Learning to Manipulate Object Collections Using Grounded State Representations

Matthew Wilson
University of Utah
United States
`matthew.b.wilson@utah.edu`

Tucker Hermans
University of Utah & NVIDIA
United States
`thermans@cs.utah.edu`

Abstract: We propose a method for sim-to-real robot learning which exploits simulator state information in a way that scales to many objects. First, we train a pair of encoders on raw object pose targets to learn representations that accurately capture the state information of a multi-object environment. Second, we use these encoders in a reinforcement learning algorithm to train image-based policies capable of manipulating many objects. Our pair of encoders consists of one which consumes RGB images and is used in our policy network, and one which directly consumes a set of raw object poses and is used for reward calculation and value estimation. We evaluate our method on the task of pushing a collection of objects to desired tabletop regions. Compared to methods which rely only on images or use fixed-length state encodings, our method achieves higher success rates, performs well in the real world without fine tuning, and generalizes to different numbers and types of objects not seen during training. Video results: bit.ly/2khSKUs.

Keywords: sim-to-real, reinforcement learning, manipulation

1 Introduction

Humans regularly manipulate object groups as collections. When scooping up a bunch of grapes or sweeping a pile of coins into their hands, humans can track and manipulate the objects without needing to know, for example, how many nickels or dimes are present. Current robot systems lack such capabilities and most recent work has focused on picking or pushing objects one at a time [1, 2, 3, 4, 5, 6]. Learning to simultaneously manipulate many objects, on the other hand, is a greater challenge, as it requires a robot to track and reason about the many possible configurations and physical interactions between objects. To overcome this challenge, we use raw objct poses from a simulator to learn a latent space that captures multi-object state information; we use this learned representation to train an image-based policy that reasons about and manipulates variably-sized object collections as a whole.

Learning policies directly from RGB images stands as a popular approach for solving manipulation tasks, but as an approach it comes with several challenges. Namely, it is not obvious how to form task-relevant features or generate meaningful reward functions directly from high-dimensional sensory inputs. Since labeled data for manipulation requires significant time and effort to collect, researchers often leverage indirect self-supervised signals for training (e.g., [7, 8, 9]). Since such signals, by definition, do not directly correspond to task-relevant state information (such as poses and velocities of relevant objects), they can be prone to failure. For example, an autoencoder loss [7, 8] will incentivize a network to waste latent space capacity modeling more visually salient large objects and background in a scene, sometimes ignoring more fine-grained information relevant to the task.

Simulation-based training is a promising route to sidestep these difficulties. Researchers have shown that by exploiting raw state information to calculate rewards [10, 11, 12, 13, 14, 15, 16, 17, 18], construct grounded latent representations [10, 11, 19], and improve value function estimation [10, 11], they can greatly improve the stability and speed of policy training. However, all existing work rely on *fixed-length* vector inputs, limiting their effective use to settings with a known number of individual objects. Ours is the first work we are aware of in this space that is capable of handling a variable number of multiple objects.

