Policy Architectures for Compositional Generalization in Control

Anonymous Authors[1]

Abstract

Several tasks in control, robotics, and planning can be specified through desired goal configurations for entities in the environment. Learning goal-conditioned policies is a natural paradigm to solve such tasks. Current approaches, however, struggle to learn and generalize as task complexity increases, such as due to variations in number of entities or compositions of goals. To overcome these challenges, we first introduce the Entity-Factored Markov Decision Process (EFMDP), a formal framework for modeling the entity-based compositional structure in control tasks. Subsequently, we outline policy architecture choices that can successfully leverage the geometric properties of the EFMDP model. Our framework theoretically motivates the use of Self-Attention and Deep Set architectures for control, and results in flexible policies that can be trained end-to-end with standard reinforcement and imitation learning algorithms. On a suite of simulated robot manipulation tasks, we find that these architectures achieve significantly higher success rates with less data, compared to the standard multilayer perceptron. Our structured policies also enable broader and more compositional generalization, producing policies that **extrapolate** to different numbers of entities than seen in training, and **stitch** together (i.e. compose) learned skills in novel ways. Video results can be found at https://sites.google.com/view/comp-gen-anon.

1. Introduction

Goal specification is a powerful abstraction for training and deploying AI agents (Kaelbling, 1993; Schaul et al., 2015; Andrychowicz et al., 2017). For instance, object reconfiguration tasks (Batra et al., 2020), like loading plates in a dishwasher or arranging pieces on a chess board, can be described through spatial (6DOF pose) and semantic (on vs off) goals for various objects. Furthermore, a broad goal for a scene can be naturally described through compositions of goals for individual entities. In this work, we introduce a framework for modeling tasks with this *entity-centric* compositional structure, and study policy architectures that can utilize such structural properties. Our framework and designs are broadly applicable for goal-conditioned reinforcement and imitation learning. Through experiments in a suite of simulated robot manipulation environments, we find that our policy architectures learn substantially faster compared to standard multi-layer perceptrons (MLPs) and demonstrate significantly improved generalization capabilities, a preview of which is depicted in Figure 1.

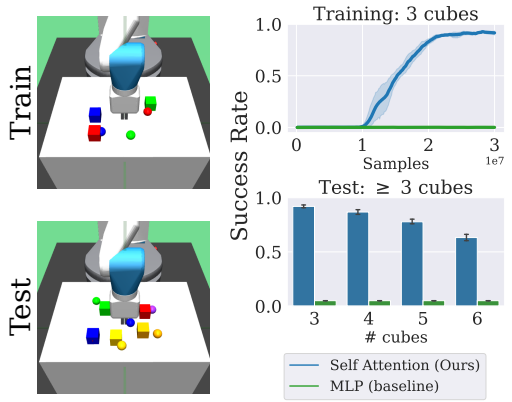

Figure 1. A family of tasks where the agent is trained to re-arrange three cubes (top-left), but tested zero-shot to re-arrange anywhere from three to six cubes (bottom-left). Reinforcement learning (RL) with standard MLPs fails to even learn the 3-cube task. Our self-attention based policy can learn the three cube task, and also solve tasks with more cubes using no additional data.

Consider the motivating task of arranging pieces on a chess board using a robot arm. A naive specification would provide goal locations for all 32 pieces simultaneously. However, we can immediately recognize that the task is a composition of 32 sub-goals involving the rearrangement of individual pieces. This understanding of compositional structure can allow us to focus on one object at a time, dramatically reducing the size of effective state space and help combat the curse of dimensionality that plagues RL (Sutton & Barto, 1998; Bertsekas & Tsitsiklis, 1996). Moreover, such a compositional understanding would make an agent invariant to the number of objects, enabling generalization to fewer or more objects. Most importantly, it can enable reusing shared skills like pick-and-place, enhancing learning efficiency. Finally, we also note that a successful policy cannot completely decouple the sub-tasks and must consider their interactions. For example, if a piece must be moved to

a square currently occupied by another piece, the piece in the destination square must be moved first.

Standard policy architectures based on MLPs lack the inductive biases and structure to exhibit the aforementioned compositional properties. To overcome these deficiencies, we turn to the general field of "geometric deep learning" (Bronstein et al., 2021) which is concerned with the study of structures, symmetries, and invariances exhibited by function classes. We first introduce the Entity-Factored MDP as a formal model of decision making in environments with multiple entities (e.g. objects), and characterize its geometric properties relative to the generic MDP. We subsequently show how set-based invariant architectures like Deep Sets (Zaheer et al., 2017) and relational architectures like Self-Attention (Vaswani et al., 2017) are well suited to leverage the geometric properties of the EFMDP. Through experiments, we demonstrate that policies and critics parameterized by these architectures can be trained to solve complex tasks using standard RL and IL algorithms, without assuming access to any options or action primitives.

Our Contributions. This paper is organized into sections that present our three main contributions:

1. We develop the Entity-Factored MDP (EFMDP) framework, a formal model for decision making in tasks comprising of multiple entities (e.g. objects), and characterize its geometric properties.
2. We show how policies and critics parameterized by set-based invariance models (e.g. Deep Sets) and relational models (e.g. Self-Attention) can leverage the geometric properties of the EFMDP.
3. We empirically evaluate our theoretically inspired architectures on a suite of simulated robot manipulation tasks, and find that they generalize more broadly compared to the standard MLP, while also learning more efficiently.

Although invariant and relational policy designs have been explored in the literature, this work represents the first direct comparison of both architecture classes in a standard suite of complex entity-centric environments. Furthermore, our experiments evaluate both **extrapolation** to varying numbers of entities in the environment, as well as **stitching**: the ability to solve unseen tasks by composing together learned skills in novel ways.

2. Related Work

We describe and study a class of learning problems where an agent must learn in an environment containing several entities. Prior works have explored solving such problems with relational models such as graph neural networks (Huang et al., 2019; Bapst et al., 2019; Li et al., 2020; Veerapaneni et al., 2020; Lin et al., 2022), transformers (Zambaldi et al., 2018; Carvalho et al., 2021; Tang & Ha, 2021), or other means (Goyal et al., 2019). In contrast, simple invariant architectures like Deep Sets (Zaheer et al., 2017) remain relatively underexplored in RL outside of basic 2D environments (Karch et al., 2020). We compare *both* transformer and Deep Set approaches on a suite of complex entity-centric robot tasks. We also introduce the EFMDP framework for entity-based compositionality in RL which theoretically motivates both architecture types via their inherent relational and invariant properties.

