objects and thereby compute the effect of its action; (3) and the number of objects that the executed action manipulates is fixed. In Chapter 4, we showed that, under these assumptions, the system can learn useful symbols that can be used for domain-independent planning. We removed the assumption of a fixed number of objects in Chapter 5 and showed a more appropriate formulation with not only object symbols but their relations in Chapter 6. Next, in Chapter 7, we showed how this system can be bootstrapped with useful skills from random parameterized actions that can be used in further stages of learning, partially addressing the first assumption. Lastly, we showed how to build rules defined over the learned symbols in Chapter 8, which allowed domain-independent planning.

The main supervision signal for this system is the *effect* that the robot observes after executing an action. In our experiments, we define the effect as the cartesian displacement of objects and assume the robot can compute this effect by recognizing the positions of objects before and after executing the action. While recognizing object positions might be a mild assumption, the definition of the effect directly changes what symbols can be learned in the given domain. For example, with the current effect definition, the robot cannot develop the necessary symbols that would explain "the pixels in the display change when pushed buttons on the remote", as no object changes its position. On the other hand, if we change our effect representation to pure pixel differences instead of object displacements (as we did in Section 4.5), then the robot needs to learn different symbols for objects with different colors and shapes even though they might result in the same cartesian displacement. This is a point where the definition of the effect introduces an inductive bias in the system. From a practical viewpoint, we believe that the environment should be assumed to consist of a finite set of entities, and actions manipulate the properties and the relations of these entities. A promising future direction is to use slot-attention-based methods [111], [130] trained with a contrastive objective (e.g., treat state–action–effect tuples as positive examples and shuffle them for negatives) to partition the state into multiple slots, where these slots emergently bind with entities in the environment. Presumably, such a contrastive training would allow semantically meaningful algebraic operations in the latent state-space (akin to Word2Vec examples such as $\texttt{King} - \texttt{Man} + \texttt{Woman} = \texttt{Queen}$ [131]) that can be used as effects. This would remove the need for manually defining the effect.

Another consideration regarding the effect is its resolution over time. In this thesis, we consider actions to be high-level movement primitives that extend throughout time, e.g., the robot hand goes near an object and grasps it. Effects are assumed to be one-step state differences after the action execution. However, one can consider the effects of multiple timesteps (e.g., applying several actions and observing their effects) while training the proposed system, similar to Xu *et al.* [109] where they learn affordances that consider several steps. This expands the effect space (as multiple effects stack) and, in turn, increases the number of symbols required to be learned. The advantage is that rules defined over these symbols can cover transitions that require several actions, reducing the depth of the search. We believe that there is a fundamental trade-off between the definition of the effect space and the number of learned symbols (and, in turn, the number of rules). The more we *learn* (by considering more effects), the less we *search* (since the tree gets wider and more shallow), and vice versa.

Throughout the thesis, we mainly used object positions and top-down images as the sensory input to form symbolic representations. As an alternative, one can possibly use proprioceptive sensors (sensors that relate to the embodiment of the robot, e.g., touch, force/torque sensors) to define the effect of actions, especially in earlier stages to learn skills that interact with objects [43]. With proprioceptive signals, the robot can easily filter out actions that do not interact with anything in the environment, which would result in a more sample efficient exploration.

In our experiment setups, we first collect a data set of samples by randomly interacting with the environment using the agent's action repertoire, then train the network with the collected data. This is a one-step process; however, for lifelong learning, we would need a system that iteratively collects data and gets trained on it since it is virtually impossible to collect one large independently and identically distributed (IID) data set of interactions (as we arguably have for image and text modalities). If we accept that a robot cannot collect IID data at a given time, then the data collection process should consider previously learned representations to avoid unnecessary interactions for sample efficiency. In this regard, open-ended learning methods [132, 133] and curiosity-based methods [124], [134–136] hold a potential for collecting novel, interesting samples.

Furthermore, if the robot will develop its representations in an explore-and-train loop, then the learned representations should be compositional through multiple generations of training. On the other hand, neural networks are not very suited for composition as the learned representations are distributed [137], making it challenging to copy and use a partition of the network. We argue that the neurosymbolic solution proposed in this thesis is better suited for lifelong learning as one can cascade previously learned symbols as shown in Section 4.4.3 where the system learns 'larger-than' relation using the learned object symbols. Given a new interaction data set, if a decoder can successfully predict the effects from previously learned symbols, then introducing a new encoder would only marginally reduce the prediction error. Thus, the system allows for building new rules induced by the new interaction data set without defining new symbols. In other words, expanding the knowledge base of the agent is possible without necessarily expanding the vocabulary (i.e., the symbols), which might be an easier approach for lifelong learning when compared with retraining a large neural network susceptible to catastrophic forgetting [138], since it is not necessary to learn new symbols from scratch.

In this thesis, we represent problems in PDDL and use AI planners (mGPT for probabilistic planning [107] in Chapter 4 and Fast Downward for classical planning in Chapter 8) to search for a solution. Once rules defined over the learned symbols are available, one can resort to other search methods such as Monte Carlo tree search [139], or even simple A* search. A promising future direction to guide the search with heuristic-based planners is to learn symbols in a way that encodes the similarity between them, as in Asai and Muise [77] where they use a prior over the latent space that forces state transitions to flip only a couple of binary units.

Collecting experience with robots is slow and cumbersome, especially in the real world, when compared with other machine learning domains. Therefore, a scalable robot learning solution is only possible with sample-efficient methods. Sample efficiency, on the other hand, mostly comes with assumptions and restrictions. For instance, in Chapter 7, the robot can learn skills that manipulate objects only if it starts near the objects. In Chapter 8, we assume that the robot manipulates only the objects that are in the action argument or in contact with them, which narrows down the symbolic effect space. While the current machine learning paradigm is trying to make as few assumptions as possible by employing a single large monolithic network with a generic objective, robot learning should do the opposite—make as many (preferably causal) assumptions about the environment and the embodiment as possible in order to reduce the sample complexity to an operable level.

10. CONCLUSION

In this thesis, we propose a framework to learn symbolic representations for objects from the continuously represented interaction experience of the robot. In this framework, we have an encoder-decoder network style with binarized activations in the bottleneck layer that preserves differentiability with an approximation. The encoder takes the state as input, which is represented as a set of object features, and generates symbolic outputs. Together with the action encoding (either given or learned), the decoder predicts the effect of the executed action. After training, the activations in the bottleneck layer can be used as symbols for objects, which allows us to convert the domain into a symbolic one by building rules defined over the learned symbols. Once these rules are translated into a domain-specific language, in our case PDDL, we can run off-the-shelf AI planners to find a solution to go from an initial state to a goal state; both expressed symbolically with the help of the encoder. The main idea is to separate the problem of perception (the neural network part) from reasoning and problem-solving (search with rules). In the end, the agent's problem-solving capability is built on two main pillars: learning frequently occurring patterns in a predictive fashion with neural networks and searching for solutions using the learned knowledge.