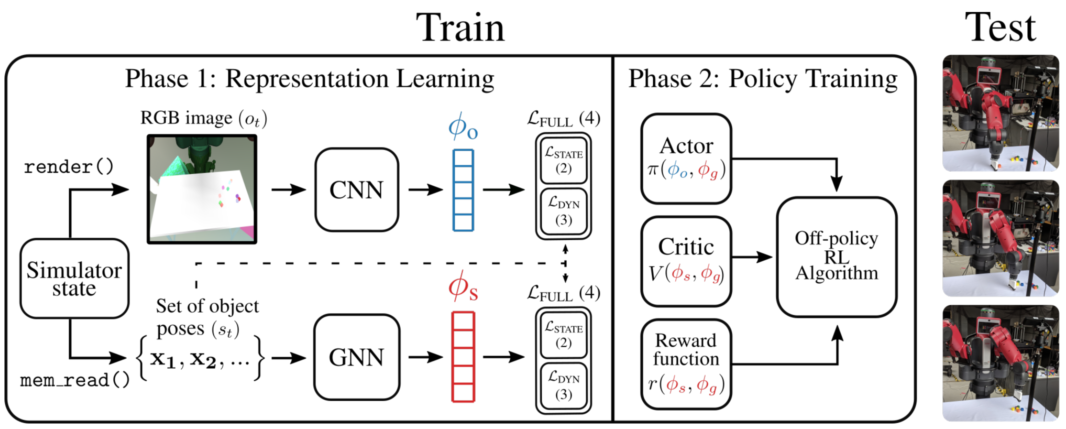
Train
Test
Phase 1: Representation Learning
Phase 2: Policy Training
RGB image (o_t)
render()
Simulator state
mem_read()
Set of object poses (s_t)
$\{\mathbf{x_1}, \mathbf{x_2}, \ldots\}$
CNN
GNN
ϕ_o
ϕ_s
$\mathcal{L}_{\text{FULL}}$ (4)
$\mathcal{L}_{\text{STATE}}$ (2)
$\mathcal{L}_{\text{DYN}}$ (3)
$\mathcal{L}_{\text{FULL}}$ (4)
$\mathcal{L}_{\text{STATE}}$ (2)
$\mathcal{L}_{\text{DYN}}$ (3)
Actor $\pi(\phi_o, \phi_g)$
Critic $V(\phi_s, \phi_g)$
Reward function $r(\phi_s, \phi_g)$
Off-policy RL Algorithm

Figure 1: Cartoon diagram of our approach. We first independently train two encoder networks, one convolutional neural network (CNN) and one graph neural network (GNN) using a multi-object state and dynamics loss function. Then, during our RL phase, we embed the observation: $o \xrightarrow{\text{CNN}} \phi_o$, state: $s \xrightarrow{\text{GNN}} \phi_s$, and goal: $g \xrightarrow{\text{GNN}} \phi_g$, and we use the embeddings in an asymmetric actor critic framework [10] to train a multi-object policy π.

Our primary contribution is an approach for exploiting simulator state information in a way that scales to multiple and variable object settings. In our approach, we train a pair of encoder networks to capture the state and dynamics information of a variable number of objects; we then use them to aid in learning multi-object manipulation policies. Our pair of encoders consists of one convolutional neural network (CNN) trained with RGB image inputs on sets of object pose targets and one graph neural network (GNN) [20] trained with sets of object poses as both its inputs and targets (i.e., it learns to autoencode the state). Since the CNN operates directly on RGB images, it gives us the ability to learn a policy that can be deployed in the real world. Since the GNN is doubly grounded in simulator state information (via input and output), it provides a more accurate and stable representation, so we use it both for calculating rewards and feeding to the value network in our actor-critic RL algorithm. (The idea of providing different inputs to the policy and value networks in this way is known as an asymmetric actor critic [10].) Fig. 1 provides an overview of our network structure and full approach.

To evaluate our approach, we construct a multi-object manipulation task (Fig. 2) which is challenging to learn from RGB images alone and ill-suited for prior fixed-state representation learning approaches. We consider a collection of 1-20 homogeneously shaped objects that start in arbitrary configurations on a tabletop, which the robot must push into desired goal configurations. We train on a simulated version of this task but we learn a policy that can be deployed in the real-world without any fine tuning. We conduct extensive comparison and ablation studies of our trained policy in both simulation and on the physical robot. We find that our approach achieves higher success rates than an autoencoder or a fixed-state training approach, while also generalizing well to configurations not seen during training.

2 Related Work

We draw on and extend two main lines of robot learning research: self-supervised representation learning and sim-to-real learning.