3. Problem Formulation and Architectures

In this section, we first formalize our problem setup by introducing the entity-factored MDP. Then we introduce policy architectures that can enable efficient learning and generalization by utilizing the unique structural properties of the entity-factored MDP.

3.1. Problem Setup

We study a learning paradigm where the agent can interact with many entities in an environment. The task for the agent is specified in the form of goals for some subset of entities (including the agent). We formalize this learning setup with the Entity-Factored Markov Decision Process (EFMDP).

Definition 1 (Entity-Factored MDP)**.** *An EFMDP with N entities is described through the tuple: $\mathcal{M}^E := \langle \mathcal{U}, \mathcal{E}, \mathfrak{g}, \mathcal{A}, \mathbb{P}, \mathcal{R}, \gamma \rangle$. Here $\mathcal{U}$ and $\mathcal{E}$ are the agent and entity state spaces, $\mathfrak{g}$ is the subgoal space and $\mathcal{A}$ is the agent's action space. The overall state space $\mathcal{S} := \mathcal{U} \times \mathcal{E}^N$ has elements $\mathbf{s} = (u, e_1, \cdots, e_N)$ and the overall goal space $\mathcal{G} := \mathfrak{g}^N$ has elements $\mathbf{g} = (g_1, \ldots, g_N)$. The reward and dynamics are described by:*

$$\mathcal{R}(\mathbf{s}, \mathbf{g}) := \mathcal{R}\left(\{\tilde{r}(e_i, g_i, u)\}_{i=1}^N\right) \tag{1}$$

$$\mathbb{P}(\mathbf{s}'|\mathbf{s}, \mathbf{a}) := \mathbb{P}\left(\left(u', \{e_i'\}_{i=1}^N\right) \mid \left(u, \{e_i\}_{i=1}^N\right), \mathbf{a}\right) \tag{2}$$

for $\mathbf{s}, \mathbf{s}' \in \mathcal{S}$, $\mathbf{a} \in \mathcal{A}$, and $\mathbf{g} \in \mathcal{G}$.

A wide variety of tasks involve an agent interacting with different entities in the environment, and can be cast as EFMDPs. At the same time, the EFMDP contains more structure compared to the standard MDP model, because the ordering of the entity-subgoal pairs is arbitrary. In fact, we prove that any optimal policy and the optimal value function are permutation invariant (proof in Appendix A):

Proposition 1 (Policy and Value Invariance)**.** *In any EFMDP with N entities, any optimal policy $\pi^\star : \mathcal{S} \times \mathcal{G} \to \mathcal{A}$ and optimal action-value function $Q^\star : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$ are both invariant to permutations of the entity-subgoal pairs. That is, for any $\sigma \in S_N$, $\pi^\star(\sigma\mathbf{s}, \sigma\mathbf{g}) = \pi^\star(\mathbf{s}, \mathbf{g})$ and $Q^\star(\sigma\mathbf{s}, \mathbf{a}, \sigma\mathbf{g}) = Q^\star(\mathbf{s}, \mathbf{a}, \mathbf{g})$.*

This invariance property motivates us to design invariant architectures for reinforcement and imitation learning on

EFMDPs.

Multilayer Perceptrons (MLPs). Standard RL and IL approaches assume they are solving a generic MDP, and do not use any additional structure. The generic approach is thus to parameterize the learned policy by an MLP, which takes a fixed size input vector and applies alternating layers of affine transforms and pointwise nonlinearities to produce a fixed size output vector. To implement $\pi(\mathbf{s}, \mathbf{g})$ with an MLP we concatenate the contents of $(\mathbf{s}, \mathbf{g})$ into a single long vector which forms the MLP input. Since MLPs expect input vectors of a fixed dimension, testing on tasks with more entities requires zero padding during training.

Deep Sets. Recall that the optimal policy in an EFMDP should be permutation invariant (Prop. 1). The MLP policy fails to guarantee this invariance, but we can use the Deep Sets (Zaheer et al., 2017) architecture to guarantee permutation invariance to a set of input vectors $x = \{x_1, \cdots, x_N\}$. In EFMDPs we choose this set of vectors to be the entity-subgoal pairs $\{(e_1, g_1), \cdots, (e, g_N)\}$, so that the learned policy will have the invariance from Prop. 1. Figure 2 visualizes how the input is arranged and processed by Deep Sets.

Self Attention. In tasks involving complex entity-entity interaction Self-Attention (Vaswani et al., 2017) can provide a superior inductive bias for modeling entity relationships. To integrate Self Attention into a policy, we treat the input state as a *sequence* of entity-subgoal pairs. We process this sequence using an encoder architecture described in Vaswani et al. (2017), but do **not** use causal masking or positional encodings so as to preserve self-attention's permutation symmetry. The entire encoder $\text{SA}(\cdot)$ maps an input sequence to another sequence of identical dimensions while modeling relationships between the sequence elements.

Since the policy must produce a single vector as output, we pool the z_i's together by summation and project the result to an action using a small MLP. The entire self attention policy's design is illustrated in Figure 2. Without the positional encodings, the output is invariant to permutations of the entities and subgoals, much like the Deep Set policy. However, the self-attention mechanism produces intermediate representations z that includes interactions between the inputs, unlike Deep Set's independent intermediate representations.

4. Experiments and Evaluation

In this section, we aim to study the following questions through our experimental evaluation.

1. Can we **learn** more efficiently by using policies that utilize the structures and invariances of the EFMDP?
2. Can the structured policies generalize better and enable **extrapolation** to more or fewer entities?
3. Can the structured policies solve tasks containing novel *combinations* of subtasks, by **stitching** together (i.e. composing) learned skills?

Extrapolation and stitching are particularly interesting as they require generalization to novel tasks with no additional training. This is particularly useful when deploying agents in real world settings with enormous task diversity.

Environment Description. We seek to answer our experimental questions in a suite of simulated robotic manipulation environments, where the policy provides low level continuous actions to control a Fetch robot and interact with any number of cubes and switches. There are three subtasks: to *push* a cube to a desired location on the table, to flip a *switch* to a specified setting, or to *stack* one cube on top of another. The higher level tasks can involve multiple cubes or switches and compose many subtasks together.

We organize the environments into **families** to test learning and generalization. Environments in the *N-Push* family require re-arranging N cubes by pushing each one to its corresponding subgoal. The *N-Switch* family requires flipping each of N switches to its specified setting, and the *N-Switch + N-Push* family involves re-arranging N cubes *and* flipping N switches. We test extrapolation by varying N within a family at test time, which changes the number of entities: for example we train a policy in *3-Switch* and evaluate it in *6-Switch*. As another example, we test stitching by training a single policy on *2-Switch* and *2-Push*, then evaluate it on *2-Switch + 2-Push* which requires combining the switch and pushing skills together in a single trajectory. Appendix B gives a full description of our environments.