In Chapter 3, we investigate how to extract high-level units from a trained network such that they can be used in newly introduced tasks. Our initial idea is presented in Chapter 4, where we introduce DeepSym, and the proceeding chapters progressively improve the DeepSym architecture. In Chapter 5, we employ self-attention layers after the encoder to propagate information from multiple object symbols to handle multi-object effects. Later in Chapter 6, we propose a better formulation for the same problem where another encoder explicitly outputs symbolic relations between objects. In Chapter 7, we show how high-level skills can be learne by extending the same pipeline with an action encoder, removing the assumption of pre-defined high-level actions. Lastly, in Chapter 8, we showed how object and relational symbols allow us to build rules defined over them that can be translated into PDDL action schemas for planning.

REFERENCES

1. Glorot, X. and Y. Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks", *International Conference on Artificial Intelligence and Statistics*, Sardinia, Italy, Vol. 13, pp. 249–256, 2010.

2. He, K., X. Zhang, S. Ren and J. Sun, "Delving Deep Into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification", *IEEE International Conference on Computer Vision*, Santiago, Chile, pp. 1026–1034, 2015.

3. Le, Q. V., N. Jaitly and G. E. Hinton, "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units", arXiv 1504.00941, 2015.

4. Mishkin, D. and J. Matas, "All You Need is a Good Init", arXiv 1511.06422, 2015.

5. Duchi, J., E. Hazan and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research*, Vol. 12, No. 7, pp. 2121–2159, 2011.

6. Tieleman, T. and G. Hinton, "Lecture 6.5-Rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude", Coursera: Neural Networks for Machine Learning, 2012.

7. Kingma, D. P. and J. Ba, "Adam: A Method for Stochastic Optimization", arXiv 1412.6980, 2014.

8. Ioffe, S. and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", arXiv 1502.03167, 2015.

9. Ba, J. L., J. R. Kiros and G. E. Hinton, "Layer Normalization", arXiv 1607.06450, 2016.

10. Miyato, T., T. Kataoka, M. Koyama and Y. Yoshida, “Spectral Normalization for Generative Adversarial Networks”, *International Conference on Learning Representations*, Vancouver, Canada, 2018.

11. Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database”, *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Miami, Florida, USA, pp. 248–255, 2009.

12. Lin, T.-Y., M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár and C. L. Zitnick, “Microsoft COCO: Common Objects in Context”, *European Conference on Computer Vision*, Zurich, Switzerland, Vol. 13/5, pp. 740–755, 2014.

13. Krizhevsky, A., I. Sutskever and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, *Neural Information Processing Systems*, Lake Tahoe, Nevada, USA, Vol. 25, pp. 1097–1105, 2012.

14. Simonyan, K. and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, arXiv 1409.1556, 2014.

15. He, K., X. Zhang, S. Ren and J. Sun, “Identity Mappings in Deep Residual Networks”, *European Conference on Computer Vision*, Amsterdam, The Netherlands, Vol. 14/4, pp. 630–645, 2016.

16. He, K., X. Chen, S. Xie, Y. Li, P. Dollár and R. Girshick, “Masked Autoencoders are Scalable Vision Learners”, *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, New Orleans, Louisiana, USA, pp. 16000–16009, 2022.

17. Ronneberger, O., P. Fischer and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation”, *Medical Image Computing and Computer-Assisted Intervention International Conference*, Munich, Germany, Vol. 18/3, pp. 234–241, 2015.

18. Long, J., E. Shelhamer and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation", *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Boston, Massachusetts, USA, pp. 3431–3440, 2015.

19. Kirillov, A., E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollar and R. Girshick, "Segment Anything", *IEEE/CVF International Conference on Computer Vision*, Paris, France, pp. 4015–4026, 2023.

20. Sutskever, I., O. Vinyals and Q. V. Le, "Sequence to Sequence Learning with Neural Networks", arXiv 1409.3215, 2014.

21. Devlin, J., M. Chang, K. Lee and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", arXiv 1810.04805, 2018.

22. Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning", arXiv 1312.5602, 2013.

23. Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis, "Human-Level Control through Deep Reinforcement Learning", *Nature*, Vol. 518, No. 7540, pp. 529–533, 2015.

24. Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis, "Mastering the Game of Go with Deep Neural Networks and Tree Search", *Nature*, Vol. 529, No. 7587, pp. 484–489, 2016.

25. Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm", arXiv 1712.01815, 2017.

26. Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, "Attention Is All You Need", *Neural Information Processing Systems*, Long Beach, California, USA, Vol. 31, pp. 6000–6010, 2017.

27. Radford, A., K. Narasimhan, T. Salimans and I. Sutskever, "Improving Language Understanding by Generative Pre-Training", 2018, https://openai.com/research/language-unsupervised, accessed on January 29, 2024.

28. Radford, A., J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, "Language Models Are Unsupervised Multitask Learners", 2019, https://openai.com/research/better-language-models, accessed on January 29, 2024.

29. Brown, T., B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever and D. Amodei, "Language Models Are Few-Shot Learners", *Neural Information Processing Systems*, Virtual, Vol. 34, pp. 1877–1901, 2020.

30. OpenAI, "GPT-4 Technical Report", arXiv 2303.08774, 2023.

31. Ouyang, L., J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. E. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike and R. J. Lowe, "Training Language Models to Follow Instructions with Human Feedback", *Neural Information Processing Systems*, New Orleans, Louisiana, USA, Vol. 36, pp. 27730–

27744, 2022.

32. Touvron, H., T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave and G. Lample, “Llama: Open and Efficient Foundation Language Models”, arXiv 2302.13971, 2023.

33. Touvron, H., L. Martin, K. R. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. M. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov and T. Scialom, “Llama 2: Open Foundation and Fine-Tuned Chat Models”, arXiv 2307.09288, 2023.

34. Bommasani, R., D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, E. Brynjolfsson, S. Buch, D. Card, R. Castellon, N. S. Chatterji, A. S. Chen, K. A. Creel, J. Davis, D. Demszky, C. Donahue, M. Doumbouya, E. Durmus, S. Ermon, J. Etchemendy, K. Ethayarajh, L. Fei-Fei, C. Finn, T. Gale, L. E. Gillespie, K. Goel, N. D. Goodman, S. Grossman, N. Guha, T. Hashimoto, P. Henderson, J. Hewitt, D. E. Ho, J. Hong, K. Hsu, J. Huang, T. F. Icard, S. Jain, D. Jurafsky, P. Kalluri, S. Karamcheti, G. Keeling, F. Khani, O. Khattab, P. W. Koh, M. S. Krass, R. Krishna, R. Kuditipudi, A. Kumar, F. Ladhak, M. Lee, T. Lee, J. Leskovec, I. Levent, X. L. Li, X. Li, T. Ma, A. Malik, C. D. Manning, S. P. Mirchandani, E. Mitchell, Z. Munyikwa, S. Nair, A. Narayan, D. Narayanan, B. Newman, A. Nie, J. C. Niebles, H. Nilforoshan, J. F. Nyarko, G. Ogut, L. Orr, I. Papadimitriou, J. S. Park,

C. Piech, E. Portelance, C. Potts, A. Raghunathan, R. Reich, H. Ren, F. Rong, Y. H. Roohani, C. Ruiz, J. Ryan, C. R'e, D. Sadigh, S. Sagawa, K. Santhanam, A. Shih, K. P. Srinivasan, A. Tamkin, R. Taori, A. W. Thomas, F. Tramèr, R. E. Wang, W. Wang, B. Wu, J. Wu, Y. Wu, S. M. Xie, M. Yasunaga, J. You, M. A. Zaharia, M. Zhang, T. Zhang, X. Zhang, Y. Zhang, L. Zheng, K. Zhou and P. Liang, "On the Opportunities and Risks of Foundation Models", arXiv 2108.07258, 2021.