Self-Supervised Representation Learning: A common pattern in robot learning methods is to pre-train a representation using a self-supervised signal and then use that representation for downstream task planning or task learning [7, 8, 9, 17, 21, 22, 23, 24, 25, 26, 27, 28]. The pattern usually goes as follows: researchers use an exploration policy (often scripted [8, 9, 21, 22, 23]) to collect a dataset of real world interactions, consisting of either single frames [7, 8], state action transitions [9, 21, 25], or long sequences of frames and actions [22, 23]. They then use this dataset to train a model via a self-supervised signal, such as an autoencoder loss [7, 8], GAN loss [17, 28], or some related pre-text task [9, 21, 25, 26, 29, 30]. Once they learn a good representation, they often use it in either a model-based/planning framework [9, 21, 22, 23, 25, 27, 30] or in a model-free RL framework [7, 8] to learn a task policy. In this work, we follow a model-free RL approach. Compared to prior work, we learn representations which are *grounded*, both in their inputs and outputs. Unlike most prior work, where representations are trained on image-based predictions, we directly train our representations to capture the state of the objects (through Eq. 2). We also directly consume ground truth information via a GNN [20]. Our experiments suggest that these components speed up and stabilize policy training and improve the quality and generalizability of trained models.

Sim-to-Real Learning: Sim-to-real methods—those trained primarily or exclusively in physics simulators and targeted to work in the real world—are starting to yield impressive results on physical robotics tasks [10, 11, 12, 13, 14, 15, 16, 17, 18, 28, 31, 32]. Simulation training offers several advantages over real-world training, including increased speed and parallelizability of data collection, safety of exploration, and full observability and control over environmental factors. However, there is often a large gap between the crude simulator phenomena and their real world analogs which causes naively-trained models to fail in the real world. Researchers have developed several methods to help enable policies to transfer across this gap, including using GANs to map from simulation visuals to the real world [28] or vice versa [17], and incorporating learned models from the real world into the simulation loop [14]. One simple and effective approach we use, known as domain randomization [32], is to randomize the physical and visual properties of an environment during training. This variability makes trained policies more robust and less prone to overfitting to simulator characteristics [10, 12, 15, 31, 32]. In this work, we add another tool to the sim-to-real toolbox. This tool speeds up training in multi-object settings, scales to a variable number of objects, is symbiotic with existing methods, and is extensible to more complex tasks through its flexible state encoding.

Graph Neural Networks in RL: GNNs are becoming popular in control and decision making after their strong performance in natural language processing [33] and visual-spatiability tasks [34]. DeepMind researchers use GNNs in RL to play the Star Craft II video game [35, 36]. Ajay et al. [37] use a GNN for robotic manipulation to create a hybrid analytic and data-driven simulator for task planning; however, they rely on precise object knowledge and motion capture in the real world. Our work is the first we know of to use GNNs to learn state representations for robotic manipulation tasks.

3 Preliminaries

Goal-conditioned RL: We use the standard multi-goal or goal-conditioned reinforcement learning formalism, as in Schaul et al. [38]. We describe an environment by states $s \in \mathcal{S}$, goals $g \in \mathcal{G}$, actions $a \in \mathcal{A}$, a distribution for sampling initial state-goal pairs $p(s_0, g)$, and a goal-conditioned reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$. To account for partial observability, we denote images as observations $o \in \mathcal{O}$, and images of goals as $g^o \in \mathcal{O}$. The objective of the agent is to learn a goal-conditioned policy $a_t = \pi(o_t, g)$ which maximizes the expected discounted return $\mathbb{E}[R_t] = \mathbb{E}[\sum_{i=t}^{\infty} \gamma^{i-t} r_i]$.

Task Overview: The task we evaluate on requires an agent to push a collection of 1-20 homogeneous objects to desired areas on a tabletop. The agent's state is factored into a set of 2D coordinates of object centers $\{\mathbf{x}_1, \mathbf{x}_2, ...\}$. Similar to prior work [9], we use pushing actions which are parameterized by 4 continuous values: a 2D starting pose (x, y, θ) and a pushing distance d. These values are constrained to the reachable table workspace of our physical Baxter robot [39]. For data collection we use a simple scripted policy, which randomly samples an object to push for a random distance in a random direction.

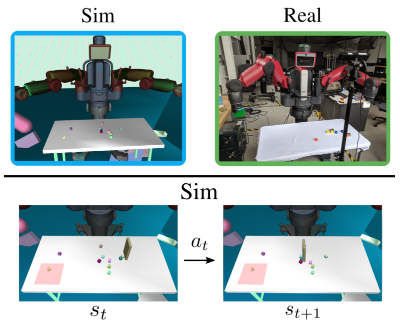

Figure 2: **Top:** Simulation and real world environments. **Bottom:** An example of a state-action transition in simulation showing a trained policy pushing towards a goal area, which is marked by a red square.