Baselines and Comparisons. Our main comparisons are with: (a) a baseline MLP that models the task as a regular MDP, and (b) an "oracle" that manually coordinates solving one subtask at a time (Appendix B).

4.1. Efficiency of Learning

To evaluate the learning efficiency of different architectures, we train RL and IL algorihtms on the *N-Switch*, *N-Push*, and *N-Switch + N-Push* environment families. We use $N = 1, 2, 3$ for the first two families and $N = 1, 2$ for the latter, with larger N corresponding to more entities and more complex tasks within a family. See Appendix C for full RL and IL training details.

Results. Appendix Fig. 4 shows full RL and IL training curves. MLP policies struggle to learn complex tasks with many entities, likely due to the lack of entity-centric processing that the Deep Set and Self Attention policies employ. Overall, this experiment suggests that architectures that utilize the structure and invariances in EFMDPs learn substantially faster than black box architectures like the MLP.

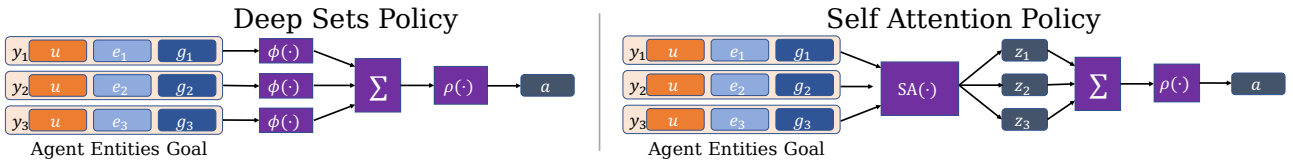

Figure 2. Visualizations of implementing an entity-based goal conditioned policy using either Deep Sets (left) or Self Attention (right). The policy $\pi : (\mathbf{s}, \mathbf{g}) \mapsto \mathbf{a}$ receives state $\mathbf{s} = (u, e_1, \cdots, e_N)$ containing agent state u and entity states e_i. The goal $(g_1, \cdots, g_N)$ contains subgoals for each entity. Both policies arrange the input into N vectors $y_i = (u, e_i, g_i)$, one per entity. The Deep Set policy processes each y_i independently with MLP $\phi(\cdot)$, aggregates the outputs, and maps the result to an action using MLP $\rho(\cdot)$. The self attention encoder $\text{SA}(\cdot)$ produces output $z_1, \cdots, z_N$ and uses self-attention to model interactions between the entities/subgoals. The z_i are mapped to an action by summation and an MLP $\rho(\cdot)$.

4.2. Zero-Shot Extrapolation Capabilities

Here we test whether trained policies can extrapolate and solve test tasks containing more or fewer entities than seen in training.

Results and Observations Appendix figure 5 shows the test performance of these policies on each environment family as the number of entities N varies. The MLP only successfully learns the training task in the *N-Switch* environments, and it generalizes decently to *fewer* than 3 switches, but fails completely in environments with *more* than 3 switches. In contrast, the Deep Set and Self Attention policies generalize well and achieve zero-shot success rates comparable to or exceeding the Oracle in most test environments. The Deep Set policy extrapolates better than the Self Attention policy on most environments. Overall, we find that by using architectures capable of utilizing the EFMDP structure, agents can perform very effective extrapolation.

4.3. Zero-Shot Stitching to solve novel tasks

When evaluating policies for stitching behavior, we use test tasks that combine subtasks from training in novel ways. In our first setting, we train a policy on *2-Push* and *2-Switch*, and then test this policy on *2-Switch + 2-Push*, which requires both pushing cubes *and* flipping switches. In our second setting, we train a single policy on *2-Push* and *Stack*, which requires stacking one cube on top of another. The test environment is *Push + Stack*, which requires pushing one cube into position and then stacking the other block on top. This setting is especially difficult because it requires zero-shot stitching of skills in a particular order (push, *then* stack). Appendix figure 6 (left) shows the train-test task relationships we use to test stitching.

Results and Observations. Appendix figure 6 (right) shows that the MLP policy fails to jointly learn the training tasks in the first setting, leading to poor performance in *2-Switch + 2-Push*. The MLP averages above a 35% success rate on both training tasks in the second setting, but still only manages a 5% success rate on *Push + Stack*. This suggests that even when MLP policies are capable of learning the training tasks, they are unable to combine them to solve new ones.

The Deep Set and Self Attention architectures show substantially better stitching capabilities compared to the MLP, though they are not as competent as the Oracle. It is particularly surprising that the Self Attention policy achieves a 60% zero-shot success rate on *Push + Stack*, which requires understanding that the push and stack subtasks must be executed in a specific order. Poor performance in *2-Switch + 2-Push* is again likely due to difficulties in training one policy on two different tasks, which suggests that improving joint training could further improve stitching performance.

5. Conclusion

In this work, we introduced the EFMDP framework for the learning paradigm where an agent can interact with many entities in an environment. We explore the structural properties of this framework like invariance of the reward and dynamics under permutation symmetry. Using this structure, we showed that the optimal policy and value function in the EFMDP are also invariant to permutations of the entities.

Building on the above result, we introduced policy architectures based on Self-Attention and Deep Sets that can leverage the symmetries and invariances in the EFMDP. More specifically, these policy architectures decompose goal-conditioned tasks into their constituent entities and subgoals. These architectures are flexible, do not require any manual task annotations or action primitives, and can be trained end-to-end with standard RL and IL algorithms.

Experimentally, we find that our policy architectures that utilize the EFMDP structure can: (a) **learn substantially faster** than black-box architectures like the MLP; (b) perform **zero-shot extrapolation** to new environments with more of fewer entities than observed in training; and (c) perform **zero-shot stitching** of learned behaviors to solve novel task combinations never seen in training.

References

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.

Bapst, V., Sanchez-Gonzalez, A., Doersch, C., Stachenfeld, K., Kohli, P., Battaglia, P., and Hamrick, J. Structured agents for physical construction. In *International Conference on Machine Learning*, pp. 464–474. PMLR, 2019.

Batra, D., Chang, A. X., Chernova, S., Davison, A. J., Deng, J., Koltun, V., Levine, S., Malik, J., Mordatch, I., Mottaghi, R., et al. Rearrangement: A challenge for embodied ai. *arXiv preprint arXiv:2011.01975*, 2020.

