NEURAL CONSTRAINT SATISFACTION: HIERARCHICAL ABSTRACTION FOR COMBINATORIAL GENERALIZATION IN OBJECT REARRANGEMENT

Michael Chang[*], Alyssa L. Dayan[*], Franziska Meier[†], Thomas L. Griffiths[‡], Sergey Levine[*], Amy Zhang[†,⋆]

ABSTRACT

Object rearrangement is a challenge for embodied agents because solving these tasks requires generalizing across a combinatorially large set of configurations of entities and their locations. Worse, the representations of these entities are unknown and must be inferred from sensory percepts. We present a hierarchical abstraction approach to uncover these underlying entities and achieve combinatorial generalization from unstructured visual inputs. By constructing a factorized transition graph over clusters of entity representations inferred from pixels, we show how to learn a correspondence between intervening on states of entities in the agent's model and acting on objects in the environment. We use this correspondence to develop a method for control that generalizes to different numbers and configurations of objects, which outperforms current offline deep RL methods when evaluated on simulated rearrangement tasks. Project website.

1 INTRODUCTION

The power of an abstraction depends on its usefulness for solving new problems. Object rearrangement (Batra et al., 2020) offers an intuitive setting for studying the problem of learning reusable abstractions. Solving novel rearrangement problems requires an agent to not only infer object representations without supervision, but also recognize that the same action for moving an object between two locations can be reused for different objects in different contexts.

We study the simplest setting in simulation with pick-and-move action primitives that move one object at a time. Even such a simple setting is challenging because the space of object configurations is combinatorially large, resulting in long-horizon combinatorial task spaces. We formulate rearrangement as an offline goal-conditioned reinforcement learning (RL) problem, where the agent is pretrained on a experience buffer of sensorimotor interactions and is evaluated on producing actions for rearranging objects specified in the input image to satisfy constraints depicted in a goal image.

Offline RL methods (Levine et al., 2020) that do not infer factorized representations of entities struggle to generalize to problems with more objects. But planning with object-centric methods that do infer entities (Veerapaneni et al., 2020) is also not easy because the difficulties of long-horizon planning with learned parametric models (Janner et al., 2019) are exacerbated in combinatorial spaces.

Instead of planning with parametric models, our work takes inspiration from non-parametric planning methods that have shown success in combining neural networks with graph search to generate long-horizon plans. These methods (Yang et al., 2020; Zhang et al., 2018; Lippi et al., 2020; Emmons et al., 2020) explicitly construct

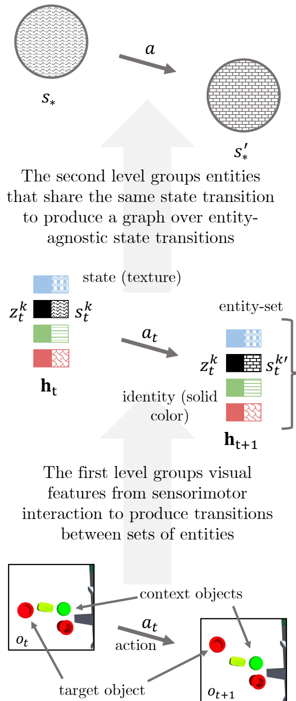


Figure 1: NCS uses a two-level hierarchy to abstract sensorimotor interactions into a graph of learned state transitions. The affected entity is in black.

[*]UC Berkeley. [†]Meta AI Research. [‡]Princeton University. [⋆]UT Austin.
Work done while MC interned at Meta AI. Correspondence to: mbchang@berkeley.edu, amyzhang@meta.com

a transition graph from the experience buffer and plan by searching through the actions recorded in the graph with a learned distance metric. The advantage of such approaches is the ability to stitch different path segments from offline data to solve new problems. The disadvantage is that the non-parametric nature of such methods requires transitions that will be used for solving new problems to have already been recorded in the buffer, making conventional methods, which store entire observations monolithically, ill-suited for combinatorial generalization. Fig. 2b shows that the same state transition can manifest for different objects and in different contexts, but monolithic non-parametric methods are not constrained to recognize that all scenarios exhibit the same state transition at an abstract level. This induces an blowup in the number of nodes in the graph. To overcome this problem, we devise a method that exploits the similarity among state transitions in different contexts.

Our method, **Neural Constraint Satisfaction** (NCS)[1], marries the strengths of non-parametric planning with those of object-centric representations. Our main contribution is to show that factorizing the traditionally monolithic entity representation into action-invariant features (its **type**) and action-dependent features (its **state**) makes it possible during planning and control to reuse action representations for different objects in different contexts, thereby addressing the core combinatorial challenge in object rearrangement. To implement this factorization, NCS constructs a two-level hierarchy (Fig. 1) to abstract the experience buffer into a graph over state transitions of individual entities, separated from other contextual entities (Fig. 3). To solve new rearrangement problems, NCS infers what state transitions can be taken given the current and goal image observations, re-composes sequences of state transitions from the graph, and translates these transitions into actions.

In §3 we introduce a problem formulation that exposes the combinatorial structure of object rearrangement tasks by explicitly modeling the independence, symmetry, and factorization of latent entities. This reveals two challenges in object rearrangement which we call the **correspondence problem** and **combinatorial problem**. In §4 we present NCS, a method for controlling an agent that plans over and acts with emergent learned entity representations, as a unified method for tackling both challenges. We show in §5 that NCS outperforms both state-of-the-art offline RL methods and object-centric shooting-based planning methods in simulated rearrangement problems.

2 Related Work

The problem of discovering re-composable representations is generally motivated by combinatorial task spaces. The traditional approach to enforcing this compositional inductive bias is to compactly represent the task space with MDPs that human-defined abstractions of entities, such as factored MDPs (Boutilier et al., 1995; 2000; Guestrin et al., 2003a), relational MDPs (Wang et al., 2008; Guestrin et al., 2003b; Gardiol & Kaelbling, 2003), and object-oriented MDPs (Diuk et al., 2008; Abel et al., 2015). Approaches building off of such symbolic abstractions (Chang et al., 2016; Battaglia et al., 2018; Zadaianchuk et al., 2022; Bapst et al., 2019; Zhang et al., 2018) do not address the problem of how such entity abstractions arise from raw data. Our work is one of the first to learn compact representations of combinatorial task spaces directly from raw sensorimotor data.

Recent object-centric methods (Greff et al., 2017; Van Steenkiste et al., 2018; Greff et al., 2019; 2020; Locatello et al., 2020a; Kipf et al., 2021; Zoran et al., 2021; Singh et al., 2021) do learn entity representations, as well as their transformations (Goyal et al., 2021; 2020), from sensorimotor data, but only do so for modeling images and video, rather than for taking actions. Instead, we study *how well entity-representations can reused for solving tasks*. Kulkarni et al. (2019) considers how object representations improve exploration, but we consider the offline setting which requires zero-shot generalization. Veerapaneni et al. (2020) also considers control tasks, but their shooting-based planning method suffers from compounding errors as other learned single-step models do (Janner et al., 2019), while our hierarchical non-parametric approach enables us to plan for longer horizons.

Non-parametric approaches have recently become popular for long horizon planning (Yang et al., 2020; Zhang et al., 2018; Lippi et al., 2020; Emmons et al., 2020; Zhang et al., 2021), but the drawback of these approaches is they represent the entire scenes monolithically, which causes a blowup of nodes in combinatorial task spaces, making it infeasible for these methods to be applied in rearrangement tasks that require generalizing to novel object configurations with different numbers of objects. Similar to our work, Huang et al. (2019) also tackles rearrangement problems by searching over a constructed latent task graph, but they require a demonstration during deployment time,

[1] See Appdx. G for an explanation behind this name.

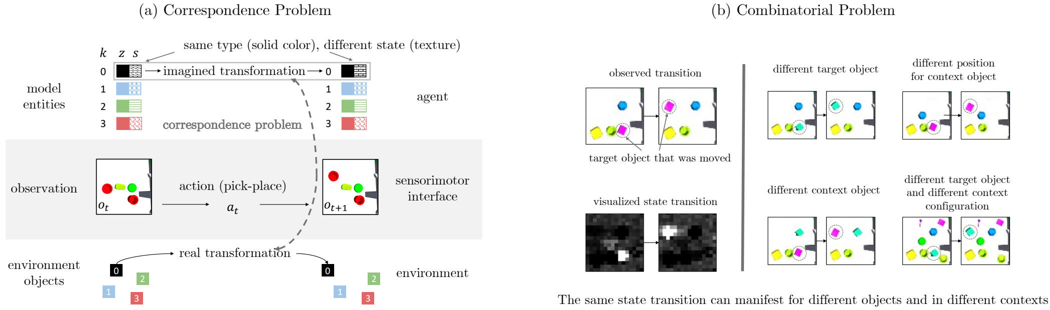


Figure 2: **Solving object rearrangement requires solving two challenges**. (a) The **correspondence problem** is the problem of abstracting raw sensorimotor signal into representations of entities such that there is a correspondence between how an agent intervenes on an entity and how its action affects an object in the environment. k denotes the index of the entity, z denotes its type (shown with solid colors), and s denotes its state (shown with textures). The entity representing the moved object is in black. (b) The **combinatorial problem** is the problem of representing the combinatorial task space in a way that enables an agent to transfer knowledge of a given state transition (indicated by the dotted circle) to different contexts.

whereas NCS does not because it reuses context-agnostic state transitions that were constructed during training. Zhang et al. (2021) conducts non-parametric planning directly on abstract subgoals rather than object-centric states — while similar, the downside of using subgoals rather than abstract states is that those subgoals are not used to represent equivalent states and therefore cannot generalize to new states at test time. Our method, NCS, captures both reachability between known states and new, unseen states that can be mapped to the same abstract state.