Graph Neural Networks: We use a simplified version of the graph neural network framework introduced by Battaglia et al. [20]. We define graphs by a 3-tuple $G = (V, E, \mathbf{u})$, where $V = \{\mathbf{v}_i\}$ is the set of nodes with each $\mathbf{v}_i$ a vector representing the 2D pose of an object; $E = \{(\mathbf{v}_r, \mathbf{v}_s)\}$ is an edge between a pair of receiver, $\mathbf{v}_r$, and sender, $\mathbf{v}_s$, nodes; and $\mathbf{u}$ is a global attribute vector which we compute to aggregrate the full graph information. GNNs have been used in a variety of domains [34, 37] and they vary widely in how they operate on graph structures [20]. However, they all use some variation of two key functions: *updates* and *aggregations* [20]. Updates run computations on individual nodes or edges (e.g., linear layer forward pass on each node). Aggregrations perform reductions across graph structures (e.g., elementwise summation over the set of all nodes). See Battaglia et al. [20] for a more in-depth introduction to GNNs.

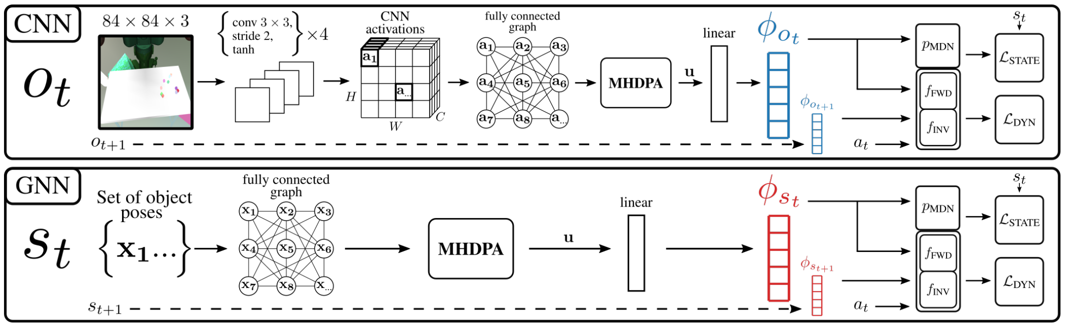

Figure 3: CNN and GNN architectures and losses for self-supervised representation learning

4 Learning to Manipulate Object Collections

To train an image-based policy capable of tracking and reasoning about the many possible configurations and interactions in multi-object tasks, we develop a method to exploit variably-sized state information from a physics simulator.

Our approach, illustrated in Fig. 1, consists of two phases: a supervised representation learning phase and a reinforcement learning (RL) policy training phase. The interesting details of our apporach lay mostly in how we train our representations to capture variably-sized information and how we incorporate them in the RL phase to aid in policy training. Subsection 4.1 covers the supervised learning phase and our solutions to the technical challenges of consuming and predicting variably-sized object state information (Fig. 3). Subsection 4.2 covers the RL phase and how we calculate rewards and incorporate our learned representations into a full policy-training algorithm.

4.1 Learning Grounded State Representations

We seek to learn representations which capture the variably-sized and order-invariant *set* of objects in the environment state s. Formally, we seek to learn two encoders $\mathbf{E}_s(s)$ and $\mathbf{E}_o(o)$ that map from their respective modalities to latent vector spaces ϕ_s and ϕ_o which each represent the relevant information of the variably-sized state. This objective introduces challenges both on encoding the inputs and decoding the outputs of the neural networks. We discuss these challenges and our solutions below.