Bertsekas, D. and Tsitsiklis, J. *Neuro-dynamic Programming*. 1996.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

Bronstein, M. M., Bruna, J., Cohen, T., and Veličković, P. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

Carvalho, W., Liang, A., Lee, K., Sohn, S., Lee, H., Lewis, R. L., and Singh, S. Roma: A relational, object-model learning agent for sample-efficient reinforcement learning. 2021.

Goyal, A., Lamb, A., Hoffmann, J., Sodhani, S., Levine, S., Bengio, Y., and Schölkopf, B. Recurrent independent mechanisms. *arXiv preprint arXiv:1909.10893*, 2019.

Huang, D.-A., Nair, S., Xu, D., Zhu, Y., Garg, A., Fei-Fei, L., Savarese, S., and Niebles, J. C. 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.

Kaelbling, L. P. Learning to achieve goals. In *IJCAI*, pp. 1094–1099. Citeseer, 1993.

Karch, T., Colas, C., Teodorescu, L., Moulin-Frier, C., and Oudeyer, P.-Y. Deep sets for generalization in rl. *arXiv preprint arXiv:2003.09443*, 2020.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Lanier, J. B. *Curiosity-driven multi-criteria hindsight experience replay*. University of California, Irvine, 2019.

Li, R., Jabri, A., Darrell, T., and Agrawal, P. Towards practical multi-object manipulation using relational reinforcement learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4051–4058. IEEE, 2020.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Lin, Y., Wang, A. S., Undersander, E., and Rai, A. Efficient and interpretable robot manipulation with graph neural networks. *IEEE Robotics and Automation Letters*, 2022.

Schaul, T., Horgan, D., Gregor, K., and Silver, D. Universal value function approximators. In *International conference on machine learning*, pp. 1312–1320. PMLR, 2015.

Sutton, R. and Barto, A. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

Tang, Y. and Ha, D. The sensory neuron as a transformer: Permutation-invariant neural networks for reinforcement learning. *Advances in Neural Information Processing Systems*, 34, 2021.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Veerapaneni, R., Co-Reyes, J. D., Chang, M., Janner, M., Finn, C., Wu, J., Tenenbaum, J., and Levine, S. Entity abstraction in visual model-based reinforcement learning. In *Conference on Robot Learning*, pp. 1439–1456. PMLR, 2020.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R., and Smola, A. Deep sets. *arXiv preprint arXiv:1703.06114*, 2017.

Zambaldi, V., Raposo, D., Santoro, A., Bapst, V., Li, Y., Babuschkin, I., Tuyls, K., Reichert, D., Lillicrap, T., Lockhart, E., et al. Relational deep reinforcement learning. *arXiv preprint arXiv:1806.01830*, 2018.

A. Permutation invariance

We start by formally defining the permutation symmetry that EFMDPs have:

Property 1 (EFMDP Permutation Symmetry)**.** *For any permutation $\sigma \in S_N$ (the group of all permutations of N items), the reward satisfies $\mathcal{R}(\sigma\mathbf{s}, \sigma\mathbf{g}) = \mathcal{R}(\mathbf{s}, \mathbf{g})$ and the transition dynamics satisfy $\mathbb{P}(\sigma\mathbf{s}'|\sigma\mathbf{s}, \mathbf{a}) = \mathbb{P}(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ for any $\mathbf{s}, \mathbf{s}' \in \mathcal{S}$ and $\mathbf{a} \in \mathcal{A}$, where:*

$$\sigma\mathbf{s} := (u, e_{\sigma(1)}, \cdots, e_{\sigma(N)}) \tag{3}$$
$$\sigma\mathbf{g} := (g_{\sigma(1)}, \cdots, g_{\sigma(N)}) \tag{4}$$

We now want to show Proposition 1:

Proposition 1 (Policy and Value Invariance)**.** *In any EFMDP with N entities, any optimal policy $\pi^\star : \mathcal{S} \times \mathcal{G} \to \mathcal{A}$ and optimal action-value function $Q^\star : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$ are both invariant to permutations of the entity-subgoal pairs. That is, for any $\sigma \in S_N$, $\pi^\star(\sigma\mathbf{s}, \sigma\mathbf{g}) = \pi^\star(\mathbf{s}, \mathbf{g})$ and $Q^\star(\sigma\mathbf{s}, \mathbf{a}, \sigma\mathbf{g}) = Q^\star(\mathbf{s}, \mathbf{a}, \mathbf{g})$.*

We want to show that any optimal policy $\pi^\star : \mathcal{S} \times \mathcal{G} \to \mathcal{A}$ and the optimal action-value function $Q^\star : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$ are both permutation invariant, that is for any $\sigma \in S_N$:

$$\pi^\star(\sigma\mathbf{s}, \sigma\mathbf{g}) = \pi^\star(\mathbf{s}, \mathbf{g}) \tag{5}$$
$$Q^\star(\sigma\mathbf{s}, \mathbf{a}, \sigma\mathbf{g}) = Q^\star(\mathbf{s}, \mathbf{a}, \mathbf{g}) \tag{6}$$

Recall that in an EFMDP the reward and dynamics have permutation symmetry:

$$\mathcal{R}(\mathbf{s}, \mathbf{a}, \mathbf{g}) = \mathcal{R}(\sigma\mathbf{s}, \mathbf{a}, \sigma\mathbf{g})$$
$$\mathbb{P}(\mathbf{s}'|\mathbf{s}, \mathbf{a}) = \mathbb{P}(\sigma\mathbf{s}'|\sigma\mathbf{s}, \mathbf{a})$$

where $\sigma\mathbf{s}$ and $\sigma\mathbf{g}$ are defined in Eq. 3 and Eq. 4. We assume for simplicity that the agent space $\mathcal{U}$ and entity space $\mathcal{E}$ are discrete, so that the state space $\mathcal{S} = \mathcal{U} \times \mathcal{E}^N$ is also discrete.