3 Goal-Conditioned Reinforcement Learning with Entities

This section introduces a set of modifications to the standard goal-conditioned partially observed Markov decision process (POMDP) problem formulation that explicitly expose the combinatorial structure of object rearrangement tasks of the following kind: "Sequentially move a subset (or all) of the objects depicted in the current observation o_1 to satisfy the constraints depicted in the goal image o_g." We assume an offline RL setting, where the agent is trained on a buffer of transitions $\{(o_1, a_1, ...a_{T-1}, o_T)\}_{n=1}^{N}$ and evaluated on tasks specified as (o_1, o_g).

The standard POMDP problem formulation assumes an observation space $\mathcal{O}$, action space $\mathcal{A}$, latent space $\mathcal{H}$, goal space $\mathcal{G}$, observation function $E : \mathcal{H} \to \mathcal{O}$, transition function $P : \mathcal{H} \times \mathcal{A} \to \mathcal{H}$, and reward function $R : \mathcal{H} \times \mathcal{G} \to \mathbb{R}$. Monolithically modeling the latent space this way does not expose commonalities among different scenes, such as scenes that contain objects in the same location or scenes with multiple instances of the same type of object. This prevents us from designing control algorithms that exploit these commonalities to collapse the combinatorial task space.

To overcome this issue, we introduce structural assumptions of independence, symmetry, and factorization to the standard formulation. The *independence* assumption encodes the intuitive property that objects can be acted upon without affecting other objects. This is implemented by decomposing the latent space into independent subspaces as $\mathcal{H} = \mathcal{H}^1 \times ... \times \mathcal{H}^K$, one for each independent degree of freedom (e.g. object) in the scene. The *symmetry* assumption encodes the property that the the same physical laws apply to all objects. This is implemented by constraining the observation, transition, and reward functions to be shared across all subspaces, thereby treating $\mathcal{H}^1 = ... = \mathcal{H}^K$. We define an **entity**[2] $h \in \mathcal{H}^k$ as a member of such a subspace, and an **entity-set** as the set of entities $\mathbf{h} = (h^1, ..., h^K)$ that explain an observation, similar to Diuk et al. (2008); Weld (2003). Lastly, the *factorization* assumption encodes that each subspace can be decomposed as $\mathcal{H}^k = \mathcal{Z} \times \mathcal{S}$, where $z \in \mathcal{Z}$ represents an entity's action-invariant features like appearance, and $s \in \mathcal{S}$ represents its action-dependent features like location. We call z the **type** and s the **state**.

Introducing these assumptions solves the problem of modeling the commonalities among different scenes stated above. It allows us to describe scenes that contain objects in the same location by assigning entities in different scenes to share the same state s. It allows us to describe a scene with multiple instances of the same type of object by assigning multiple entities in the scene to share the same type z. This formulation also makes it natural to express goals as a set of constraints

[2]We use "object" to refer to an independent degree of freedom in the environment, and "entity" to refer to the agent's representation of the object.

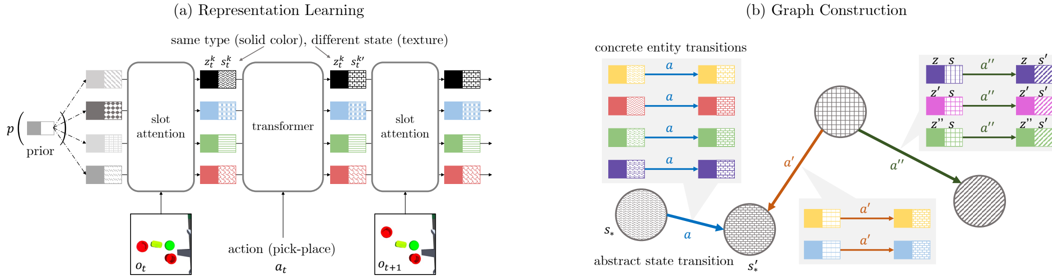


Figure 3: **Modeling**. NCS constructs a two-level abstraction hierarchy to model transitions in the experience buffer. (a) **Level 1:** NCS learns to infer a set of entities from sensorimotor transitions with pick-and-move actions, in which one entity is moved per transition. We enforce that the type z (shown with solid colors) of an entity remains unchanged between time-steps. The GPT dynamics model learns to sparsely predict the states s (shown with textures) of the entities at the next time-step. *This addresses the correspondence problem by forcing the network to use predict and reconstruct observations through the entity bottleneck.* (b) **Level 2:** NCS abstracts transitions over entity-sets into transitions over states of individual entities, constructing a graph where states are nodes and transitions between them are edges. This is done by clustering entity transitions that share similar initial states and final states. *This addresses the combinatorial problem by making it possible for state transitions to reused for different entity types and with different context entities.*

$\mathbf{h}_g = (h_g^1, ..., h_g^k)$. To solve a task is to take actions that transform the subset of entities in the initial observation o_1 whose types are given by $\mathbf{z}_g$ to new states specified by $\mathbf{s}_g$.

Exposing this structure in our problem formulation gives us a language for designing methods that represent entities in an independent, symmetric, and factorized way and that use these three properties to collapse the combinatorial task space. These methods need to solve two problems: the **correspondence problem** of learning to represent entities in this way and the **combinatorial problem** of using these properties to make planning tractable. The correspondence problem is hard because it assumes no human supervision of what the entities are. It also goes beyond problems solved by existing object-centric methods for images and videos because it involves action: it requires representing entities such that there is a correspondence between how the agent models how its actions affect entities and how its actions actually affect objects in the environment. The combinatorial problem goes beyond problems solved by methods for solving object-oriented MDPs, relational MDPs, and factorized MDPs because it requires the agent to recognize whether and how previously observed state transitions can be used for new problems, using learned, rather than human-defined, entity representations. The natural evaluation criterion for both problems is to test to what extent an agent can zero-shot-generalize to solve rearrangement tasks involving new sets of object configurations that aree disjoint from the configurations observed in training, assuming that the training configurations have collectively covered $\mathcal{Z}$ and $\mathcal{S}$. Our experiments in §5 test exactly this.

Simplifying assumptions To focus on the combinatorial nature of rearrangement, we are not interested in low-level manipulation, so we represent each action as $(w, \Delta w)$, where w are Cartesian coordinates $w = (x, y, z)$. We assume actions sparsely affect one entity at a time and how an action affects an object's state does not depend on its identity. We are not interested in handling occlusion, so we assume that objects are constrained to the xy plane or xz plane and are directly visible to the camera. Following prior work (Hansen-Estruch et al., 2022; Castro et al., 2009), we make a *bisimulation* assumption that the state space can be partitioned into a finite set of equivalence classes, and that there is one action primitive that transitions between each pair of equivalence classes. Lastly, we assume objects can be moved independently. Preliminary experiments suggest that NCS can be augmented to support tasks like block-stacking that involve dependencies among objects, but how to handle these dependencies would warrant a standalone treatment in future work.

4 Neural Constraint Satisfaction

In §3 we introduced a structured problem formulation for object rearrangement and reduced it to solving the correspondence and combinatorial problems. We now present our method, Neural Constraint Satisfaction (NCS) as a method for controlling an agent that plans over and acts with a state transition graph constructed from learned entity representations. This section is divided into two parts: modeling and control. The modeling part is further divided into two parts: representation learning and graph construction. The representation learning part addresses the correspondence problem, while the graph construction and control parts address the combinatorial problem.

4.1 MODELING

The modeling component of NCS abstracts the experience buffer into a factorized state transition graph that can be reused across different rearrangement problems. Below we describe how we first train an object-centric world model to infer entities that are independent, symmetric, and factorized and then construct the state transition graph by clustering entities with similar state transitions. These two steps comprise a two-level abstraction hierarchy over the raw sensorimotor transitions.

Level 1: representation learning The first level concerns the unsupervised learning of entity representations that factorizes into their action-invariant features (their **type**) and their action-dependent features (their **state**). Concretely our goal is to model a video transition $o_t, a_t \to o_{t+1}$ as a transition over entity-sets $\mathbf{h}_t, a_t \to \mathbf{h}_{t+1}$, where each entity h^k is factorized as a pair $h^k = (z^k, s^k)$. Given our setting where an action moves only a single object in the environment at a time, successful representation learning implies three criteria: (1) the world model properly identifies the individual entity h^k corresponding to the moved object, (2) only the state s^k of that entity should change, while its type z^k should remain unaffected, and (3) other entity representations $h^{\neq k}$ should also remain unaffected. Criteria (1) and (3) rule out standard approaches that represent an entire scene with a monolithic representation, so we need an object-centric world model instead of a monolithic world model. But criterion (2) rules out standard object-centric world models (e.g. (Veerapaneni et al., 2020; Elsayed et al., 2022; Singh et al., 2022b)), which do not decompose entity representations into action-invariant and action-dependent features.