The Input Side - Encoding Raw State: If our state, s, consisted of a single object pose $\mathbf{x}$, we could simply learn our state encoder $\mathbf{E}_s$ via a multi-layer perceptron, $\phi = \mathbf{MLP}(\mathbf{x})$. To scale this approach to many objects, we could increase the input size of the MLP and set unused values to zero when there are fewer than the maximum number of objects present. Unfortunately, in practice, the weak inductive bias of this approach makes it poorly suited for moderately large numbers of objects. With a set of n objects, there are $n!$ ways that these objects can be arranged: $(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)$, $(\mathbf{x}_n, \mathbf{x}_{n-1}, ..., \mathbf{x}_1)$, etc. Each of these will appear unique to the MLP, requiring an exponential order of training examples for the network to learn in the worst case [20]. To scale to many objects, we instead use a graph neural network (GNN) that operates on set structures in an order-invariant way.

The specific graph network model we use is a single encoder block from the transformer model [33]. One of the transformer's primary mechanisms, the multi-head dot-product attention (MHDPA) can be interpretted as implementing a special parallelized version of updates and aggregations over a fully connected graph [20]. (Others have demonstrated the use of MHDPA on reinforcement learning tasks like Star Craft II [35, 36].) We convert our state into a fully connected graph G associating a vertex, $\mathbf{v}_i$, with each object centroid, $\mathbf{x}_i$. We use the **MHDPA** operation to compute an updated graph of the same shape, $G' = \mathbf{MHDPA}(G)$. To aggregate information from the nodes $\mathbf{v}'_i$ of G', we compute a gated activation [40] on each node $\mathbf{v}'_i$ followed by an elementwise summation over the set of all nodes. This yields a single vector that summarizes the full graph information: $\mathbf{u} = \sum_{i=1}^{n} \tanh(W_1 \mathbf{v}'_i) \odot \sigma(W_2 \mathbf{v}'_i)$, where $\odot$ denotes element-wise multiplication. A simpler aggregation (without the gated activation): $\mathbf{u} = \sum_{i=1}^{n} \mathbf{v}'_i$ also works, just not as well. We then linearly transform $\mathbf{u}$ to achieve the desired vector size of our latent space: $\phi = W_3 \mathbf{u}$.

The Output Side - Learning State Representations: We face similar concerns in constructing an appropriate loss function for learning to predict a variable number of outputs. For the single object or fixed-length state setting, we could use mean-squared error regression from the latent space ϕ to

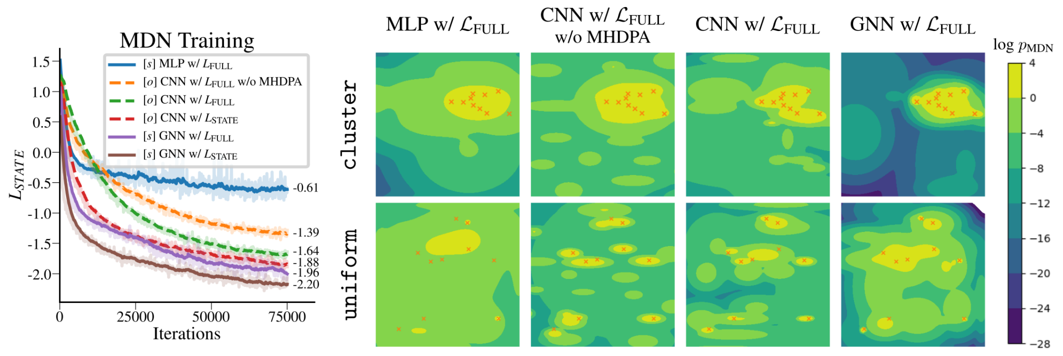

Figure 4: **Left:** MDN learning curves for various networks trained with either $\mathcal{L}_{\text{FULL}}$ or $\mathcal{L}_{\text{STATE}}$. **Right:** Visualizations of the trained MDNs; orange crosses represent ground truth locations. All networks capture the state of clustered objects well, but the MLP fails to precisely localize the individual objects. Incorporating MHDPA into the CNN leads to fewer false positives and greater precision. And the GNN produces the tightest bounds and places the greatest amount of probability mass correctly on the objects and not other points.

that object pose $\mathbf{x}$: $\frac{1}{2}\|g(\phi) - \mathbf{x}\|_2^2$ [10, 11, 19]. This approach, however, is prone to failure in the multi-object setting. For our image-based encoder network, $\mathbf{E}_o$, it would be ambiguous which labels $(\mathbf{x}_1, \mathbf{x}_2, ...)$ belong to which of several visually indentical objects in the state; in this case, the network has no way of knowing which object predictions it should place in which of its arbitrarily assigned prediction slots $(\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, ...)$.