We begin with $Q^\star$, which can be obtained by value iteration, where $Q^\star_k$ denotes the k'th iterate. We initialize $Q^\star_0 \equiv 0$, which is (trivially) permutation invariant. Permutation invariance is then preserved during each step of value iteration $Q^\star_k \mapsto Q^\star_{k+1}$:

$$Q^\star_{k+1}(\sigma\mathbf{s}, \mathbf{a}, \sigma\mathbf{g}) = \mathcal{R}(\sigma\mathbf{s}, \mathbf{a}, \sigma\mathbf{g}) + \gamma \max_{\mathbf{a}'} \sum_{\mathbf{s}' \in \mathcal{S}} \mathbb{P}(\mathbf{s}'|\sigma\mathbf{s}, \mathbf{a}) Q^\star_k(\mathbf{s}', \mathbf{a}') \tag{7}$$
$$= \mathcal{R}(\mathbf{s}, \mathbf{a}, \mathbf{g}) + \gamma \max_{\mathbf{a}'} \sum_{\mathbf{s}' \in \mathcal{S}} \mathbb{P}(\sigma^{-1}\mathbf{s}'|\mathbf{s}, \mathbf{a}) Q^\star_k(\sigma^{-1}\mathbf{s}', \mathbf{a}') \tag{8}$$
$$= \mathcal{R}(\mathbf{s}, \mathbf{a}, \mathbf{g}) + \gamma \max_{\mathbf{a}'} \sum_{\mathbf{s}' \in \mathcal{S}} \mathbb{P}(\mathbf{s}'|\mathbf{s}, \mathbf{a}) Q^\star_k(\mathbf{s}', \mathbf{a}') \tag{9}$$
$$= Q^\star_{k+1}(\mathbf{s}, \mathbf{a}, \mathbf{g}) \tag{10}$$

Hence $Q^\star_k$ is permutation invariant for all $k = 0, 1, \cdots$, with $Q^\star_k \xrightarrow[k\to\infty]{} Q^\star$. Line 8 follows from the permutation invariance of the reward, transition probability, and the previous iterate $Q^\star_k$. Line 9 uses the fact that summing over $\sigma^{-1}\mathbf{s}'$ for all $\mathbf{s}' \in \mathcal{S}$ is the same as simply summing over all states $\mathbf{s}' \in \mathcal{S}$. This can be seen more explicitly by expanding a sum over arbitrary function $f(\cdot)$:

$$\sum_{\mathbf{s} \in \mathcal{S}} f(\sigma^{-1}\mathbf{s}) = \sum_{u \in \mathcal{U}} \sum_{e_1 \in \mathcal{E}} \cdots \sum_{e_N \in \mathcal{E}} f(u, e_{\sigma^{-1}(1)}, \cdots, e_{\sigma^{-1}(N)}) = \sum_{\mathbf{s} \in \mathcal{S}} f(\mathbf{s})$$

The permutation invariance of $Q^\star$ leads to the permutation invariance of $\pi^\star$:

$$\pi^\star(\sigma\mathbf{s}, \sigma\mathbf{g}) = \arg\max_{\mathbf{a}} Q^\star(\sigma\mathbf{s}, \mathbf{a}, \sigma\mathbf{g}) = \arg\max_{\mathbf{a}} Q^\star(\mathbf{s}, \mathbf{a}, \mathbf{g}) = \pi^\star(\mathbf{s}, \mathbf{g})$$

B. Environments

Our environments are modified from OpenAI Gym's Fetch environments (Brockman et al., 2016), with our stacking environment in particular being modified from the Fetch stacking environments of Lanier (2019). They have a 4D continuous

action space with 3 values for end effector displacement and 1 value for controlling the distance between the gripper fingers. The final action is disabled when the neither the training or test tasks involve stacking, since gripping is not required for block pushing or switch flipping. Input actions are scaled and bounded to be between $[-1, 1]$. We set the environment episode length based on the number of entities and subtasks involved. Each switch added 20 timesteps, and each cube pushing or stacking task added 50 timesteps. For example, *2-Switch + 2-Push* had a max episode length of $2 \times 50 + 2 \times 20 = 140$ timesteps.

For non-stacking settings such as *N-Push* and *N-Switch + N-Push*, we disable cube-cube collision physics to make training easier for all methods. Note that subgoals may still interfere with each other since the gripper can interact with all cubes, so the agent may accidentally knock one cube away when manipulating another one. We repeat the extrapolation experiments for *N-Push with collisions* in Appendix D.2.

State and goals. The agent state describe the robot's end effector position and velocity the gripper finger's positions and velocities. The entity state for cubes include the cube's pose and velocity, and for switches include the switch setting $\theta \in [-0.7, 0.7]$ and the position of the switch base on the table. The switch entity state is padded with zeros to match the shape of the cube entity state, and all entity states include an extra bit to distinguish cubes from switches. Subgoals specify a target position for cubes and a target setting $\theta^\star \in \{-0.7, 0.7\}$ for switches.

Reward. The dense reward is defined as the average distance between each entity and its desired state as specified by the subgoal. For cubes, this is the L2 distance between current and desired position. For switches, this is $|\theta - \theta^\star|$, where θ is the current angle of the switch and $\theta^\star$ is the desired setting. The sparse reward is 0 if all entities are within a threshold distance of their subgoals, and -1 otherwise.

Oracle. We construct subpolicies for the oracle by training one policy on each distinct subtask (pushing, flipping switches, and stacking). The oracle chooses an initial entity and subgoal arbitrarily, and uses the corresponding subpolicy until that subtask is solved. The oracle then selects the appropriate subpolicy for the next entity-subgoal pair and continues until the entire task is complete. The oracle is **not** guaranteed to achieve a 100% success rate since it does not consider entity-entity interactions. An example failure mode is pushing one cube into position but knocking another one off the table while doing so.

C. Training details

C.1. Reinforcement learning

We train RL agents using a publicly available implementation[1] of DDPG (Lillicrap et al., 2015) and Hindsight Experience Replay (HER) (Andrychowicz et al., 2017). Table 1 contains the default hyperparameters shared across all experiments. Our modified implementation collects experience from 16 environments in parallel into a single replay buffer, and trains the policy and critic networks on a single GPU. We collect 2 episodes for every 5 gradient updates, and for HER we relabel the goals in 80% of sampled minibatches (the "relabel prob"). The reward scale is simply a multipler of the collected reward used during DDPG training. For exploration we use action noise η and random action probability ϵ; the output action is:

$$\tilde{a} \sim \begin{cases} a + \mathcal{N}(0, \eta), & \text{with prob } 1 - \epsilon \\ \text{Uniform}(-1, 1), & \text{with prob } \epsilon \end{cases}$$

Table 2 shows environment specific RL hyperparameters. "Epochs" describes the total amount of RL training done, with 1 epoch corresponding to $50 \times$ PARALLEL ENVS episodes. Sparse reward is used for the simpler environments, and dense reward for the harder ones. For some environments we decay the exploration parameters η, ϵ by a ratio computed per-epoch. Lin(.01, 100, 150) means that η, ϵ are both decayed linearly from η_0 and ϵ_0 to $.01 \times \eta_0$ and $.01 \times \epsilon_0$ between epochs 100 and 150. The constant exploration decay schedule maintains the initial η_0, ϵ_0 values throughout training. The target network parameters are updated as $\theta^{\text{target}} \leftarrow (1 - \tau)\theta + \tau\theta^{\text{target}}$, where τ is the target update speed.

