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

We found that we obtained better clusterings when we used the SA attention mask α as the state s for *block-rearrange* and when we used the action-dependent part of the SA slot $\boldsymbol{\lambda}^s$ as the state s for *robogym-rearrange*. We also empirically found that certain choices of distance metric used for K-means `clustering` and `binding` (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. textttselect-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 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.

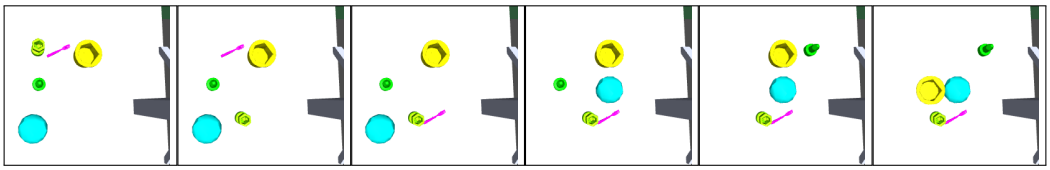


Figure 7: An example of solving a task in the robogym rearrange environment used in this paper.

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:

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

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

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

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.

C 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* 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.

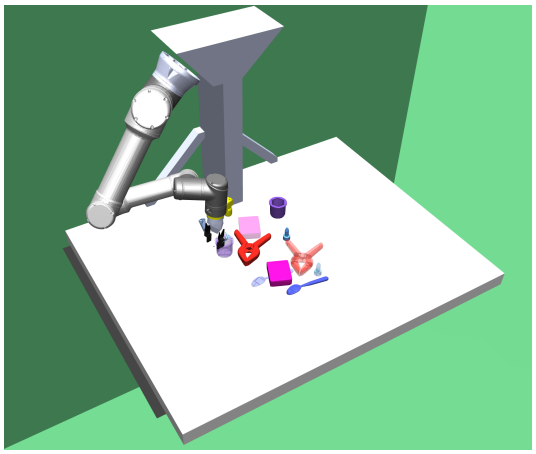


Figure 8: The original Robogym rearrange setup

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.

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.

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

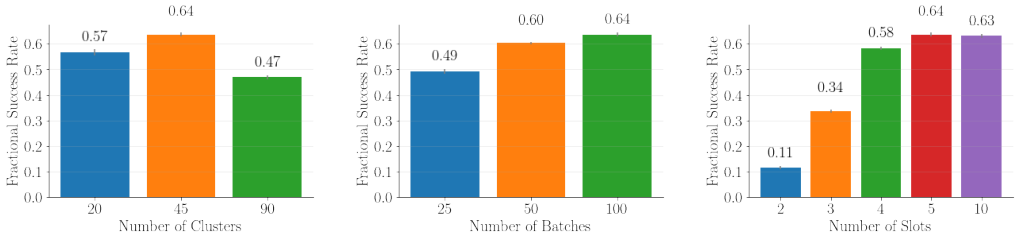
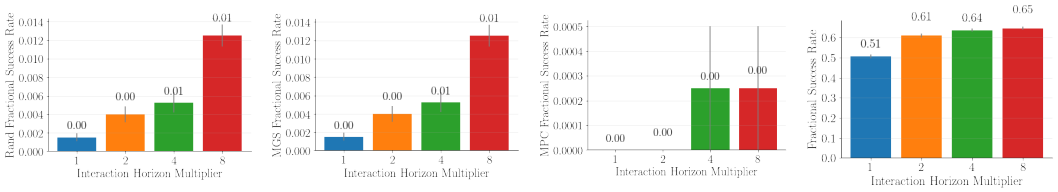


Figure 9: The performance of our method as the number of initialized clusters and batches from the training set used to construct the graph, and the number of slots are varied.

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. B 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 $k \times (|S| - k)$. 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

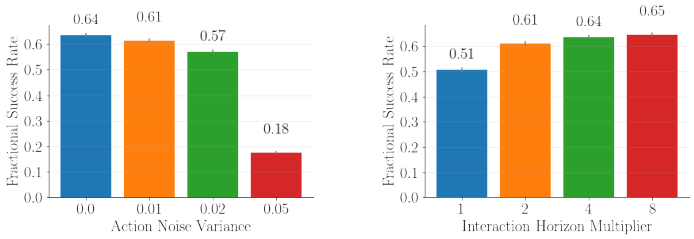


Figure 11: **Stress testing NCS** This figure shows the performance of NCS on *robogym-rearrange* as we vary the amount of noise added to the actions (left) and vary the interaction horizon, defined as a multiple of the minimum steps needed to complete the task (right).

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 probem, where states are variables, identities are variable values, and actions carry out variable assignments. Crucially the variables, their domains, the assignment operator, and the constraints are all learned from the sensorimotor interface, hence the name Neural Constraint Satisfaction.