Because the parameters of a mixture model are independent and symmetric by construction, we propose to construct our factorized object-centric world model as an equivariant sequential Bayesian filter with a mixture model as the latent state, where entity representations are the parameters of the mixture components. Recall that a filter consists of two major components, latent estimation and latent prediction. We implement latent estimation with the state-of-the-art slot attention (SA) (Locatello et al., 2020b), based on the connection between mixture components and SA slots (Chang et al., 2022). We implement latent prediction with the transformer decoder (TFD) architecture (Vaswani et al., 2017) because TFD is equivariant with respect to its inputs. We denote the SA `slots` as $\boldsymbol{\lambda}$ and SA `attn` masks as $\boldsymbol{\alpha}$. We split each `slot` $\lambda \in \mathbb{R}^n$ into two halves $\lambda^z \in \mathbb{R}^{\frac{n}{2}}$ and $\lambda^s \in \mathbb{R}^{\frac{n}{2}}$. Given observations o and actions a, we embed the actions as $\tilde{a}$ with an feedforward network and implement the filter as:

$$\hat{\boldsymbol{\lambda}}_1 \sim \text{Gaussian} \qquad \hat{\boldsymbol{\lambda}^s}_{t+1} = TFD\left(\text{queries} = \boldsymbol{\lambda}^s_t, \text{keys/values} = [\boldsymbol{\lambda}^s, \tilde{a}_t]\right)$$

$$\boldsymbol{\lambda}_t, \boldsymbol{\alpha}_t = SA\left(\hat{\boldsymbol{\lambda}}_t, o_t\right) \qquad \hat{\boldsymbol{\lambda}}_{t+1} = \left[\boldsymbol{\lambda}^z_t, \hat{\boldsymbol{\lambda}^s}_{t+1}\right]$$

where $[\cdot, \cdot]$ is the concatenation operator, $\hat{\boldsymbol{\lambda}}$ is the output of the latent prediction step, and $\boldsymbol{\lambda}$ is the output of the latent estimation step. We embed this filter inside the SLATE backbone (Singh et al., 2022a) and call this implementation **dynamic SLATE** (dSLATE). For a background on SLATE, as well as dSLATE hyperparamters, see Appdx. A.1.

By constructing $\hat{\boldsymbol{\lambda}^z}_{t+1}$ as a copy of $\boldsymbol{\lambda}^z_t$, dSLATE enforces the information contained $\boldsymbol{\lambda}^z$ to be action-invariant, hence we treat $\boldsymbol{\lambda}^z$ as dSLATE's representation of the entities' types. As for the entities' states, either the action-dependent part of the slots $\boldsymbol{\lambda}^s$ or the attention masks α can be used. Using α may be sufficient and more intuitive to analyze if all objects looks similar and there is no occlusion, while $\boldsymbol{\lambda}^s$ may be more suitable in other cases, and we provide an example of each in the experiments. To simplify notation going forward and connect with the notation in §3, we use $\mathbf{h}$ to refer to $(\boldsymbol{\lambda}, \boldsymbol{\alpha})$, use $\boldsymbol{z}$ to refer to $\boldsymbol{\lambda}^z$, and use $\boldsymbol{s}$ to refer to $\boldsymbol{\lambda}^s$ or $\boldsymbol{\alpha}$. Thus by construction dSLATE satisfies criterion (2). Empirically we observe that it satisfies criterion (1) as well as SLATE does, and that TFD learns to sparsely edit $\boldsymbol{\lambda}^s_t$, thereby satisfying criterion (3).

Level 2: graph construction Having produced from the first level a buffer of entity-set transitions $\{\mathbf{h}_t, a_t \to \mathbf{h}_{t+1}\}_{n=1}^N$, the goal of the second level (Fig. 3b) is to use this buffer to construct a factorized state transition graph. The key to solving the combinatorial problem is to construct the edges of this graph to represent not state transitions of entire entity-sets (i.e. $\mathbf{s}_t, a_t \to \mathbf{s}_{t+1}$) as prior work does (Zhang et al., 2018), but state transitions of *individual entities* (i.e. $s^k_t, a_t \to s^k_{t+1}$). Constructing edges over transitions for individual entities rather than entity sets enables the same transition to be reused with different context entities present. Constructing edges over state transitions instead of entity transitions enables the same transition to be reused across entities with different types. This would enable the agent to recompose sequences of previously encountered state transitions for solv-

ing new rearrangement problems with different entities in different contexts. Henceforth our use of "state" refers specifically to the state of individual entity unless otherwise stated.

Given our bisimulation assumption that states can be partitioned into a finite number of groups, we construct our graph such that nodes represent equivalence classes among individual states and the edges represent actions that transform a state from one equivalence class to another. To implement this we cluster state transitions of individual entities in the buffer, which reduces to clustering the states of individual entities before and after the transition. We treat each cluster centroid as a node in the graph, and an edge between nodes is tagged with the single action that transforms one node's state to another's. The algorithm for constructing the graph is shown in Alg. 1 and involves three steps: (1) isolating the state transition of an individual entity from the state transition of the entity-set, (2) creating graph nodes from state clusters, and (3) tagging graph edges with actions.

The first step is to identify which object was moved in each transition, i.e. identifying the entity h^k that dSLATE predicted was affected by a_t in the transition $(\mathbf{h}_t, a_t, \mathbf{h}_{t+1})$. We implement a function `isolate` that achieves this by solving $k = \arg\max_{k' \in \{1,\dots,K\}} d(s_t^{k'}, s_{t+1}^{k'})$ to identify the index of the entity whose state has most changed during the transition, where $d(\cdot,\cdot)$ is a distance function, detailed in Table 3 of the Appendix. This converts the buffer of transitions over entity-sets $\mathbf{h}_t, a_t \rightarrow \mathbf{h}_{t+1}$ into a buffer of transitions over individual entities $h_t^k, a_t \rightarrow h_{t+1}^k$.

Algorithm 1 Building the Graph

```
1: input model, buffer
2: for {(o_t, a_t, o_{t+1})}_n in buffer do
3:     # infer entities from transition
4:     {(h_t, a_t, h_{t+1})}_n ← model({o_t, a_t, o_{t+1}}_n).
5:     # identify which entity changed in transition
6:     {(h_t^k, a_t, h_{t+1}^k)}_n ← isolate({(h_t, a_t, h_{t+1})}_n)
7: end for
8: # partition transitions by clustering entities
9: {s_*}_{m=1}^M ← cluster({(s_t^k, a_t, s_{t+1}^k)}_{n=1}^N)
10: # transitions between clusters are edges
11: initialize graph with nodes s_*^[m], for m ∈ [1 : M]
12: for each {(h_t^k, a_t, h_{t+1}^k)}_n do
13:     # infer cluster assignments
14:     [i], [j] ← bind(h_t^k), bind(h_{t+1}^k)
15:     # tag edge with action a_t
16:     graph.edges[i, j] ← create-edge(s_*^[i] --a_t--> s_*^[j])
17: end for
18: return graph
```

The second step is to cluster the states before and after each transition. We implement a function `cluster` that uses K-means to returns graph nodes as the centroids $\{s_*\}_{m=1}^M$ of these state clusters.

The third step is to connect the nodes with edges that record actions that actually were taken in the buffer to transform one state to the next. We implement a function `bind` that, given entity h^k, returns the index $[i]$ of the centroid s_* that is the nearest neighbor to the entity's state s^k. For each entity transition (h_t^k, a_t, h_{t+1}^k) we `bind` entity h_t^k and h_{t+1}^k to their associated nodes $s_*^{[i]}$ and $s_*^{[j]}$ and create an edge between $s_*^{[i]}$ and $s_*^{[j]}$ tagged with action a, overwriting previous edges based on the assumption that with a proper clustering there should only be one action per pair of nodes.

In our experiments both `cluster` and `bind` use the same distance metric (see Table 2 in the Appendix), but other clustering algorithms and distance metrics can also be used. Our experiments (Fig. 11) also show that it is also possible to have more than one action primitive per pair of nodes as long as these actions all map between states bound to the same pair of nodes.

4.2 CONTROL

To solve new rearrangement problems, we re-compose sequences of state transitions from the graph. Specifically, the agent decomposes the rearrangement problem into a set of per-entity subproblems (e.g. initial and goal positions for individual objects), searches the transition graph for a transition that transforms the current entity's state to its goal state, and executes the action tagged with this transition in the environment. This problem decomposition is possible because the transitions in our graph are constructed to be agnostic to type and context, enabling different rearrangement problems to share solutions to the same subproblems. The core challenge in deciding which transitions to compose is in determining which transitions are *possible* to compose. That is, the agent must determine which nodes in the graph correspond to the given goal constraints and which nodes correspond to the entities in the current observation, but the current entities $\mathbf{h}_t$ and goal constraints $\mathbf{h}_g$ must themselves be inferred from the current and goal observations o_t and o_g, requiring the agent to infer both what to do and how to do it purely from its sensorimotor interface.