We use the same RL hyperparameters regardless of architecture type except that the learning rate is lower for Self Attention and the exploration decay schedule may vary. Where Table 1 lists "Fast" and "Slow" decay schedules, we sweep over both options for each architecture and use the schedule that works best. In each case, the Self Attention policy prefers the slower exploration schedule and Deep Sets prefers the faster one, while the MLP typically fails to learn with either exploration

[1] https://github.com/TianhongDai/hindsight-experience-replay

Table 1. General shared RL hyperparameters

Hyperparameter	Value
Discount γ	0.98
Parallel envs	16
Replay buffer size	10^6
Relabel prob	0.8
Ratio of episodes : updates	$2:5$
Optimizer	Adam
Learning rate	MLP, Deep Set: 0.001 Self Attention: 0.0001
Reward Scale	Sparse: 1; Dense: 5
Action noise η_0 (initial)	0.2
Random action prob ϵ_0 (initial)	0.3

Table 2. Environment specific RL hyperparameters

Environment	Reward	Epochs	Exploration decay	Target update speed τ
1-Push	Sparse	50	Constant(1)	0.95
2-Push	Dense	150	Lin(.01, 75, 125)	0.99
3-Push	Dense	250	Fast: Lin(.01, 30, 80) Slow: Lin(.01, 100, 175)	0.99
{1,2,3}-Switch	Sparse	{10, 50, 100}	Constant(1)	0.95
1-Switch + 1-Push	Dense	150	Lin(.01, 60, 100)	0.99
2-Switch + 2-Push	Dense	250	Fast: Lin(.01, 75, 150) Slow: Lin(.01, 100, 150)	0.99

schedule on the more complex environments.

Architectures. The exact actor and critic architectures uses for each architecture family is shown in Table 3. Linear(256) represents an affine layer with 256 output units. ReLU activations follow every layer except the last. The final actor layer is followed by a Tanh nonlinearity, and the critic has no activation function after the final layer. A represents the action space dimension, and Block(N, M, H) represents a Transformer encoder block (Vaswani et al., 2017) with embedding size N, feedforward dimension M, and H heads. We disable dropout within the Transformer blocks for RL training.

C.2. Imitation Learning

The IL dataset is generated using the best performing RL agent in that environment–we record $M \in \{1000, 2000, 3000, 4000, 5000\}$ demonstration trajectories. This creates a dataset of $M \times T$ transitions $\mathcal{D} = \{(\mathbf{s}_i, \mathbf{a}_i)\}_{i=1}^{M \times T}$ for behavior cloning. However, in practice we filter the dataset slightly by discarding the transitions corresponding to trajectories that are not successful.

Table 3. RL architectures

Family	Actor	Critic
MLP	Linear(256)×3, Linear(A)	Linear(256) ×3, Linear(1)
Deep Set	ϕ: Linear(256) ×3 ρ: Linear(A)	ϕ: Linear(256) ×2 ρ: Linear(256), Linear(1)
Self Attention	SA: Linear(256), Block(256, 256, 4)×2 ρ: Linear(A)	SA: Linear(256), Block(256, 256, 4)×2 ρ: Linear(A)

We use the same policy architectures shown in Table 3 and optimize mean squared error loss over the dataset:

$$\arg\min_{\pi} J(\pi) := \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{s},\mathbf{a})\sim\mathcal{D}} ||\pi(\mathbf{s}) - \mathbf{a}||^2$$

We use the Adam (Kingma & Ba, 2014) optimizer with learning rate 0.001 (MLP, Deep Sets) or 0.0001 (Self Attention). Each policy is trained for $60,000$ gradient steps with a batch size of 128.

C.3. Training and inference speed

Here we consider the computational complexity of using different architecture classes (MLPs, Deep Sets, and Self Attention), as we scale the number of entities N. We consider the number of parameters, activation memory, and computation time (for a forward pass). For MLPs with fixed hidden layer sizes, the number of parameters and computation time increase linearly with N while the memory required for activations stays fixed (due to fixed hidden layer sizes). In Deep Sets and Self Attention, the number of parameters does not depend on the number of entities N. The activation memory and computation time grow linearly in Deep Sets, and quadratically for the pairwise interactions of Self Attention. In practice, the number of entities N is modest in all our environments (e.g., fewer than 10), but computational complexity may be relevant in more complex scenes with lots of entities.

For a more holistic real-world comparison of execution and training speed, Figure 3 shows both inference time and training time in the *N-Push* environments for $N \in \{1, 2, 3\}$. The inference time is the number of milliseconds it takes an actor do a single forward pass (using a GPU) on a single input observation. The Self Attention policy involves more complex computations and is significantly slower than Deep Set and MLP policies. The RL training time is the actual number of hours required to run the reinforcement learning algorithms of Figure 4, for each architecture. Not surprisingly, we see that *3-Push* takes significantly longer to train than *1-Push*, since it is a harder environment. For a fixed environment, however, all three architecture types are comparable in speed, with the Self Attention version being slightly slower than the others. The surprising similarity in RL training time (despite much slower inference time for the Self Attention policy) suggests that most of the RL time is spent on environment simulation rather than policy or critic execution. Hence, the difference between architectures presented in this paper has only a minor effect on reinforcement learning speeds in practice.

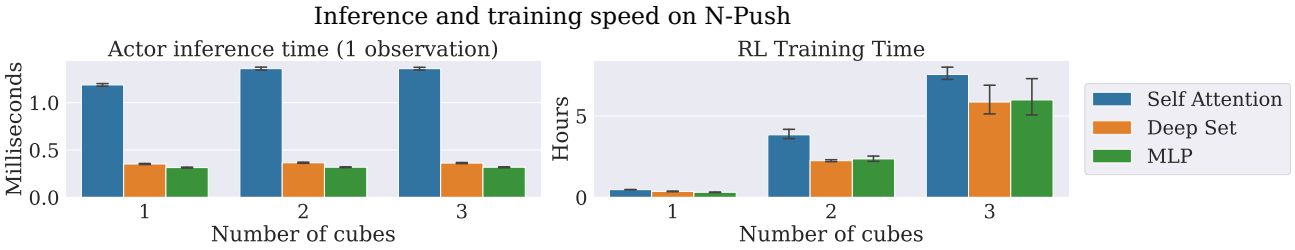

Figure 3. Left: the time (in milliseconds) it takes for each policy architecture to execute a single forward pass on a single input observation from the *N-Push* environments, where $N \in \{1, 2, 3\}$. The self attention policy is significantly slower, while the Deep Set and MLP policies are comparable. Right: Real world reinforcement learning times (in hours) training each policy/critic architecture on the *N-Push* environments. Although the Self Attention policy is slightly slower, all policies train at comparable speeds in the same environment. This suggests that environment simulation, not policy execution, is the dominant time consuming element.