35. Celik, B., A. Ahmetoglu, E. Ugur and E. Oztop, "Developmental Scaffolding with Large Language Models", *International Conference on Development and Learning*, Macau, China, pp. 396–402, 2023.

36. Searle, J. R., "The Chinese Room Revisited", *Behavioral and Brain Sciences*, Vol. 5, No. 2, pp. 345–348, 1982.

37. Harnad, S., "The Symbol Grounding Problem", *Physica D: Nonlinear Phenomena*, Vol. 42, No. 1-3, pp. 335–346, 1990.

38. Konidaris, G., "On the Necessity of Abstraction", *Current Opinion in Behavioral Sciences*, Vol. 29, pp. 1–7, 2019.

39. Konidaris, G., L. P. Kaelbling and T. Lozano-Perez, "Constructing Symbolic Representations for High-Level Planning", *AAAI Conference on Artificial Intelligence*, Québec City, Québec, Canada, Vol. 28, pp. 1932–1940, 2014.

40. Ha, D. and J. Schmidhuber, "World Models", arXiv 1803.10122, 2018.

41. Hafner, D., T. P. Lillicrap, M. Norouzi and J. Ba, "Mastering Atari with Discrete World Models", *International Conference on Learning Representations*, Virtual, 2020.

42. Ugur, E. and J. Piater, "Bottom-up Learning of Object Categories, Action Effects and Logical Rules: From Continuous Manipulative Exploration to Symbolic

Planning", *IEEE International Conference on Robotics and Automation*, Seattle, WA, USA, pp. 2627–2633, 2015.

43. Ugur, E., Y. Nagai, E. Sahin and E. Oztop, "Staged Development of Robot Skills: Behavior Formation, Affordance Learning and Imitation with Motionese", *IEEE Transactions on Autonomous Mental Development*, Vol. 7, No. 2, pp. 119–139, 2015.

44. James, S., B. Rosman and G. Konidaris, "Learning Portable Representations for High-Level Planning", *International Conference on Machine Learning*, Vienna, Austria, Vol. 37, pp. 4682–4691, 2020.

45. Ugur, E. and J. Piater, "Refining Discovered Symbols with Multi-Step Interaction Experience", *International Conference on Humanoid Robots*, Seoul, Korea, Vol. 15, pp. 1007–1012, 2015.

46. Ahmetoglu, A., M. Y. Seker, J. Piater, E. Oztop and E. Ugur, "DeepSym: Deep Symbol Generation and Rule Learning for Planning From Unsupervised Robot Interaction", *Journal of Artificial Intelligence Research*, Vol. 75, pp. 709–745, 2022.

47. Zhuang, F., Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong and Q. He, "A Comprehensive Survey on Transfer Learning", *Proceedings of the IEEE*, Vol. 109, No. 1, pp. 43–76, 2020.

48. Hospedales, T., A. Antoniou, P. Micaelli and A. Storkey, "Meta-Learning in Neural Networks: A Survey", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 44, No. 9, pp. 5149–5169, 2021.

49. Silver, D. L., Q. Yang and L. Li, "Lifelong Machine Learning Systems: Beyond Learning Algorithms", *2013 AAAI Spring Symposium Series*, Palo Alto, California, USA, 2013.

50. Ahmetoglu, A., E. Ugur, M. Asada and E. Oztop, “High-Level Features for Resource Economy and Fast Learning in Skill Transfer”, *Advanced Robotics*, Vol. 36, No. 5-6, pp. 291–303, 2022.

51. Maddison, C. J., A. Mnih and Y. W. Teh, “The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables”, arXiv 1611.00712, 2016.

52. Jang, E., S. Gu and B. Poole, “Categorical Reparameterization with Gumbel-Softmax”, arXiv 1611.01144, 2016.

53. Ahmetoglu, A., E. Oztop and E. Ugur, “Learning Multi-Object Symbols for Manipulation with Attentive Deep Effect Predictors”, arXiv 2208.01021, 2022.

54. Ahmetoglu, A., E. Oztop and E. Ugur, “Deep Multi-Object Symbol Learning with Self-Attention Based Predictors”, *Signal Processing and Communications Applications Conference*, İstanbul, Türkiye, Vol. 31, pp. 1–4, 2023.

55. Ahmetoglu, A., B. Celik, E. Oztop and E. Ugur, “Discovering Predictive Relational Object Symbols with Symbolic Attentive Layers”, *IEEE Robotics and Automation Letters*, Vol. 9, No. 2, pp. 1977–1984, 2024.

56. Ahmetoglu, A., E. Oztop and E. Ugur, “Symbolic Manipulation Planning with Discovered Object and Relational Predicates”, arXiv 2401.01123, 2024.

57. Kuipers, B., E. A. Feigenbaum, P. E. Hart and N. J. Nilsson, “Shakey: From Conception to History”, *AI Magazine*, Vol. 38, No. 1, pp. 88–103, 2017.

58. Murphy, R. R., *Introduction to AI Robotics*, MIT Press, Cambridge, 2000.

59. Klingspor, V., K. J. Morik and A. D. Rieger, “Learning Concepts from Sensor Data of a Mobile Robot”, *Machine Learning*, Vol. 23, No. 2-3, pp. 305–332, 1996.

60. Petrick, R., D. Kraft, K. Mourao, N. Pugeault, N. Krüger and M. Steedman,

"Representation and Integration: Combining Robot Control, High-Level Planning, and Action Learning", *International Cognitive Robotics Workshop*, Patras, Greece, Vol. 6, pp. 32–41, 2008.

61. Mourao, K., R. P. Petrick and M. Steedman, "Using Kernel Perceptrons to Learn Action Effects for Planning", *International Conference on Cognitive Systems*, Washington, DC, USA, pp. 45–50, 2008.

62. Wörgötter, F., A. Agostini, N. Krüger, N. Shylo and B. Porr, "Cognitive Agents—A Procedural Perspective Relying on the Predictability of Object-Action-Complexes (OACs)", *Robotics and Autonomous Systems*, Vol. 57, No. 4, pp. 420–432, 2009.

63. Kulick, J., M. Toussaint, T. Lang and M. Lopes, "Active Learning for Teaching a Robot Grounded Relational Symbols", *International Joint Conference on Artificial Intelligence*, Beijing, China, Vol. 23, pp. 1451–1457, 2013.

64. Sun, R., "Symbol Grounding: A New Look at an Old Idea", *Philosophical Psychology*, Vol. 13, No. 2, pp. 149–172, 2000.

65. Pisokas, J. and U. Nehmzow, "Experiments in Subsymbolic Action Planning with Mobile Robots", *Adaptive Agents and Multi-Agent Systems II*, pp. 80–87, Springer, 2005.

66. Ugur, E., E. Oztop and E. Sahin, "Goal Emulation and Planning in Perceptual Space Using Learned Affordances", *Robotics and Autonomous Systems*, Vol. 59, No. 7–8, pp. 580–595, 2011.