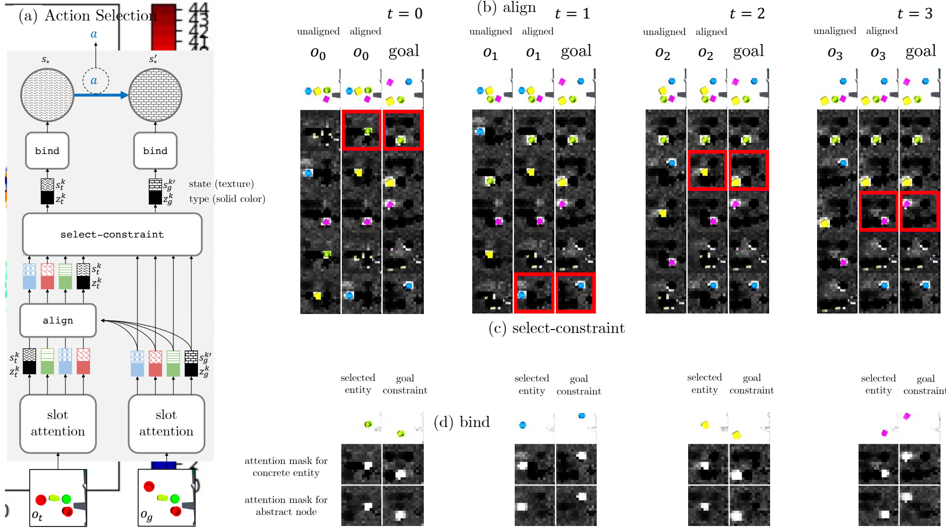


Figure 4: **Planning and control**. Given a rearrangement problem specified only by the current and goal observations (o_0, o_g), NCS decomposes the rearrangement problem into one subproblem (o_t, o_g) per entity. (a) shows the computations NCS uses to solve each subproblem and (b-d) show these steps in context. For each subproblem (o_t, o_g), NCS infers entities from both the current and goal observations. The states of the goal entities indicate constraints on the desired locations of the current entities. (b) NCS aligns the indices of the current entities to those of the goal entities with corresponding types. (c) It selects the index k of the next goal constraint s_g^k to satisfy, as indicated by the red box. The selected goal constraint and current entity are also colored black in (a), and note that their types are the same but states are different: we want to choose the action to transform the state of the current entity to the state of the goal constraint. (d) It binds the selected goal constraint and its corresponding current entity to nodes s_* and s'_* in the transition graph. Lastly, it identifies the edge connecting those two nodes and executes the action tagged to that edge in the environment.

Our approach takes four steps, summarized in Alg. 2 and Fig. 4, with further details in Appdx. A.3. In the first step, we use dSLATE to infer $\mathbf{h}_t$ and $\mathbf{h}_g$ from o_t and o_g (e.g. the positions and types of all objects in the initial and goal images). In the second step (Fig. 4b), because of the permutation symmetry among entities, we find a bipartite matching that matches each entities in h_g^j with a corresponding entity in h_t^k that shares the same type and permute the indices k of $\mathbf{h}_t$ to match those of $\mathbf{h}_g$. We implement a function `align` that uses the Hungarian algorithm to perform this matching over $(z_t^1, ...z_t^K)$ and $(z_g^1, ...z_g^K)$, with Euclidean distance as the matching cost. The third step selects which goal constraint h_g^k to satisfy next (Fig. 4c). W implement this `select-constraint` procedure by determining which constraint h_g^k has the highest difference in state with its counterpart h_t^k, which reduces to solving the same argmax problem as in `isolate` with the same distance function used in `isolate`. The last step chooses an action given the chosen goal constraint h_g^k and its counterpart h_t^k, by `binding` h_t^k and h_g^k to the graph based on their state components and returning the action tagged to the edge between their respective nodes (Fig. 4d). If an edge does not exist between the inferred nodes, then we simply take a random action.

Algorithm 2 Action Selection

1: **given** `model`, `graph`
2: **input** goal o_g, observation o_t
3: # infer goal constraints and current entities
4: $\mathbf{h_g}, \mathbf{h_t} \leftarrow$ `model` (o_g) , `model` (o_t)
5: align entity indices of $\mathbf{h_t}$ with those of $\mathbf{h_g}$
6: $\pi \leftarrow$ `align` $(\mathbf{h_t}, \mathbf{h_g})$
7: permute indices of $\mathbf{h_t}$ according to π
8: $\mathbf{h_t} \leftarrow (h_t^{\pi[1]}, ..., h_t^{\pi[K]})$
9: identify kth goal constraint to satisfy next
10: $k \leftarrow$ `select-constraint` $(\mathbf{h_t}, \mathbf{h_g})$
11: infer cluster assignments
12: $[i], [j] \leftarrow$ `bind` $\left(h_t^k\right)$, `bind` $\left(h_g^k\right)$
13: action that transforms node $[i]$ to node $[j]$
14: **return** `graph.edges`$[i, j]$`.action`

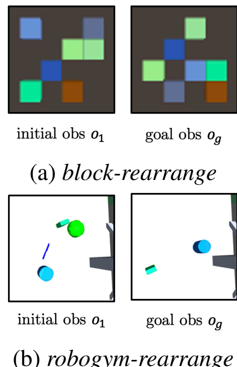
5 Experiments

We have proposed NCS as a solution to the object rearrangement problem that addresses two challenges: NCS addresses the correspondence problem by learning a factorized object-centric world

model with dSLATE and it addresses the combinatorial problem by abstracting entity representations into a queryable state transition graph. Now we test NCS's efficacy in solving both problems.

The key question is whether NCS is better than state-of-the-art offline RL algorithms in generalizing over combinatorially-structured task spaces from perceptual input. As stated in §3, the crucial test for answering this question is to evaluate all methods on solving new rearrangement problems with a disjoint set of object configurations from those encountered during training. The most straightforward way to find a disjoint subset of the combinatorial space is to evaluate with a novel number of objects. We compare NCS to several offline RL baselines and ablations on two rearrangement environments and find a significant gap in performance between our method and the next best method.

Environments. In *block-rearrange* (Fig. 5a), all objects are the same size, shape, and orientation. $\mathcal{S}$ covers 16 locations in a grid. $\mathcal{Z}$ is the continuous space of red-green-blue values from 0 to 1. *robogym-rearrange* (Fig. 5b) is adapted from the OpenAI (2020) rearrange environment and removes the assumptions from *block-rearrange* that all objects have the same size, shape, and orientation. The objects are uniformly sampled from a set of 94 meshes consisting of the YCB object set (Calli et al., 2015) and a set of basic geometric shapes, with colors sampled from a set of 13. Although locations are not pre-defined in *robogym-rearrange* as in *block-rearrange*, in practice there is a limit to the number of ways to arrange objects on the table to still be visible to the camera, which makes the bisimulation still a reasonable assumption here. For *block-rearrange* we use the SA attention mask α as the state s, and for *robogym-rearrange* we use the action-dependent part of the SA slot $\boldsymbol{\lambda}^s$ as the state s. Further environment details are in Appdx. B.



(a) *block-rearrange*

(b) *robogym-rearrange*

Figure 5: Our environments are *block-rearrange* and *robogym-rearrange*. Fig. 5a shows a complete specification of goal constraints; Fig. 5b shows a partial specification that only specifies the desired locations for two objects.

Experimental setup. We evaluate two settings: *complete* and *partial*. In the *complete* setting, the goal image shows all objects in new locations. The *partial* setting is underspecified: only a subset of objects have associated goal constraints (Fig. 5b). In *block-rearrange*, all constraints are unsatisfied in the start state. In *robogym-rearrange*, four constraints are unsatisfied in the start state. Our metric is the *fractional success rate*, the average change in the number of satisfied constraints divided by the number of initially unsatisfied constraints.

The experiences buffer consists of 5000 trajectories showing 4 objects. We evaluate on 4-7 objects for 100 episodes across 10 seeds. Even if we assume full access to the underlying state space, the task spaces are enormous: with $|S|$ object locations and k objects, the number of possible trajectories over object configurations of t timesteps is $\binom{|S|}{k} \times (k \times (|S|-k))^t$, which amounts to searching over more than 10^{16} possible trajectories for the complete specification setting of *block-rearrange* with $k = 7$ objects (see Appdx. E for derivation). Our setting of assuming access to only pixels makes the problem even harder.

Baselines. The claim of this paper are that, for object rearrangement, (1) object-centric methods fare better than monolithically-structured offline RL methods (2) non-parametric graph search fares better than parametric planning for object rearrangement and (3) a factorized graph search over state transitions of individual entities fares better than a non-factorized graph search over state transitions over entire entity-sets. To test (1), we compare with state-of-the-art pixel-based behavior cloning (BC) and implicit Q-learning (IQL) implementations based off of Kostrikov (2021). To test (2), we compare against a version of object-centric model predictive control (MPC) (Veerapaneni et al., 2020) that uses the cross entropy method over dSLATE rollouts. To test (3), we compare against an ablation (abbrv. NF, for "non-factorized") that constructs a graph with state transitions of entity-sets than of individual states. Our last baseline just takes random actions (Rand). Baseline implementation details are in Appdx. C.

5.1 Results

Figure 1 shows that NCS performs significantly better than all baselines (about a 5-10x improvement), thereby refuting the alternatives to our claims in our experimental context. Most of the baselines perform no better or only slightly better than random. We observe that it is indeed difficult to perform shooting-based planning with an entity-centric world model trained to predict a single step forward (Janner et al., 2019): the MPC baseline performs poorly because its rollouts are

poor, and it is significantly more computationally expensive to run (11 hours instead of 20 minutes). We also observe that the NF ablation performs poorly, showing the importance of factorizing the non-parametric graph search. Additional results are in Appdx. D, with limitations in Appdx. F

Table 1: This table compares NCS with various baselines in the complete and partial evaluation settings of *block-rearrange* and *robogym-rearrange*. The methods were trained on 4 objects and evaluated on generalizing to 4, 5, 6, and 7 objects. We report the fractional success rate, with a standard error computed over 10 seeds.