To sidestep this and other, similar issues, instead of producing individual predictions, we construct a mixture density network (MDN) [41], $p_{\text{MDN}}(\mathbf{p}|\phi)$, that encodes the probability of a 2D point $\mathbf{p}$ on the table corresponding to one of the object centers $\mathbf{x}_i$ conditioned on our latent representation ϕ:

$$p_{\text{MDN}}(\mathbf{p}|\phi) = \sum_{k=1}^{K} \alpha(\phi)_k \, \mathcal{N}\Big(\mathbf{p}\Big|\mu(\phi)_k,\ \Sigma(\phi)_k\Big) \tag{1}$$

The MDN is a Gaussian mixture model, with weights α_k, means μ_k, and variances Σ_k, computed as learned linear transformations of the latent space. (See Fig. 4 for visualizations of MDN predictions.) We train p_{MDN} using a maximum likelihood loss which optimizes the network to assign high probability to locations of ground truth object centers $(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)$: [1]

$$\mathcal{L}_{\text{STATE}} = -\frac{1}{n}\sum_{i=1}^{n} \log p_{\text{MDN}}(\mathbf{p} = \mathbf{x}_i|\phi) \tag{2}$$

While $\mathcal{L}_{\text{STATE}}$ enables our network to learn a latent representation that can accurately capture object position information, it does not account for dynamics information. Some state encodings may be close according to $\mathcal{L}_{\text{STATE}}$, but difficult to traverse between, based on the actions available to an agent. To account for state traversability and actionability, we train our latent space using forward and inverse dynamics functions, as in prior work (e.g. [9, 42]):

$$\mathcal{L}_{\text{DYN}} = \mathcal{L}_{\text{FWD}} + \mathcal{L}_{\text{INV}} = \frac{1}{2}\left\|f_{\text{FWD}}(\phi_t, a_t) - \phi_{t+1}\right\|_2^2 + \frac{1}{2}\left\|f_{\text{INV}}(\phi_t, \phi_{t+1}) - a_t\right\|_2^2 \tag{3}$$

Here f_{FWD} is a small MLP trained to predict the forward dynamics in the latent space; while f_{INV} is a small MLP trained to predict the inverse dynamics—the necessary action to move between two consecutive latent-space states. We use $\mathcal{L}_{\text{FULL}}$ to train our encoders:

$$\mathcal{L}_{\text{FULL}} = \mathcal{L}_{\text{STATE}} + \mathcal{L}_{\text{DYN}} \tag{4}$$

We hypothesized that the addition of $\mathcal{L}_{\text{DYN}}$ would help regularize the latent space and condition it for use in deciding actions, leading to better performance in the real world. Our results seem to weakly support this hypothesis, with an $\mathcal{L}_{\text{STATE}}$ model performing slightly (but not significantly) better in simulation and an $\mathcal{L}_{\text{FULL}}$ model generalizing slightly (but not significantly) better to the real world. Figure 4 summarizes the training results for our encoder networks.

[1] We note that our MDN loss is specific to tasks where object types can be ignored. One could instead use an object-type-sensitive loss or perhaps one which assigns probability density to space contained within objects.

Architecture Summary: To summarize, we learn two separate encoders: $\mathbf{E}_{\text{GNN}} : s \to \phi_s$ which maps from state, and $\mathbf{E}_{\text{CNN}} : o \to \phi_o$ which maps from image observations. Each encoder learns separate forward and inverse dynamics functions as well as separate MDN parameters.[2] Fig. 3 shows the full architecture. We incorporate an **MHDPA** operation into the CNN as in prior work [35].