67. Ozturkcu, O. B., E. Ugur and E. Oztop, "High-Level Representations through Unconstrained Sensorimotor Learning", *International Conference on Development and Learning and Epigenetic Robotics*, Valparaiso, Chile, Vol. 10, pp. 1–6, 2020.

68. Mota, T. and M. Sridharan, "Commonsense Reasoning and Knowledge Acquisi-

tion to Guide Deep Learning on Robots", *Robotics: Science and Systems*, Messe Freiburg, Freiburg, Germany, Vol. 15, 2019.

69. Sridharan, M. and H. Riley, "Integrating Deep Learning and Non-monotonic Logical Reasoning for Explainable Visual Question Answering", *European Conference on Multi-Agent Systems*, Thessaloniki, Greece, Vol. 17, pp. 558–570, 2020.

70. Law, M., A. Russo and K. Broda, "The Complexity and Generality of Learning Answer Set Programs", *Artificial Intelligence*, Vol. 259, pp. 110–146, 2018.

71. Taniguchi, T., E. Ugur, M. Hoffmann, L. Jamone, T. Nagai, B. Rosman, T. Matsuka, N. Iwahashi, E. Öztop, J. H. Piater and F. Wörgötter, "Symbol Emergence in Cognitive Developmental Systems: A Survey", *IEEE Transactions on Cognitive and Developmental Systems*, Vol. 11, No. 4, pp. 494–516, 2019.

72. Konidaris, G., L. P. Kaelbling and T. Lozano-Perez, "Symbol Acquisition for Probabilistic High-Level Planning", *International Joint Conference on Artificial Intelligence*, Buenos Aires, Argentina, Vol. 24, pp. 3619–3627, 2015.

73. Konidaris, G., L. P. Kaelbling and T. Lozano-Perez, "From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning", *Journal of Artificial Intelligence Research*, Vol. 61, pp. 215–289, 2018.

74. James, S., B. Rosman and G. Konidaris, "Autonomous Learning of Object-Centric Abstractions for High-Level Planning", *International Conference on Learning Representations*, Virtual, 2021.

75. Pelleg, D. and A. W. Moore, "X-Means: Extending K-Means with Efficient Estimation of the Number of Clusters", *International Conference on Machine Learning*, Stanford, California, USA, Vol. 1, pp. 727–734, 2000.

76. Asai, M. and A. Fukunaga, "Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary", arXiv 1705.00154, 2017.

77. Asai, M. and C. Muise, “Learning Neural-Symbolic Descriptive Planning Models via Cube-Space Priors: The Voyage Home (to STRIPS)”, arXiv 2004.12850, 2020.

78. Asai, M., H. Kajino, A. Fukunaga and C. Muise, “Classical Planning in Deep Latent Space”, *Journal of Artificial Intelligence Research*, Vol. 74, pp. 1599–1686, 2022.

79. Gibson, J. J., *The Ecological Approach to Visual Perception*, Houghton Mifflin, Boston, 1979.

80. Zech, P., S. Haller, S. R. Lakani, B. Ridge, E. Ugur and J. Piater, “Computational Models of Affordance in Robotics: A Taxonomy and Systematic Classification”, *Adaptive Behavior*, Vol. 25, No. 5, pp. 235–271, 2017.

81. Silver, T., R. Chitnis, J. Tenenbaum, L. P. Kaelbling and T. Lozano-Pérez, “Learning Symbolic Operators for Task and Motion Planning”, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Prague, Czech Republic, pp. 3182–3189, 2021.

82. Chitnis, R., T. Silver, J. B. Tenenbaum, T. Lozano-Perez and L. P. Kaelbling, “Learning Neuro-Symbolic Relational Transition Models for Bilevel Planning”, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Kyoto, Japan, pp. 4166–4173, 2022.

83. Silver, T., A. Athalye, J. B. Tenenbaum, T. Lozano-Perez and L. P. Kaelbling, “Learning Neuro-Symbolic Skills for Bilevel Planning”, *Conference on Robot Learning*, Auckland, New Zealand, pp. 701–714, 2022.

84. Silver, T., R. Chitnis, N. Kumar, W. McClinton, T. Lozano-Perez, L. P. Kaelbling and J. Tenenbaum, “Inventing Relational State and Action Abstractions for Effective and Efficient Bilevel Planning”, arXiv 2203.09634, 2022.

85. Silver, T., R. Chitnis, N. Kumar, W. McClinton, T. Lozano-Pérez, L. Kaelbling

and J. B. Tenenbaum, “Predicate Invention for Bilevel Planning”, *AAAI Conference on Artificial Intelligence*, Washington, DC, USA, Vol. 37, pp. 12120–12129, 2023.

86. Achterhold, J., M. Krimmel and J. Stueckler, “Learning Temporally Extended Skills in Continuous Domains as Symbolic Actions for Planning”, *Conference on Robot Learning*, Atlanta, Georgia, USA, pp. 225–236, 2023.

87. Kumar, N., W. McClinton, R. Chitnis, T. Silver, T. Lozano-Pérez and L. P. Kaelbling, “Learning Efficient Abstract Planning Models that Choose What to Predict”, *Conference on Robot Learning*, Atlanta, Georgia, USA, 2023.

88. Yuan, W., C. Paxton, K. Desingh and D. Fox, “SORNet: Spatial Object-Centric Representations for Sequential Manipulation”, *Conference on Robot Learning*, Auckland, New Zealand, pp. 148–157, 2022.

89. Schmidhuber, J., “Driven by Compression Progress: A Simple Principle Explains Essential Aspects of Subjective Beauty, Novelty, Surprise, Interestingness, Attention, Curiosity, Creativity, Art, Science, Music, Jokes”, *Anticipatory Behavior in Adaptive Learning Systems*, pp. 48–76, 2008.

90. Wolff, J. G., “Information Compression, Multiple Alignment, and the Representation and Processing of Knowledge in the Brain”, *Frontiers in Psychology*, Vol. 7, pp. 1584–1584, 2016.

91. Wiskott, L. and T. J. Sejnowski, “Slow Feature Analysis: Unsupervised Learning of Invariances”, *Neural Computation*, Vol. 14, No. 4, pp. 715–770, 2002.

92. Goodfellow, I., Y. Bengio, A. Courville and Y. Bengio, *Deep Learning*, MIT Press, Cambridge, 2016.

93. Trefethen, L. N. and D. Bau, *Numerical Linear Algebra*, SIAM, Philadelphia, 1997.

94. LeCun, Y., L. Bottou, G. B. Orr and K.-R. Müller, “Efficient BackProp”, *Neural Networks: Tricks of the Trade*, pp. 9–50, Springer-Verlag, 1998.

95. Rohmer, E., S. P. N. Singh and M. Freese, “V-REP: A Versatile and Scalable Robot Simulation Framework”, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, Japan, pp. 1321–1326, 2013.

96. Universal Robots, “The UR10 Collaborative Industrial Robot”, 2012, https://www.universal-robots.com/products/ur10-robot, accessed on January 29, 2024.

97. Welch, B. L., “The Generalization of ‘Student’s’ Problem when Several Different Population Variances are Involved”, *Biometrika*, Vol. 34, No. 1/2, pp. 28–35, 1947.

98. Younes, H. L. and M. L. Littman, “PPDDL1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects”, Technical Report CMU-CS-04-162, 2004.