(a) *block-rearrange*, complete specification.

Method	4	5	6	7
NCS (ours)	**0.94** ± 0.01	**0.93** ± 0.00	**0.93** ± 0.00	**0.89** ± 0.00
Rand	0.06 ± 0.02	0.07 ± 0.03	0.07 ± 0.03	0.08 ± 0.03
MPC	0.16 ± 0.06	0.12 ± 0.04	0.11 ± 0.04	0.10 ± 0.03
NF	0.07 ± 0.03	0.06 ± 0.02	0.07 ± 0.02	0.08 ± 0.03
IQL	0.07 ± 0.01	0.03 ± 0.00	0.02 ± 0.00	0.02 ± 0.00
BC	0.03 ± 0.00	0.02 ± 0.00	0.01 ± 0.00	0.01 ± 0.00

(b) *block-rearrange*, complete specification.

Method	4	5	6	7
NCS (ours)	**0.89** ± 0.01	**0.86** ± 0.01	**0.78** ± 0.01	**0.70** ± 0.01
Rand	0.06 ± 0.02	0.08 ± 0.03	0.08 ± 0.03	0.08 ± 0.03
MPC	0.13 ± 0.05	0.11 ± 0.04	0.10 ± 0.04	0.08 ± 0.03
NF	0.06 ± 0.03	0.07 ± 0.03	0.08 ± 0.03	0.07 ± 0.03
IQL	0.01 ± 0.01	0.07 ± 0.01	0.05 ± 0.01	0.05 ± 0.00
BC	0.05 ± 0.01	0.04 ± 0.00	0.03 ± 0.00	0.03 ± 0.00

(c) *robogym-rearrange*, complete specification.

Method	4	5	6	7
NCS (ours)	**0.64** ± 0.01	**0.47** ± 0.01	**0.49** ± 0.01	**0.41** ± 0.01
Rand	0.01 ± 0.00	0.01 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
MPC	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
NF	0.01 ± 0.00	0.01 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
IQL	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
BC	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00

(d) *robogym-rearrange*, partial specification.

Method	4	5	6	7
NCS (ours)	**0.47** ± 0.01	**0.33** ± 0.01	**0.27** ± 0.01	**0.22** ± 0.01
Rand	0.005 ± 0.001	0.001 ± 0.00	0.002 ± 0.001	0.001 ± 0.00
MPC	0.00 ± 0.00	0.001 ± 0.001	0.00 ± 0.00	0.00 ± 0.00
NF	0.005 ± 0.001	0.001 ± 0.00	0.002 ± 0.001	0.001 ± 0.00
IQL	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
BC	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00

5.2 Analysis

Having quantitatively shown the relative strength of NCS in combinatorial generalization from pixels, we now examine how our key design choices of (1) factorizing entity representations into action-invariant and action-dependent features and (2) querying a state transition graph constructed from action-dependent features contribute to NCS's behavior and performance. Is copying the entity type during latent prediction as dSLATE does sufficient for disentangling the location and appearance of objects into the state and type respectively? Does dSLATE learn to sparsely modify only the entity that corresponds to the moved object in the sensorimotor transition, such that the nodes of the state transition graph meaningfully can be reused across entities? These are nontrivial capabilities because NCS is self-supervised on only the experience buffer.

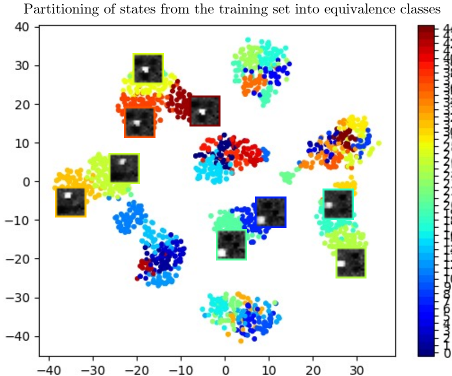


Figure 6: **Nodes as equivalent classes over states.** We show a clustering of states inferred for *robogym-rearrange*, where each cluster centroid is treated as a node in our transition graph. We label some clusters with an attention mask computed by averaging the slot attention masks for the entities associated with the cluster.

Fig. 4b, which visualizes the `align`, `select-constraint`, and `bind` functions of NCS on *robogym-rearrange*, suggests that, at least for the simplified setting we consider, the answer to both questions is yes. NCS has learned to represent different objects in different slots and construct a graph whose nodes capture location information. Fig. 6 shows a t-SNE (Van der Maaten & Hinton, 2008) plot that clusters entities inferred from the *robogym* environment. Because we have not provided supervision on what states should represent, we observe there are multiple cluster indices that map onto similar groups of points. This reveals that multiple different regions of $\mathcal{S}$ appear to be modeling similar states. We also tried merging redundant clusters, but found that this did not improve quantitative performance.

6 Discussion

Object rearrangement offers an intuitive setting for studying how an agent can learn reusable abstractions from its sensorimotor experience. This paper takes a first step toward connecting the world of symbolic planning with human-defined abstractions and the world of representation learning with

deep networks by introducing NCS. NCS is a method for controlling an agent that plans over and acts with state transition graph constructed with entity representations learned from raw sensorimotor transitions, without any other supervision. We showed that factorizing the entity representation into action-invariant and action-dependent features are important for solving the correspondence and combinatorial problems that make the object rearrangement difficult, and enable NCS to significantly outperform existing methods on combinatorial generalization in object rearrangement. The implementation of NCS provides a proof-of-concept for how learning reusable abstractions might be done, which we hope inspires future work to engineer methods like NCS for real-world settings.

ACKNOWLEDGMENTS

This work was done while MC was an intern at Meta AI. We would like to thank Leslie Kaelbling for valuable feedback and Yash Sharma and Yilun Du for valuable discussions. This material is supported in part by the Fannie and John Hertz Foundation, as well as with ONR grant #N00014-18-1-2873.

REFERENCES

David Abel, D. Ellis Hershkowitz, Gabriel Barth-Maron, Stephen Brawner, Kevin O'Farrell, James MacGlashan, and Stefanie Tellex. Goal-based action priors. In *ICAPS*, pp. 306–314. AAAI Press, 2015.

Victor Bapst, Alvaro Sanchez-Gonzalez, Carl Doersch, Kimberly Stachenfeld, Pushmeet Kohli, Peter Battaglia, and Jessica Hamrick. Structured agents for physical construction. In *International conference on machine learning*, pp. 464–474. PMLR, 2019.

Dhruv Batra, Angel X Chang, Sonia Chernova, Andrew J Davison, Jia Deng, Vladlen Koltun, Sergey Levine, Jitendra Malik, Igor Mordatch, Roozbeh Mottaghi, et al. Rearrangement: A challenge for embodied ai. *arXiv preprint arXiv:2011.01975*, 2020.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pp. 1104–1111, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artif. Intell.*, 121(1-2):49–107, 2000.

Berk Calli, Arjun Singh, Aaron Walsman, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. The ycb object and model set: Towards common benchmarks for manipulation research. In *2015 international conference on advanced robotics (ICAR)*, pp. 510–517. IEEE, 2015.

Pablo Samuel Castro, Prakash Panangaden, and Doina Precup. Equivalence relations in fully and partially observable markov decision processes. In *Twenty-First International Joint Conference on Artificial Intelligence*. Citeseer, 2009.

Chien-Yi Chang, De-An Huang, Danfei Xu, Ehsan Adeli, Li Fei-Fei, and Juan Carlos Niebles. Procedure planning in instructional videos. In *European Conference on Computer Vision*, pp. 334–350. Springer, 2020.

Michael Chang, Thomas L Griffiths, and Sergey Levine. Object representations as fixed points: Training iterative refinement algorithms with implicit differentiation. *arXiv preprint arXiv:2207.00787*, 2022.

Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. A compositional object-based approach to learning physical dynamics. *arXiv preprint arXiv:1612.00341*, 2016.

Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. 2016.

Coline Devin, Payam Rowghanian, Chris Vigorito, Will Richards, and Khashayar Rohanimanesh. Self-supervised goal-conditioned pick and place. *arXiv preprint arXiv:2008.11466*, 2020.

Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In *ICML*, volume 307 of *ACM International Conference Proceeding Series*, pp. 240–247. ACM, 2008.

Gamaleldin F Elsayed, Aravindh Mahendran, Sjoerd van Steenkiste, Klaus Greff, Michael C Mozer, and Thomas Kipf. Savi++: Towards end-to-end object-centric learning from real-world videos. *arXiv preprint arXiv:2206.07764*, 2022.

Scott Emmons, Ajay Jain, Misha Laskin, Thanard Kurutach, Pieter Abbeel, and Deepak Pathak. Sparse graphical memory for robust planning. *Advances in Neural Information Processing Systems*, 33:5251–5262, 2020.

Ben Eysenbach, Russ R Salakhutdinov, and Sergey Levine. Search on the Replay Buffer: Bridging Planning and Reinforcement Learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/5c48ff18e0a47baaf81d8b8ea51eec92-Paper.pdf`.

Natalia Gardiol and Leslie Kaelbling. Envelope-based Planning in Relational MDPs. In S. Thrun, L. Saul, and B. Schölkopf (eds.), *Advances in Neural Information Processing Systems*, volume 16. MIT Press, 2003. URL `https://proceedings.neurips.cc/paper/2003/file/4a06d868d044c50af0cf9bc82d2fc19f-Paper.pdf`.