As illustrated in Fig. 3, we treat each of the ($1 \times 1 \times C$ sized) slices of the last ($W \times H \times C$ sized) CNN activation as elements of a set (of cardinality $N^{W \times H}$), and compute the same **MHDPA** and aggregation operations we use in the GNN. Incorporating GNN mechanisms into CNN architectures this way has been shown to avoid some shortcomings of vanilla CNN models in tasks that require tracking object counts and reasoning about relative positions of objects [34], and we also find it helps.

4.2 RL Training

During the RL phase, we use our grounded state representations to train an image-based policy via an off-policy actor critic algorithm (Soft Actor Critic (SAC) [43] in this case). We use the doubly-grounded state-based representation ϕ_s to both compute rewards and train the critic efficiently. We apply domain randomization and use the image-based representation ϕ_o to train the actor. Our full approach, which we call Multi-Object Asymmetric Actor Critic (MAAC), is described in Algorithm 1.

Algorithm 1 MAAC: Learning manipulation tasks using a Multi-Object Asymmetric Actor Critic

```
Require: Encoders $\mathbf{E}_{\text{CNN}}(o_t)$, $\mathbf{E}_{\text{GNN}}(s_t)$, replay buffer $R$, SAC [43] or other off-policy RL algorithm
 1: Collect dataset $\mathcal{D}$ in simulation using scripted policy, while applying domain randomization to images
 2: Train encoders $\mathbf{E}_{\text{CNN}}$ and $\mathbf{E}_{\text{GNN}}$ on $\mathcal{D}$ by optimizing (4)
 3: for simulation episode $e = 1, \dots, M$ do
 4:     Sample and embed initial state $\phi_{s_0} = \mathbf{E}_{\text{GNN}}(s_0)$, observation $\phi_{o_0} = \mathbf{E}_{\text{CNN}}(o_0)$, and goal $\phi_g = \mathbf{E}_{\text{GNN}}(g)$
 5:     for $t = 0, \dots, T$ do
 6:         Execute action $a_t = \pi(\phi_{o_t}, \phi_g)$, and obtain new state $s_{t+1}$ and observation $o_{t+1}$
 7:         Embed new state and observation $\phi_{s_{t+1}} = \mathbf{E}_{\text{GNN}}(s_{t+1})$, $\phi_{o_{t+1}} = \mathbf{E}_{\text{CNN}}(o_{t+1})$
 8:         Compute reward $r_t = r(\phi_{s_t}, \phi_g)$, and store transition $(\phi_{s_t}, \phi_{o_t}, \phi_g, a_t, r_t, \phi_{s_{t+1}}, \phi_{o_{t+1}})$ in $R$
 9:     end for
10:     Generate virtual goals $(g'_1, g'_2, ...)$ and rewards $(r'_1, r'_2, ...)$ for each step $t$ and store in $R$ (w/ HER [44])
11:     Optimize actor $(\pi(\phi_{o_t}, \phi_g))$ using SAC with $o_t$ embeddings
12:     Optimize critic $(V(\phi_{s_t}, \phi_g), Q_1(\phi_{s_t}, \phi_g, a), Q_2(\phi_{s_t}, \phi_g, a))$ using SAC with $s_t$ embeddings
13: end for
```

Algorithm 1 Overview: First, before the RL phase, we run a scripted policy to collect a dataset $\mathcal{D}$ of transitions of states and observations $(s_t, o_t, a_t, s_{t+1}, o_{t+1})$ (line 1). We use $\mathcal{D}$ to train our encoders via supervised learning (line 2). Then, for each episode, we first embed the goal and initial observation and state (line 4). At each step, we pass the observation embedding ϕ_{o_t} through the policy network to get an action (line 6). We also use the state embedding ϕ_{s_t} (along with ϕ_g) to compute a reward (line 8), and later to feed to the value network. We parameterize the policy network $\pi(\phi_{\mathbf{o_t}}, \phi_g)$ and the value networks $(V(\phi_{\mathbf{s_t}}, \phi_g), Q_1(\phi_{\mathbf{s_t}}, \phi_g, a), Q_2(\phi_{\mathbf{s_t}}, \phi_g, a))$ by MLPs with two hidden layers with the learned embeddings as input; we optimize them using standard SAC losses.