99. Ugur, E., E. Şahin and E. Oztop, “Self-Discovery of Motor Primitives and Learning Grasp Affordances”, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura, Algarve, Portugal, pp. 3260–3267, 2012.

100. Seker, M. Y., M. Imre, J. Piater and E. Ugur, “Conditional Neural Movement Primitives”, *Robotics: Science and Systems*, Messe Freiburg, Freiburg, Germany, Vol. 15, 2019.

101. Akbulut, M., E. Oztop, M. Y. Seker, X. Hh, A. Tekden and E. Ugur, “ACNMP: Skill Transfer and Task Extrapolation through Learning From Demonstration and Reinforcement Learning via Representation Sharing”, *Conference on Robot Learning*, London, UK (Virtual), pp. 1896–1907, 2021.

102. Russell, S. J. and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th Edition, Pearson, 2020.

103. Hinton, G. E. and R. R. Salakhutdinov, “Reducing the Dimensionality of Data with Neural Networks”, *Science*, Vol. 313, No. 5786, pp. 504–507, 2006.

104. Kingma, D. P. and M. Welling, “Auto-encoding Variational Bayes”, arXiv 1312.6114, 2013.

105. Bengio, Y., N. Léonard and A. C. Courville, “Estimating or Propagating Gradients through Stochastic Neurons for Conditional Computation”, arXiv 1308.3432, 2013.

106. Townsend, W., “The BarrettHand Grasper-Programmably Flexible Part Handling and Assembly”, *Industrial Robot: An International Journal*, Vol. 27, No. 3, pp. 181–188, 2000.

107. Bonet, B. and H. Geffner, “mGPT: A Probabilistic Planner Based on Heuristic Search”, *Journal of Artificial Intelligence Research*, Vol. 24, pp. 933–944, 2005.

108. Radford, A., L. Metz and S. Chintala, “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”, arXiv 1511.06434, 2015.

109. Xu, D., A. Mandlekar, R. Martín-Martín, Y. Zhu, S. Savarese and L. Fei-Fei, “Deep Affordance Foresight: Planning through What Can Be Done in the Future”, *IEEE International Conference on Robotics and Automation*, Xi’an, China, pp. 6206–6213, 2021.

110. Johnson, M., K. Hofmann, T. Hutton and D. Bignell, “The Malmo Platform for Artificial Intelligence Experimentation”, *International Joint Conference on Artificial Intelligence*, New York City, New York, USA, Vol. 25, pp. 4246–4247, 2016.

111. Locatello, F., D. Weissenborn, T. Unterthiner, A. Mahendran, G. Heigold, J. Uszkoreit, A. Dosovitskiy and T. Kipf, “Object-Centric Learning with Slot

Attention", *Neural Information Processing Systems*, Virtual, Vol. 34, pp. 11525–11538, 2020.

112. Elsayed, G. F., A. Mahendran, S. van Steenkiste, K. Greff, M. C. Mozer and T. Kipf, "SAVi++: Towards End-to-End Object-Centric Learning from Real-World Videos", arXiv 2206.07764, 2022.

113. Paraschos, A., C. Daniel, J. R. Peters and G. Neumann, "Probabilistic Movement Primitives", *Neural Information Processing Systems*, Lake Tahoe, Nevada, USA, Vol. 26, pp. 2616–2624, 2013.

114. Schaal, S., "Dynamic Movement Primitives-A Framework for Motor Control in Humans and Humanoid Robotics", *Adaptive Motion of Animals and Machines*, pp. 261–280, Springer, 2006.

115. Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga and A. Lerer, "Automatic Differentiation in PyTorch", *Neural Information Processing Systems Workshop on Autodiff*, Long Beach, California, USA, 2017.

116. Ugur, E., Y. Nagai, H. Celikkanat and E. Oztop, "Parental Scaffolding as A Bootstrapping Mechanism for Learning Grasp Affordances and Imitation Skills", *Robotica*, Vol. 33, No. 5, pp. 1163–1180, 2015.

117. Asai, M. and A. Fukunaga, "Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary", *AAAI Conference on Artificial Intelligence*, New Orleans, Louisiana, USA, Vol. 32, pp. 6094–6101, 2018.

118. Schaul, T., J. Quan, I. Antonoglou and D. Silver, "Prioritized Experience Replay", arXiv 1511.05952, 2015.

119. Ghallab, M., A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld and D. Wilkins, "Pddl—The Planning Domain Definition Language", AIPS-98

Planning Competition, 1998.

120. Rosen, E., B. M. Abbatematteo, S. Thompson, T. Akbulut and G. Konidaris, "On the Role of Structure in Manipulation Skill Learning", *Conference on Robot Learning Workshop on Learning, Perception, and Abstraction for Long-Horizon Planning*, Auckland, New Zealand, 2022.

121. Ten Pas, A., M. Gualtieri, K. Saenko and R. Platt, "Grasp Pose Detection in Point Clouds", *The International Journal of Robotics Research*, Vol. 36, No. 13-14, pp. 1455–1473, 2017.

122. Yoshikawa, T., "Manipulability of Robotic Mechanisms", *The International Journal of Robotics Research*, Vol. 4, No. 2, pp. 3–9, 1985.

123. Martín-Martín, R., M. A. Lee, R. Gardner, S. Savarese, J. Bohg and A. Garg, "Variable Impedance Control in End-Effector Space: An Action Space for Reinforcement Learning in Contact-Rich Tasks", *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Macau, China, pp. 1010–1017, 2019.

124. Ten, A., P.-Y. Oudeyer and C. Moulin-Frier, "Curiosity-Driven Exploration", *The Drive for Knowledge: The Science of Human Information Seeking*, p. 53, 2022.

125. Cully, A., J. Clune, D. Tarapore and J.-B. Mouret, "Robots That Can Adapt Like Animals", *Nature*, Vol. 521, No. 7553, pp. 503–507, 2015.

126. Stanley, K. O., "Compositional Pattern Producing Networks: A Novel Abstraction of Development", *Genetic Programming and Evolvable Machines*, Vol. 8, pp. 131–162, 2007.

127. Stanley, K. O. and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies", *Evolutionary Computation*, Vol. 10, No. 2, pp. 99–127, 2002.

128. Salimans, T. and D. P. Kingma, "Weight Normalization: A Simple Reparame-

terization to Accelerate Training of Deep Neural Networks", *Neural Information Processing Systems*, Barcelona, Spain, Vol. 30, pp. 901–909, 2016.

129. Helmert, M., "The Fast Downward Planning System", *Journal of Artificial Intelligence Research*, Vol. 26, pp. 191–246, 2006.

130. Biza, O., S. van Steenkiste, M. S. Sajjadi, G. F. Elsayed, A. Mahendran and T. Kipf, "Invariant Slot Attention: Object Discovery with Slot-Centric Reference Frames", arXiv 2302.04973, 2023.

131. Mikolov, T., I. Sutskever, K. Chen, G. S. Corrado and J. Dean, "Distributed Representations of Words and Phrases and Their Compositionality", *Neural Information Processing Systems*, Lake Tahoe, Nevada, USA, Vol. 26, pp. 3111–3119, 2013.

132. Wang, R., J. Lehman, J. Clune and K. O. Stanley, "Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions", arXiv 1901.01753, 2019.