Tejas Gokhale, Shailaja Sampat, Zhiyuan Fang, Yezhou Yang, and Chitta Baral. Cooking with blocks: A recipe for visual reasoning on image-pairs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 5–8, 2019.

Anirudh Goyal, Alex Lamb, Phanideep Gampa, Philippe Beaudoin, Sergey Levine, Charles Blundell, Yoshua Bengio, and Michael Mozer. Object files and schemata: Factorizing declarative and procedural knowledge in dynamical systems. *arXiv preprint arXiv:2006.16225*, 2020.

Anirudh Goyal, Aniket Didolkar, Nan Rosemary Ke, Charles Blundell, Philippe Beaudoin, Nicolas Heess, Michael C Mozer, and Yoshua Bengio. Neural production systems. *Advances in Neural Information Processing Systems*, 34:25673–25687, 2021.

Klaus Greff, Sjoerd Van Steenkiste, and Jürgen Schmidhuber. Neural expectation maximization. *arXiv preprint arXiv:1708.03498*, 2017.

Klaus Greff, Raphaël Lopez Kaufman, Rishabh Kabra, Nick Watters, Christopher Burgess, Daniel Zoran, Loic Matthey, Matthew Botvinick, and Alexander Lerchner. Multi-object representation learning with iterative variational inference. In *International Conference on Machine Learning*, pp. 2424–2433. PMLR, 2019.

Klaus Greff, Sjoerd van Steenkiste, and Jürgen Schmidhuber. On the binding problem in artificial neural networks. *arXiv preprint arXiv:2012.05208*, 2020.

C. Guestrin, D. Koller, R. Parr, and S. Venkataraman. Efficient Solution Algorithms for Factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468, October 2003a. ISSN 1076-9757. doi: 10.1613/jair.1000. URL `https://jair.org/index.php/jair/article/view/10341`. tex.ids: guestrin2003EfficientSolutionAlgorithms.

Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing plans to new environments in relational mdps. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pp. 1003–1010, San Francisco, CA, USA, 2003b. Morgan Kaufmann Publishers Inc.

Philippe Hansen-Estruch, Amy Zhang, Ashvin Nair, Patrick Yin, and Sergey Levine. Bisimulation makes analogies in goal-conditioned reinforcement learning. *arXiv preprint arXiv:2204.13060*, 2022.

De-An Huang, Suraj Nair, Danfei Xu, Yuke Zhu, Animesh Garg, Li Fei-Fei, Silvio Savarese, and Juan Carlos Niebles. Neural task graphs: Generalizing to unseen tasks from a single video demonstration. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 8565–8574, 2019.

Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

Thomas Kipf, Gamaleldin F Elsayed, Aravindh Mahendran, Austin Stone, Sara Sabour, Georg Heigold, Rico Jonschkowski, Alexey Dosovitskiy, and Klaus Greff. Conditional object-centric learning from video. *arXiv preprint arXiv:2111.12594*, 2021.

Ilya Kostrikov. JAXRL: Implementations of Reinforcement Learning algorithms in JAX, 10 2021. URL https://github.com/ikostrikov/jaxrl.

Tejas D Kulkarni, Ankush Gupta, Catalin Ionescu, Sebastian Borgeaud, Malcolm Reynolds, Andrew Zisserman, and Volodymyr Mnih. Unsupervised learning of object keypoints for perception and control. *NeurIPS*, 2019.

Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

Martina Lippi, Petra Poklukar, Michael C Welle, Anastasiia Varava, Hang Yin, Alessandro Marino, and Danica Kragic. Latent space roadmap for visual action planning of deformable and rigid object manipulation. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5619–5626. IEEE, 2020.

Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention. *arXiv preprint arXiv:2006.15055*, 2020a.

Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention. In *NeurIPS*, 2020b. URL https://proceedings.neurips.cc/paper/2020/hash/8511df98c02ab60aea1b2356c013bc0f-Abstract.html.

OpenAI. Robogym. https://github.com/openai/robogym, 2020.

Luis Pineda, Brandon Amos, Amy Zhang, Nathan O. Lambert, and Roberto Calandra. Mbrl-lib: A modular library for model-based reinforcement learning. *Arxiv*, 2021. URL https://arxiv.org/abs/2104.10159.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021.

Gautam Singh, Fei Deng, and Sungjin Ahn. Illiterate DALL·E learns to compose. *arXiv preprint arXiv:2110.11405*, 2021.

Gautam Singh, Fei Deng, and Sungjin Ahn. Illiterate DALL-e learns to compose. In *International Conference on Learning Representations*, 2022a. URL https://openreview.net/forum?id=h0OYV0We3oh.

Gautam Singh, Yi-Fu Wu, and Sungjin Ahn. Simple unsupervised object-centric learning for complex and naturalistic videos. *arXiv preprint arXiv:2205.14065*, 2022b.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pp. 5026–5033. IEEE, 2012.

Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

Sjoerd Van Steenkiste, Michael Chang, Klaus Greff, and Jürgen Schmidhuber. Relational neural expectation maximization: Unsupervised discovery of objects and their interactions. *arXiv preprint arXiv:1802.10353*, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Rishi Veerapaneni, John D Co-Reyes, Michael Chang, Michael Janner, Chelsea Finn, Jiajun Wu, Joshua Tenenbaum, and Sergey Levine. Entity abstraction in visual model-based reinforcement learning. In *Conference on Robot Learning*, pp. 1439–1456. PMLR, 2020.

C. Wang, S. Joshi, and R. Khardon. First order decision diagrams for relational mdps. *Journal of Artificial Intelligence Research*, 31:431–472, Mar 2008. ISSN 1076-9757. doi: 10.1613/jair.2489. URL http://dx.doi.org/10.1613/jair.2489.

Daniel S Weld. Solving Relational MDPs with First-Order Machine Learning. pp. 8, 2003.

Ge Yang, Amy Zhang, Ari S. Morcos, Joelle Pineau, Pieter Abbeel, and Roberto Calandra. Plan2vec: Unsupervised Representation Learning by Latent Plans. In *Proceedings of The 2nd Annual Conference on Learning for Dynamics and Control*, volume 120 of *Proceedings of Machine Learning Research*, pp. 1–12, 2020.

Shuo Yang, Wei Zhang, Ran Song, Jiyu Cheng, and Yibin Li. Learning multi-object dense descriptor for autonomous goal-conditioned grasping. *IEEE Robotics and Automation Letters*, 6(2):4109–4116, 2021.

Andrii Zadaianchuk, Georg Martius, and Fanny Yang. Self-supervised reinforcement learning with independently controllable subgoals. In *Conference on Robot Learning*, pp. 384–394. PMLR, 2022.

Amy Zhang, Adam Lerer, Sainbayar Sukhbaatar, Rob Fergus, and Arthur Szlam. Composable Planning with Attributes. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80, pp. 5837–5846. JMLR.org, 2018.

Lunjun Zhang, Ge Yang, and Bradly C Stadie. World model as a graph: Learning latent landmarks for planning. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 12611–12620. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/zhang21x.html.

Daniel Zoran, Rishabh Kabra, Alexander Lerchner, and Danilo J Rezende. Parts: Unsupervised segmentation with slots, attention and independence maximization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 10439–10447, 2021.

A IMPLEMENTATION DETAILS

This section details the implementation design decisions for each component of NCS. The hyperparameters of dSLATE are given in Tab. 2.

A.1 BACKGROUND: SLATE BACKBONE

SLATE (Singh et al., 2022a) is an autoencoder architecture that uses slot attention (SA) (Locatello et al., 2020b) as a bottleneck. It preprocesses the image with a discrete variational autoencoder (Ramesh et al., 2021) into a grid of image features, encodes these features into a grid of tokens, infers slots from this token grid with SA, which also produces an attention mask over the features each slot attends to. These slots are trained using a transformer decoder (Vaswani et al., 2017; Radford et al., 2018) to autoregressively reconstruct the tokens using the slots as keys/values.

Number of epochs		200
Episodes per epoch		5K
Episode length		5
Batch size		32
Peak LR		0.0002
LR warmup steps		30000
Dropout		0.1
Discrete VAE	Vocabulary Size	4096
	Temp. Cooldown	1.0 to 0.1
	Temp. Cooldown Steps	30000
	LR (no warmup)	0.0003
	Image Size	64
	Image Tokens	Image Size / 4
transformer decoder	Layers	4
	Heads	4
	Hidden Dim.	192
Slot attention	Slots	5
	Iterations	3
	Slot Heads	1
	Slot Dim. (h)	192
	Type Dim. (λ^z)	96
	State Dim. (λ^s)	96
transformer dynamics	Layers	4
	Heads	4
	Hidden Dim.	96

Table 2: **Hyperparameters for training dSLATE** These hyperparameters are almost identical to those found in Singh et al. (2022a, Fig. 7), but because dSLATE operates on video demonstrations rather than static images, we changed some hyperparameters to save memory cost. We changed the batch size from 50 to 32, the number of transformer layers and heads from 8 to 4, the number of slot attention iterations from 7 to 3 without observing a significant change in performance. Because each video in the experience buffer contains four objects, we used five slots, one more than the number of objects, following the convention used in Van Steenkiste et al. (2018); Veerapaneni et al. (2020).