D. Further results

D.1. Deep set architecture size

Recall that our Deep Set policy architecture involves two MLPs ϕ and ρ, where ϕ produces intermediate representations for each entity, those intermediate representations are summed, and then ρ produces the final output. In full generality, both ϕ and ρ may have two or more layers with nonlinearities in between. While our ϕ is a 3-layer MLP, we use a *linear* ρ throughout the main paper because we found that it often works comparably or better than using a larger 2-layer MLP ρ. Here we repeat the *N-Push* extrapolation and *Push + Stack* stitching experiments from the main paper using a 2-layer ρ, which we call "Deep Set (large)." The results from the main paper uses a 1-layer ρ which we refer to here as "Deep Set (small)."

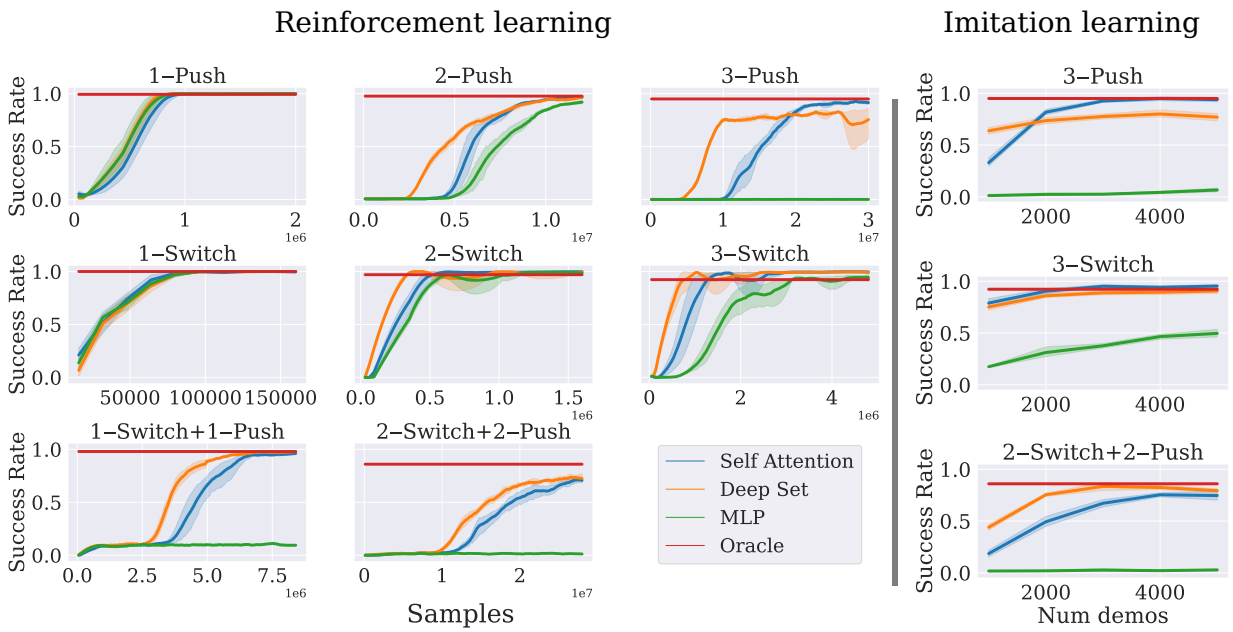

Figure 4. Training on environments of varying complexity using either reinforcement or imitation learning. Each row corresponds to a single environment family (*N-Push*, *N-Switch*, and *N-Switch + N-Push*), where environments with larger N contain more entities and are more complex. For RL (left), each plot is a training curve of success rate vs the number of steps taken in the environment. RL with standard MLPs can solve the simpler tasks such as *1-Push*, but Deep Set and Self Attention policies are superior on the more complex environments. For IL (right), we show success rates of behavior cloning against number of expert demonstrations in the dataset. Both Deep Set and Self Attention policies far outperform the MLP even when given less data. Shaded regions indicate 95% CIs over 5 seeds.

Figure 7 shows the results. In *N-Push*, the larger Deep Set model achieves higher training success rates in the 3-cube environment, but has worse extrapolation success rates for large numbers of cubes. For example, the smaller Deep Set model is significantly better at solving *6-Push*. Meanwhile, the large and small Deep Sets achieve very similar results in the pushing and stacking training environments. However, the larger Deep Set model achieves a higher success rate in the *Push + Stack* environment, indicating superior stitching capability. This suggests that simpler Deep Set architectures may be better for extrapolating to a large number of entities, but more complex architectures may be superior for solving complex tasks with a fixed number of entities.

D.2. N-Push with cube-cube collisions

As noted in Appendix B, we disable cube-cube collisions in the *N-Push* and *N-Switch+N-Push* experiments of the main paper (of course, the stacking settings require cube-cube collisions to be enabled). Here we repeat the *N-Push* extrapolation experiments with cube-cube collisions *enabled*. Figure 8 shows the results, which are qualitatively similar to when collisions are disabled. All methods observe a decrease in success rates of about 15%, with the Self Attention method often outperforming the Deep Set policy. This is likely because *N-Push* involves more interaction between entities once cube-cube collisions are enabled, and Self Attention's relational inductive biases are better suited for modeling these interactions.