133. Wang, G., Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan and A. Anandkumar, "Voyager: An Open-Ended Embodied Agent with Large Language Models", arXiv 2305.16291, 2023.

134. Ugur, E., M. R. Dogar, M. Cakmak and E. Sahin, "Curiosity-Driven Learning of Traversability Affordance on a Mobile Robot", *International Conference on Development and Learning*, London, UK, pp. 13–18, 2007.

135. Sancaktar, C., S. Blaes and G. Martius, "Curious Exploration via Structured World Models Yields Zero-Shot Object Manipulation", *Neural Information Processing Systems*, New Orleans, Louisiana, USA, Vol. 36, pp. 24170–24183, 2022.

136. Sancaktar, C., J. Piater and G. Martius, "Regularity as Intrinsic Reward for Free Play", *Neural Information Processing Systems*, New Orleans, Louisiana, USA,

Vol. 37, 2023.

137. Bengio, Y., “Learning Deep Architectures for AI”, *Foundations and Trends in Machine Learning*, Vol. 2, No. 1, pp. 1–127, 2009.

138. French, R. M., “Catastrophic Forgetting in Connectionist Networks”, *Trends in Cognitive Sciences*, Vol. 3, No. 4, pp. 128–135, 1999.

139. Browne, C. B., E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis and S. Colton, “A Survey of Monte Carlo Tree Search Methods”, *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 1, pp. 1–43, 2012.

140. Reddi, S. J., S. Kale and S. Kumar, “On the Convergence of Adam and Beyond”, *International Conference on Learning Representations*, Vancouver, Canada, 2018.

APPENDIX A: DEEPSYM EXTENDED RESULTS

A.1. Network Architecture and Hyperparameters

A.1.1. Tabletop Environment

The network architectures of encoders f_1 and f_2 are shown in Tables A.1 and A.2, respectively. Each convolution is followed by a batch normalization layer and ReLU activation after the normalization. The network architectures of decoders g_1 and g_2 are shown in Tables A.3 and A.4, respectively. Decoders consist of fully connected (FC) layers with no batch normalization. Adam optimizer [7] with AMSGrad [140] is used. The learning rate is set to 0.00005 with a 128 batch size. Each model is trained for 300 epochs, and we select the best model based on the mean square error.

Table A.1. Encoder f_1.

Layer	In ch.	Out ch.	Stride	Pad
Conv3x3	1	32	1	1
Conv3x3	32	32	2	1
Conv3x3	32	64	1	1
Conv3x3	64	64	2	1
Conv3x3	64	128	1	1
Conv3x3	128	128	2	1
Conv3x3	128	256	1	1
Conv3x3	256	256	2	1
Global average pooling over channels				
FC	256	2	-	-
Gumbel-sigmoid				
Number of parameters: 1,174,114				

Table A.2. Encoder f_2.

Layer	In ch.	Out ch.	Stride	Pad
Conv3x3	2	32	1	1
Conv3x3	32	32	2	1
Conv3x3	32	64	1	1
Conv3x3	64	64	2	1
Conv3x3	64	128	1	1
Conv3x3	128	128	2	1
Conv3x3	128	256	1	1
Conv3x3	256	256	2	1
Global average pooling over channels				
FC	256	1	-	-
Gumbel-sigmoid				
Number of parameters: 1,174,145				

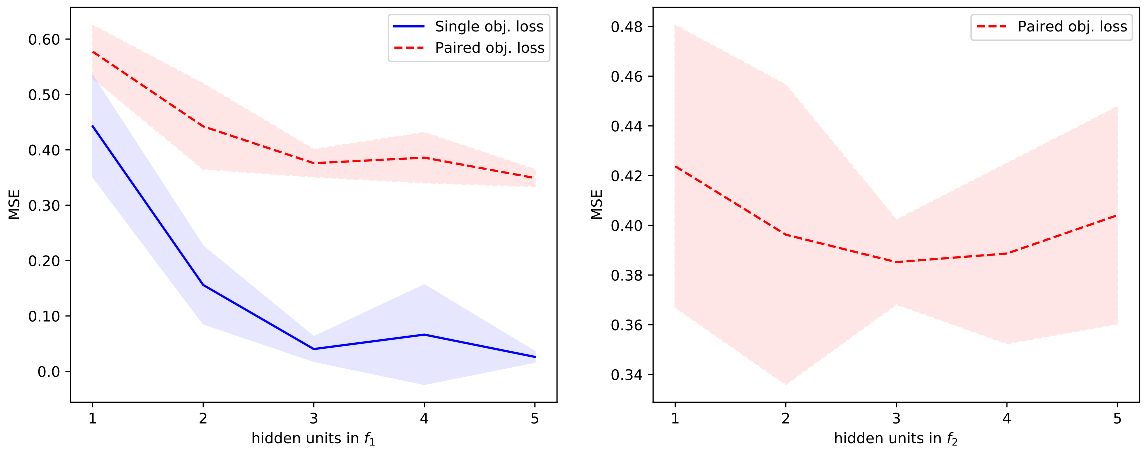
Table A.3. Decoder g_1.

Layer	Input units	Output units
FC+ReLU	5	128
FC+ReLU	128	128
FC	128	3
Number of parameters: 17,667		

Table A.4. Decoder g_2.

Layer	Input units	Output units
FC+ReLU	5	128
FC+ReLU	128	128
FC	128	6
Number of parameters: 18,054		

While finding the number of hidden units, we take five runs and record the MSE. We increase the number of units if the one-sided Welch's t-test rejects the null hypothesis $\mathcal{H}_0$: "Two numbers result in the same MSE" in favor of $\mathcal{H}_1$: "Increased number results in lower MSE".



(a) (b)

Figure A.1. The mean square error losses for (a) f_1-g_1 and (b) f_2-g_2 network pairs. In (a), we also show the paired object MSE with a single unit for a varying number of units in the bottleneck of f_1.

We realize that this requires multiple runs and, in fact, is quite inefficient. It would be better if we had a well-defined metric such as the Bayesian Information Criterion (BIC). We did not use BIC since it does not lead to plausible results with deep neural networks that have large parameter sizes.

A.1.2. MNIST 8-puzzle Environment

Network architectures of the encoder and the decoder for the MNIST 8-puzzle environment are given in Tables A.5 and A.6. For 8-puzzle w/r and 15-puzzle w/r versions, the bottleneck size is changed from 13 to 14 and 15, respectively. To ensure the output size for the 15-puzzle, we change the padding of the third and the fourth convolutional layer in the decoder from one to zero.

Table A.5. The encoder for the 8-puzzle environment.

Layer	In ch.	Out ch.	Stride	Pad
Conv4x4	1	64	2	1
Conv4x4	64	128	2	1
Conv4x4	128	256	2	1
Conv4x4	256	512	2	1
Global average pooling over channels				
FC	512	13	-	-
Gumbel-sigmoid				
Number of parameters: 2,763,085				

A.2. Using the Straight-Through Estimator

The experiment results with STE on the tabletop environment are reported in Table A.7. Here, objects vary in their sizes and initial positions. The mean and the standard deviation of 10 runs are reported.

Table A.6. The decoder for the 8-puzzle environment. ConvT stands for transposed convolutional layers. The last layer of the decoder does not include a batch normalization layer.