A.2 CONSTRUCTING NODES BY CLUSTERING STATES

For *block-rearrange*, we found that we obtained better clusterings when we used the SA attention mask α as the state s. For *robogym-rearrange*, we found that we obtained better clusterings when we used the action-dependent part of the SA slot $\boldsymbol{\lambda}^s$ as the state s. We also empirically found that certain choices of distance metric used for K-means `cluster`ing and `bind`ing (implemented as nearest-neighbors) depended on which choice of state representation we used, and this is summarized in Table 3. The K-means implmentation is adapted from `https://github.com/overshiki/kmeans_pytorch`.

When applying the trained dSLATE to the experience buffer to construct the graph we found that increasing the number of SA iterations improved the entity representations, so even though we trained

dSLATE with slot attention three iterations, for constructing the graph we used seven iterations. Lastly, we found that the number of clusters used to for K-Means is the most important hyperparameter for creating a graph that reflected the state transitions. We swept over 16 to 50 clusters and report the optimal number of clusters we found in Table 4.

State representation	α	λ^s
`isolate` distance metric	cosine	cosine
`cluster` distance metric	IoU	squared Euclidean
`bind` distance metric	cosine	squared Euclidean

Table 3: **Hyperparameters for constructing the transition graph with NCS**. This table shows the distance metrics we use for the `isolate`, `cluster`, and `bind` functions described in 4.1. For *block-rearrange* we use the SA attention mask α as the state s, and for *robogym-rearrange* we use the action-dependent part of the SA slot $\boldsymbol{\lambda}^s$ as the state s.

	block-rearrange	*robogym-rearrange*	*block-stacking*
number of clusters	30	45	47

Table 4: **Number of clusters used for constructing the nodes of the transition graph.**

A.3 Action selection

To implement `align` we use the `scipy.optimize.linear_sum_assignment` implementation of the Hungarian algorithm, with Euclidean distances between the z^k's as the matching cost.

Given the set of current entities $\mathbf{h}_t$ and goal constraints $\mathbf{h}_g$, `select-constraint` returns the index k of the goal constraint to satisfy next. By NCS' construction, the edge between the nodes that h_t^k and h_g^k are bound to is the state transition that would be executed if the action associated to the edge were taken in the environment. If NCS does not find an edge between the two nodes, such as if h_t^k and h_g^k were incorrectly bound to the graph, then NCS simply takes a random action. texttselect-constraint consists of two steps: (1) ranking transitions (2) sampling a transition.

Ranking The goal of the ranking step is to compute a ranking among the indices of $(h_g^1, ..., h_g^K)$ to choose which index k to actually select to affect with an action. Intuitively, we should rank indices k according to how different s_t^k and s_g^k are because a large difference would indicate that the constraint h_g^k is not satisfied, which means we would need to take an action to move the corresponding object represented by h_t^k. We reuse the distance metric $d(\cdot, \cdot)$ used for `isolate` to implement this ranking.

Sampling Given our ranking, the goal of the sampling step is to select a $k \in \{1, ..., K\}$ whose associated entity we will affect with an action. One way to do this is to simply choose k as $k = \arg\max_{k' \in \{1,...,\tilde{K}\}} d(s_t^{k'}, s_{t+1}^{k'})$ as in `isolate`, but we empirically found that sampling k from a categorical distribution whose pre-normalized probabilities are given by $d(s_t^{k'}, s_{t+1}^{k'})$ resulted in better task performance so we used this stochastic sampling approach. One explanation for why using the argmax may be worse is that it relies on the distance metric $d(\cdot, \cdot)$, and the state representation s, to be such that the distance metric flawlessly assigns high value to entities k that need to be moved and low value to entities k that do not need to be moved. But because the state space $\mathcal{S}$ is learned through the dSLATE training process without explicit supervision on the geometry of the space, a pair of points that should be farther apart than another set of points may not be accurately reflected by using a fixed distance metric $d(\cdot, \cdot)$. Future work will investigate imposing explicit supervision on the geometry of $\mathcal{S}$.

B Environment Details

Environments *Block-rearrange* is implemented in PyBullet (Coumans & Bai, 2016) while *robogym-rearrange* is implemented in Mujoco (Todorov et al., 2012).

Robogym-rearrange (see figures 7 and 8) is adapted from the rearrange environment in OpenAI's Robogym simulation framework (OpenAI, 2020) and removes the assumption from *block-rearrange*

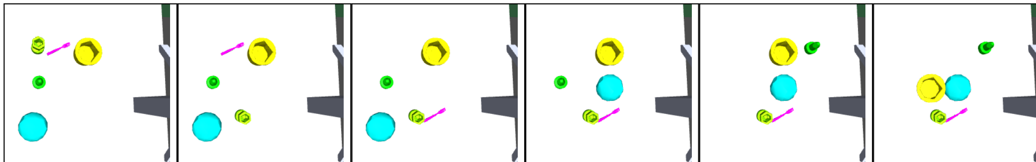


Figure 7: An example of solving a task in the robogym rearrange environment used in this paper.

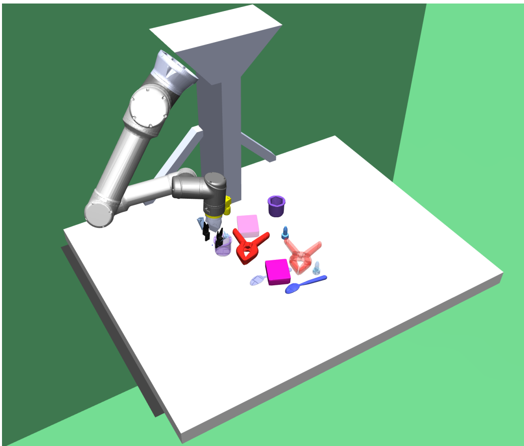


Figure 8: The original Robogym rearrange setup

that all objects are the same size, shape, and orientation and the assumption of predefined locations. Furthermore, due to 3D perspective, the objects can look slightly different in different locations. Objects are uniformly sampled from a set of 94 meshes consisting of the YCB object set Calli et al. (2015) and a set of basic geometric shapes, with colors sampled from a set of 13. The camera angle is a bird's eye view over the table, and the size of each object is normalized by its longest dimension, so tall thin objects appear smaller. The objects' target positions are randomly sampled such that they don't overlap with each other or any of the initial positions, and the target orientation is set to be unchanged. Because locations take continuous values, we define a match threshold of at most 0.05 for both the initial pick position and the goal placement (the table dimensions are 0.6 by 0.8).

Sensorimotor interface Each observation is a tuple of an initial image displaying the current observation and a goal image displaying constraints to be satisfied – the goal locations of the objects. Each action is a tuple $(w, \Delta w)$, where w is a three-dimensional Cartesian coordinate (x, y, z) in the environment arena. Objects are initialized at random non-overlapping locations that also do not overlap with their goal locations. For these tasks the z (height) coordinate is always fixed. An object is picked if w is within a certain threshold of its location. For *block-rearrange* where object locations are fixed points in a grid, the object is snapped to the nearest grid location to $w + \Delta w$. Constraints are considered satisfied if objects are placed within a certain threshold of their target location.

C Baseline Implementation Details

Random (Rand) The random policy takes actions using `env.action_space.sample()`.

Behavior cloning (BC) This approach trains a policy to output the actions directly taken in the provided dataset. We use an MSE loss to train the policy to imitate the actions.

Implicit Q-learning (IQL) IQL is a simple, offline RL approach that uses temporal difference (TD) learning with the dataset actions and trains a behavior policy value function. To produce an optimal value function, IQL estimates the maximum of the Q-function using expectile regression

with an asymmetric MSE using the following objectives:

$$L_V(\psi) = \mathbb{E}_{(s,a)\sim\mathcal{D}}[L_2^\tau(Q_{\hat{\theta}}(s,a) - V_\psi(s))] \text{ where } L_2^\tau(u) = |\tau - \mathbb{1}(u<0)|u^2 \quad (1)$$

$$L_Q(\theta) = \mathbb{E}_{(s,a,s')\sim\mathcal{D}}[(r(s,a) + \gamma V_\psi(s') - Q_\theta(s,a))^2] \quad (2)$$

$$L_\pi(\phi) = \mathbb{E}_{(s,a)\sim\mathcal{D}}[\exp(\beta(Q_{\hat{\theta}}(s,a) - V_\psi(s)))\log \pi_\phi(a|s)]. \quad (3)$$

The $V(s)$ estimates are used for TD-backups and the optimal policy is extracted with advantage-weighted behavioral cloning.

Model predictive control (MPC) This approach uses model predictive control with the cross entropy method (CEM) to select actions, using the transformer dynamics model of dSLATE to perform rollouts in latent space. This is similar to the approached used in OP3 (Veerapaneni et al., 2020), except that we use more recently proposed architectural components (slot attention (Locatello et al., 2020b) instead of IODINE (Greff et al., 2019), a transformer instead of a graph network (Battaglia et al., 2018; Van Steenkiste et al., 2018; Chang et al., 2016)) so our MPC results are not directly comparable to that of OP3. We use the same dSLATE checkpoint that was used for NCS.

We implement this MPC baseline using the `mbrl-lib` library (Pineda et al., 2021) with 10 CEM iterations, an elite ratio of 0.05, and a population size of 250 which was the best configuration we found that fit within a wall clock budget of two days for 8 objects and 100 test episodes. We swept over CEM iterations of $[5, 10, 20]$, elite ratio of $[0.05, 0.1, 0.2]$, and population sizes of $[250, 500, 1000]$, and found that the elite ratio was the most important hyperparameter.