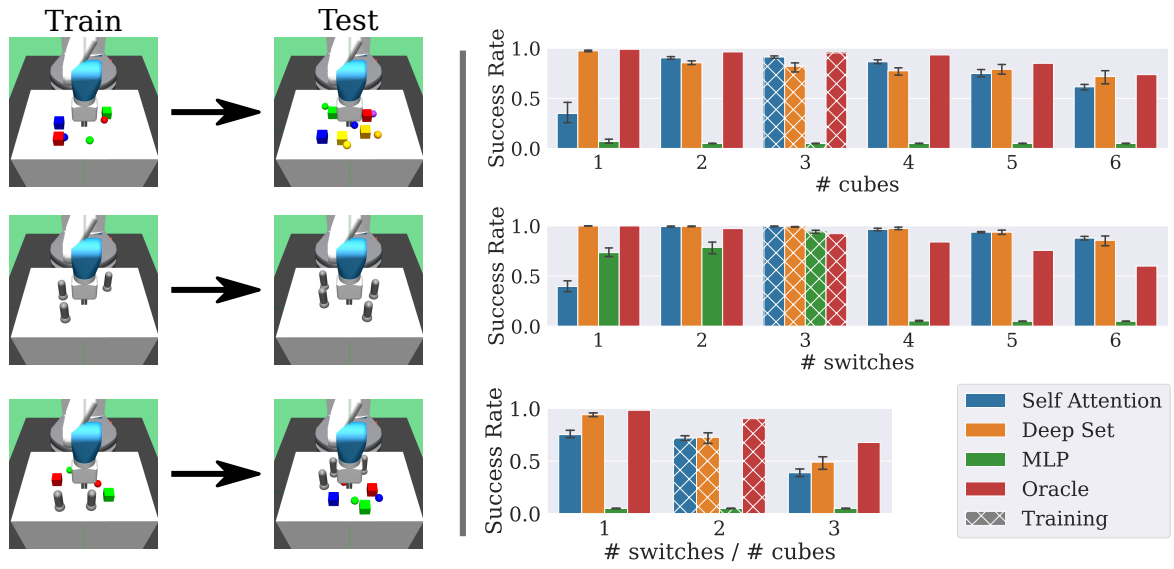

Figure 5. Extrapolation capabilities of RL-trained policies with different architectures. Each row is an environment family that contains environments with a varying number of entities. Policies are trained on a *single* environment from each family before being tested on all the others, with no additional training. Bar charts show success rates in each environment, with the hatched bars corresponding to training environments. The Self Attention and Deep Set policies extrapolate beyond the training environment to solve tasks with more or fewer entities than seen in training, while MLP policies struggle on more complex testing environments. Error bars are 95% CIs on 5 seeds.

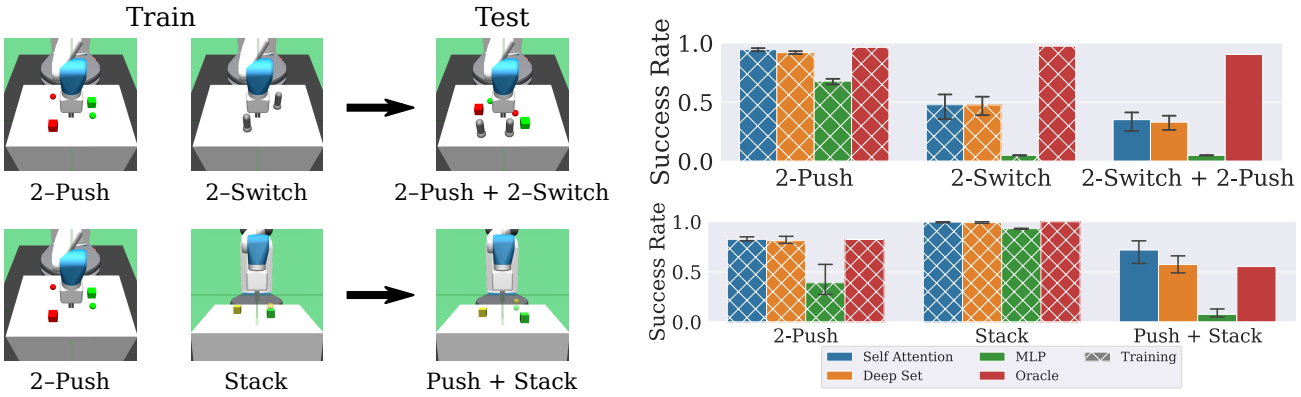

Figure 6. Left: each row corresponds to a setting where policies are evaluated on test tasks that require stitching together skills learned in training, with no additional data. In the top setting, a single agent is trained on *2-Switch* and *2-Push*, and must solve *2-Switch+2-Push* at test time. In the bottom setting, a single agent is trained on *2-Push* and *Stack*, and must solve *Push + Stack* at test time. Right: average success rates of policies with different architectures in each setting. Deep Set and Self Attention policies are moderate successful at solving the test tasks, and are comparable to the Oracle in *Push + Stack*. The MLP fails to achieve nontrivial success rates on both test environments. Error bars indicate 95% CIs over 5 seeds.

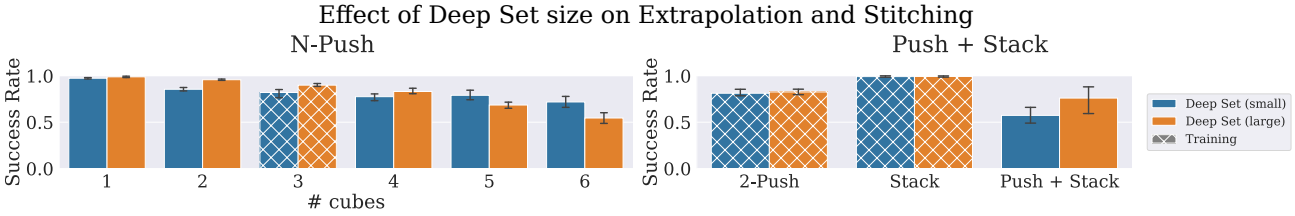

Figure 7. Comparison of *N-Push* extrapolation and *Push + Stack* stitching performance when using small and large variants of the Deep Set policy architecture. The small version implements ρ with a 1-layer linear map, while the large version implements ρ with a 2-layer MLP. For *N-Push*, the larger network achieves greater success rates in the training environment (3 cubes) but is actually worse when extrapolating to 5 or 6 cubes. On the other hand, the larger Deep Set displays superior stitching capability and achieves a higher average success rate when generalizing to *Push + Stack* from *2-Push* and *Stack*.

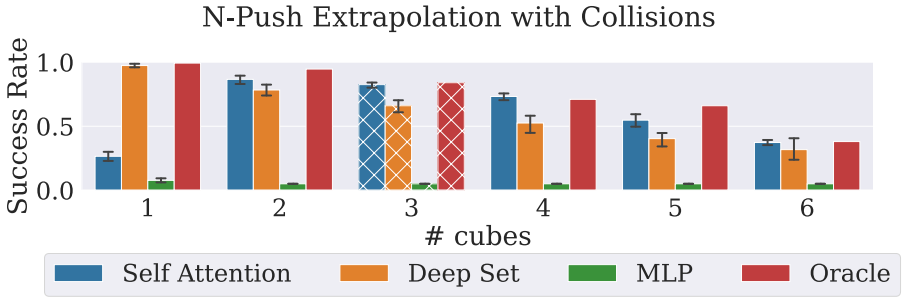

Figure 8. N-Push extrapolation with cube-cube collisions enabled. All methods observe some drop in performance relative to Figure 5, where *N-Push* has cube-cube collisions disabled. Self Attention tends to outperform Deep Sets when collisions enabled, likely because its relational inductive biases are better suited to handling interactions between entities that arise from collisions.