Layer	In ch.	Out ch.	Stride	Pad
FC	13+4	512	–	–
Reshape (-1, 512) $\rightarrow$ (-1, 512, 1, 1)				
ConvT5x5	512	256	1	0
ConvT4x4	256	128	2	1
ConvT4x4	128	64	2	1
ConvT4x4	64	32	2	1
ConvT4x4 (no batch norm.)	32	1	2	1
Number of parameters: 3,976,097				

A.3. The Number of States in 8-puzzle w/r and 15-puzzle w/r

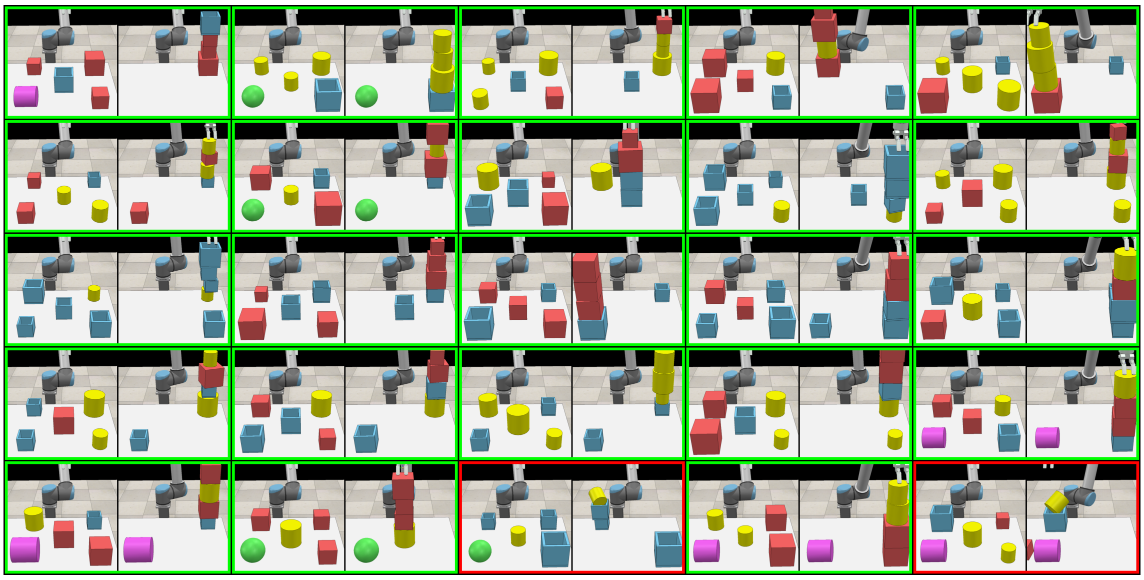
For the 8-puzzle w/r, the number of possible states increases from $9! = 362880$ to $9 \times 9^8 = 387420489$, which is an increase by about a factor of 1000. In general, the number of states is $n^2 k^{(n^2-1)}$ where n stands for the size of the board (the size is three for 8-puzzle and four for 15-puzzle), and k is the number of possible digits other than the empty tile. This translates to $\approx 3.29 \times 10^{15}$ states for 15-puzzle w/r. On the other hand, the number of states that the encoder should represent is $(n-2)^2k^4 + 4(n-2)k^3 + 4k^2$, which translates to 9801 and 32400 states for 8-puzzle w/r and 15-puzzle w/r, respectively. Therefore, we train DeepSym with 14 units for 8-puzzle w/r ($\log_2 9801 \approx 13.26$) and with 15 units for 15-puzzle w/r ($\log_2 32400 \approx 14.98$). In a sense, we use the most strict limit for the number of units.

Table A.7. The relative assignment frequencies of objects to different symbols.

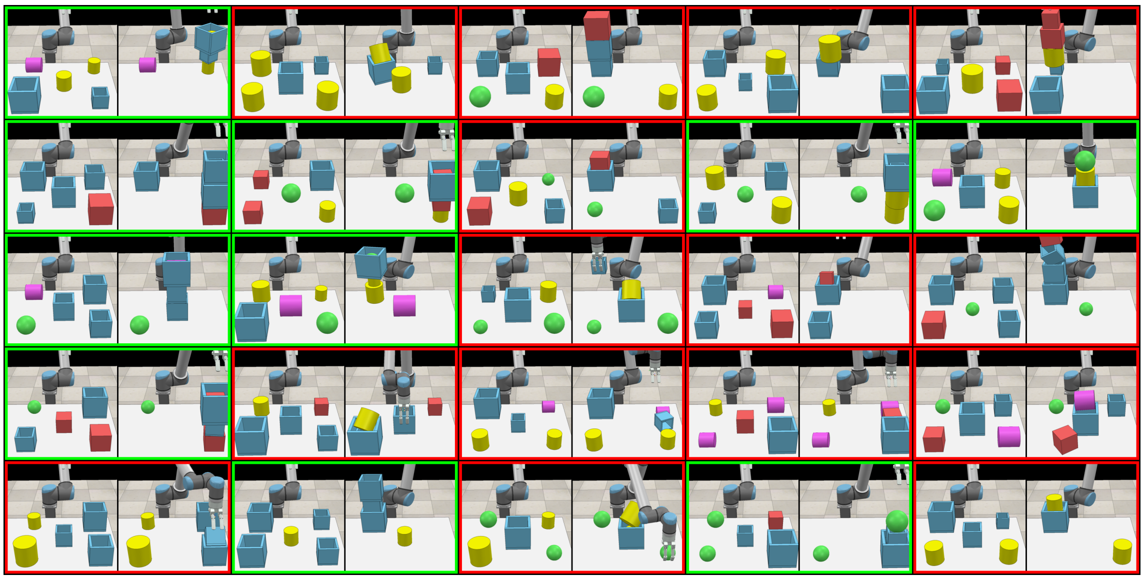
DeepSym with STE				
Category	(0, 0)	(0, 1)	(1, 0)	(1, 1)
Sphere	92.9 ± 8.6	2.5 ± 4.4	3.2 ± 6.8	1.4 ± 2.3
Cube	1.4 ± 3.1	92.9 ± 10.3	3.4 ± 5.6	2.3 ± 3.7
Vertical Cylinder	1.9 ± 4.6	93.6 ± 5.4	1.8 ± 2.7	2.7 ± 3.1
Horizontal Cylinder	15.8 ± 27.3	7.6 ± 10.7	74.7 ± 27.0	2.0 ± 3.8
Cup	0.1 ± 0.2	0.0 ± 0.0	0.0 ± 0.0	99.9 ± 0.2
Autoencoder with STE (OBO)				
Category	(0, 0)	(0, 1)	(1, 0)	(1, 1)
Sphere	85.1 ± 12.1	12.3 ± 9.6	2.6 ± 4.3	0.0 ± 0.0
Cube	74.6 ± 17.7	20.4 ± 12.0	5.0 ± 7.0	0.0 ± 0.0
Vertical Cylinder	76.2 ± 15.6	18.3 ± 9.4	5.5 ± 8.5	0.0 ± 0.0
Horizontal Cylinder	78.1 ± 14.4	19.2 ± 11.1	2.8 ± 3.8	0.0 ± 0.0
Cup	92.4 ± 14.1	6.6 ± 13.5	0.9 ± 2.8	0.1 ± 0.2

A.4. Generated Plans in DeepSym

Generated plans for different goals are shown in Figure A.2. Successes are framed in green and fails are framed in red.