The cost function is computed by first aligning the predicted slots $\mathbf{h}_T$ and goal constraints $\mathbf{h}_g$ using the same `align` procedure in Appendx. A.3, and then adding up the squared Euclidean distance between slots as $cost = \sum_k (h_T^k - h_g^k)^2$.

Non-factorized graph search (NF) This approach is an ablation to NCS that does not construct a graph over state transitions of individual entities but instead constructs a graph over state transition over entity sets, i.e. each transition is $(\mathbf{s}, a, \mathbf{s}')$ rather than $(s^k, a, s^{k\prime})$. As with MPC, we use the same dSLATE checkpoint that was used for NCS.

The purpose of this ablation is to elucidate the benefit of factorizing the transition graph over *individual entities* rather than *entity sets*. Because nodes in the transition graph for NF represent a set of entity states rather than individual entity states, we use Dijkstra's algorithm, as in (Eysenbach et al., 2019; Yang et al., 2020; Zhang et al., 2018) to plan a unbroken path from the node the initial observation is bound to to the node a goal observation is bound to. For each time-step, we plan a path along the nodes using Dijkstra's algorithm, then return the action associated with the first edge along that path. Like NCS, NF is a non-parametric model, which means that for a set of entities to be bound to a node in the graph, that node must contain the exact set of entity states corresponding to the states of the entities. If we do not successfully bind to the graph, or if we do not find a path between the current node and the goal node, we sample a random action as NCS does.

D Additional Results

This section presents additional results and analyses of NCS.

D.1 Analysis of key hyperparameters

In this section, we analyze the sensitivity of task performance to several hyperparameters used in NCS when creating the graph: the number of clusters, the number of examples from the experience buffer to use, and the number of slots used in slot attention. We perform this evaluation in the robogym environment with four objects in the complete goal specification. As Fig. 9 shows, performance depends on the number of initialized clusters and the number of batches from the training set used to construct the graph. With too few clusters, the clusters are too coarse-grained to differentiate objects in significantly different positions. With too many, the performance deteriorates as the data is needlessly split into duplicate clusters. Performance improves with more data, as the graph has better coverage. Although NCS performs worse when there are insufficient slots to represent all objects present in the environment, performance is barely impacted by having double the number of necessary slots. Our method can thus still work in environments with an unknown but upper-bounded number of objects.

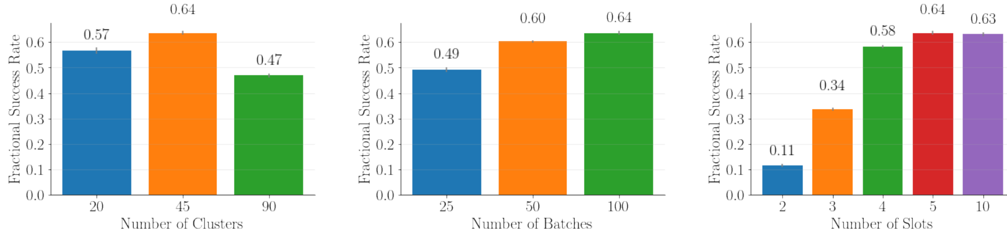
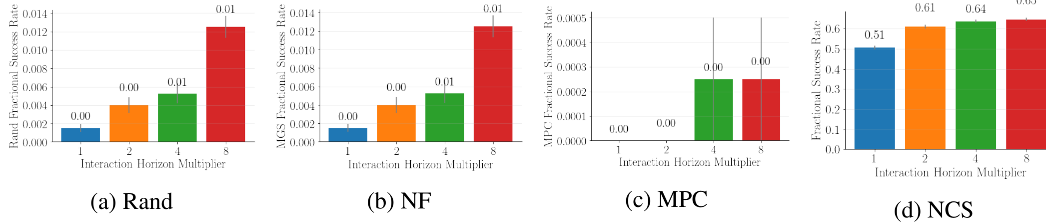


Figure 9: The performance of our method as the number of initialized clusters and batches from the training set used to construct the graph, and the number of slots are varied.

D.2 More computation time for model-based baselines

We tested whether doubling the computation time for the model-based baselines would improve their performance to be comparable to NCS's. For the results in the main paper, we capped the length of the episode as 4x the minimum number of actions required to solve the task. In Fig. 10, we vary this interaction horizon multiplier from 1x to 8x. NCS degrades less with shorter interaction horizons compared to the baselines. We find that NF performs similar to the random baseline. Since NF takes a random action if it cannot bind the given entity set to its graph, this result suggests that the space of subsets of entities is so combinatorially large that NF does not successfully bind to the graph most of the time. We verified that this is the case by inspecting when NF takes random actions. MPC performs the worst out of all the methods, performing worse than random. We tested that the cost function described in Appdx. C ranks latents that match the goal constraint with a lower cost than randomly sampled latents, which suggests that the main source of error is due to the inaccuracy in the prediction rollouts. This can be expected, as learned models suffer from compounding errors when rolled out (Janner et al., 2019) and prior methods that use MPC for object-centric methods only roll out for very short horizons (Veerapaneni et al., 2020).



(a) Rand (b) NF (c) MPC (d) NCS

Figure 10: **Varying interaction horizon.** The performance of the NF (b) and MPC (c) baselines compared to NCS (d, reproduced from Fig. 11) and the random baseline (a) on *robogym-rearrange* as we vary the interaction horizon (as a multiple of the minimum steps needed to complete the task). *Note that the scale of the y-axis is not the same.* While a longer horizon improves performance, NCS still achieves at least 50x better accuracy with an interaction horizon multiplier of 1 than the performance obtained by increasing the interaction horizon multiplier for the model-based baselines to 8.

D.3 More challenging settings

Finally, we analyzed NCS in more challenging settings that crudely emulate the noisy nature of real-world robotics. As Fig. 11 (left) shows, NCS is more robust than the baselines to the addition of Gaussian noise to the action at every time step, up until the noise variance is comparable to the maximum distance for successful picking and goal placements. The performance remains high given significantly fewer interaction steps (Fig. 11, right). Nevertheless, our success rate is still nowhere perfect, signifying much more work to do in scaling NCS to the real world.

E Combinatorial Space

This section details the calculation of the combinatorial size of the task space described in § 5. The number of object configurations in the initial state is $\binom{|S|}{k}$. In the complete specification setting, all objects must be moved, so $t \geq k$. At each step, any of the k occupied grid cells can be moved to any of the $\binom{|S|}{k}$ unoccupied grid cells, so the number of successor states is $k \times (|S| - k)$. With $|S|$ object locations and k objects, the number of possible trajectories over object configurations of t timesteps

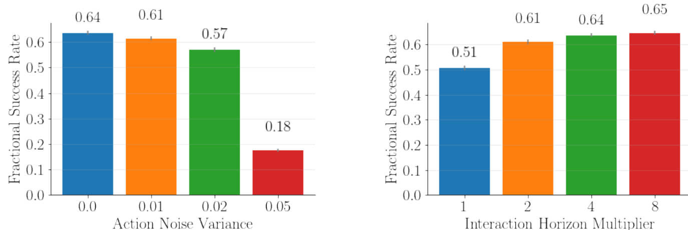


Figure 11: **Stress testing NCS** This figure shows the performance of NCS on *robogym-rearrange* as we vary the amount of noise added to the actions (left) and vary the interaction horizon, defined as a multiple of the minimum steps needed to complete the task (right).

is $\binom{|S|}{k} \times (k \times (|S| - k))^t$. For *block-rearrange* $|S| = 16$ so with $k = 7$ the number of possible trajectories is $\geq 4.5 \times 10^{16}$.

F Limitations and future work.

NCS relies on a nonparametric, non-learning-based approach for control to highlight the generalization capability of our representation of the combinatorial task space, but this limits NCS to only composing previously seen transitions for previously seen entities. Collapsing the combinatorial space along state transitions already provides significant gains but does not adapt to the introduction of novel objects at test time. NCS is currently implemented with tools such as SLATE and K-means that have much potential for improvement. We expect future variations of NCS will improve upon our results by replacing SLATE and K-means with their future successors.

Beyond the challenge of improving object-centric models to robustly model real pixels, extending our method to real world environments, such as those studied in Gokhale et al. (2019); Chang et al. (2020) would require overcoming the additional challenge of translating our high-level pick-and-move action primitives into motor torques for a real robot in a way that handles different object geometries, masses, and properties. Given that many works in learning robotics (e.g. Devin et al. (2020); Yang et al. (2021)) tackle this exact problem of goal-conditioned object grasping and manipulation, one potential approach to scale our method to real world environments is to train such goal-conditioned policies as the pick-and-move primitives for NCS to compose.

In this paper, we have assumed objects can be moved independently. Preliminary experiments suggest that NCS can be augmented to support tasks like block-stacking that involve dependencies among objects, but how to handle these dependencies would warrant a standalone treatment in future work.

G Why the name "Neural Constraint Satisfaction?"

NCS can be seen as physically solving a embodied constraint satisfaction problem, where states are variables, identities are variable values, and actions carry out variable assignments. Unlike symbolic constraint satisfaction, these variables, their domains, the assignment operator, and the constraints are all learned from the sensorimotor interface, hence the name Neural Constraint Satisfaction.