(a)



(b)

Figure A.2. Plan executions. (a) Tower with a height of four using four objects (H4S4). (b) Tower with a height of three using four objects (H3S4).

A.5. Symbols Learned in 8-puzzle w/r and 15-puzzle w/r

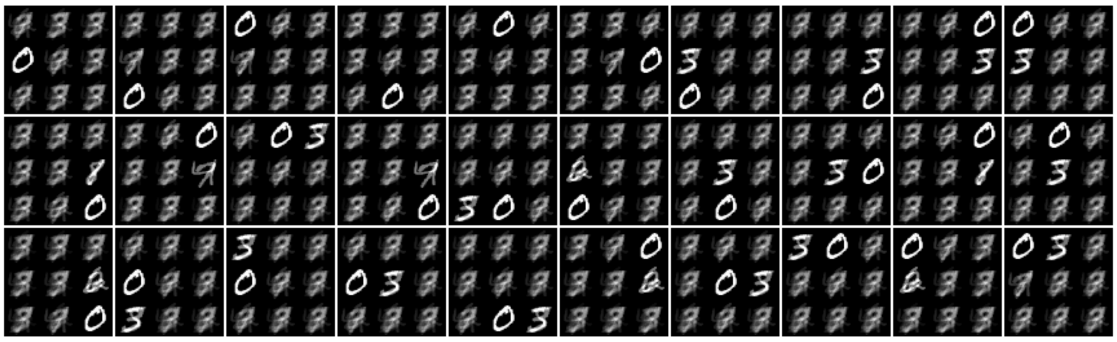


Figure A.3. Average states that correspond to symbols learned in 8-puzzle w/r environment.

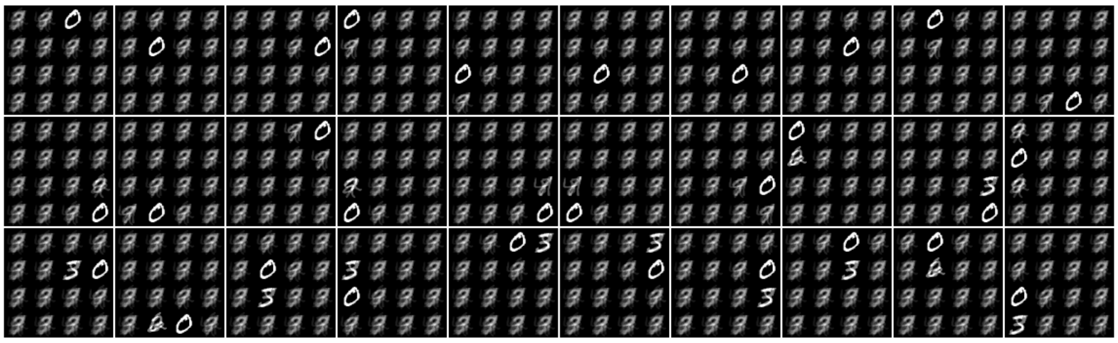


Figure A.4. Average states that correspond to symbols learned in 15-puzzle w/r environment.

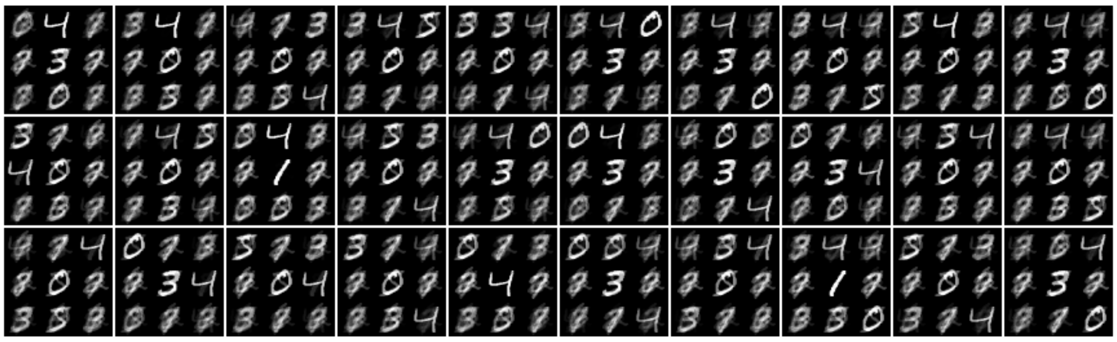


Figure A.5. Average states that correspond to autoencoder symbols learned in the 8-puzzle environment.

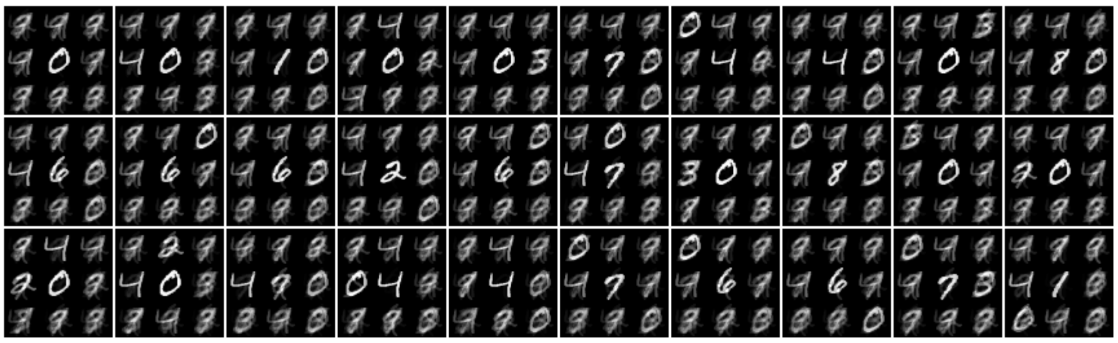


Figure A.6. Average states that correspond to autoencoder symbols learned in 8-puzzle w/r environment.

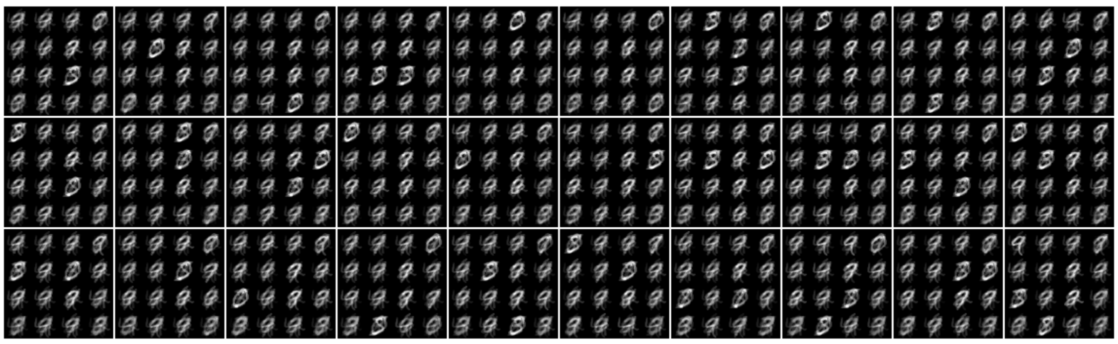


Figure A.7. Average states that correspond to autoencoder symbols learned in 15-puzzle w/r environment.