Specifying a Reward: To specify a reward for multi-object tasks, we face similar considerations as when embedding a variably sized state (Section 4.1). Our approach is to use the GNN-based latent space vector to specify a distance between the current state and goal state. To do this, we embed the current and goal states $\phi_{s_t} = \mathbf{E}_{\text{GNN}}(s_t)$ and $\phi_g = \mathbf{E}_{\text{GNN}}(g)$. Then we specify a goal success condition $f_g(s_t)$ by computing the cosine distance between these embeddings; when the cosine distance falls below a threshold ϵ, the goal is considered met. This can be described by the following equation: $f_g(s_t) = \texttt{cos\_dist}(\phi_{s_t}, \phi_g) < \epsilon$. For every step t that the agent has not reached the goal, we provide a -1 reward. Once it reaches the goal we provide a $+1$ reward. [3] We terminate episodes with a final -1 reward if the goal condition has not been met after T steps (where $T \approx 50$).

Tricks to Speed Up Training: We keep the encoder networks frozen in this phase and cache the latent embeddings, increasing the maximum batch size and size of replay buffer we can keep

[2]We first tried learning ϕ_s and setting it as a regression target for our CNN encoder, but this failed to produce useful estimates. We also tried sharing the output dynamics and density weights, but this performed worse.

[3] We originally tried using a denser reward based on changes in cosine distance to the goal, but found it did not work as well; we think likely because it penalized incorrect movement too much, hindering exploration [44].

in memory and reducing the computation required to train the SAC heads. To incentivize object movement, we add an extra reward of $+0.1$ when the agent moves any of the objects more than δ cm (where $\delta \approx 2.0$). To improve exploration, we usually sample actions from our learned policy $a_t = \pi(\phi_{o_t}, \phi_g)$ (line 6), but with probability p, we sample actions from our scripted policy. We set p to 1.0 early in training and linearly anneal it over time. To help learn from our sparse reward, we use Hindsight Experience Replay (HER) [44] to augment the replay buffer (line 10). Also, our actual implementation is a parallelized version of Algorithm 1, where we intermix SAC optimization cycles with experience collection. [4] See the supplementary material for these implementation details.

5 Experimental Results

In our experiments we study the following questions:

1. How does the full method compare with alternative training formulations in simulation? (Sec. 5.1)
2. What is the effect of domain randomization and using ϕ_s in policy training? (Sec. 5.2)
3. How do the learned policies compare and generalize in real-world experiments? (Sec. 5.3)

We train our policies purely in simulation and evaluate them in simulation and the real world. We use two methods for sampling states; `uniform`: where all objects are uniform randomly distributed across the table; and `cluster`: where objects are sampled from 25cm x 25cm square areas on the table. For initial state sampling $p(s_0)$, we use `uniform` and `cluster` each 50% of the time. For goal sampling, we always use the `cluster` approach. We train with 10 cubes, but evaluate on various objects in the real world. We compare our full method to several methods and ablations, including an MLP and autoencoder, an image-based goal ($\pi(\phi_s, \phi_{g^o})$ vs. $\pi(\phi_s, \phi_g)$), a CNN that does not incorporate the MHDPA mechanism, and models trained with only $\mathcal{L}_{\text{STATE}}$ or $\mathcal{L}_{\text{DYN}}$ rather than $\mathcal{L}_{\text{FULL}}$.

Unless otherwise specified, we always use domain randomization and an asymmetric actor critic (AAC) approach—using ϕ_s for both computing rewards and value functions. ϕ_s and ϕ_o are always vectors of size 128. We train all models on $\mathcal{L}_{\text{FULL}}$ (or some ablation of it), besides the autoencoder approach, which uses a reconstruction loss. [5] Recall that during RL training, we determine goal success as $f_g(s_t) = \texttt{cos\_dist}(\phi_{s_t}, \phi_g) < \epsilon$. If ϵ is too high, the task is trivially easy, and if it is too low, the agent will never reach the goal. To ensure ϵ corresponds to approximately equal state similarities across approaches, we collect a set of transitions, compute a list of cosine distances for each model, and apply a scaling based on the relative values that the models produce.

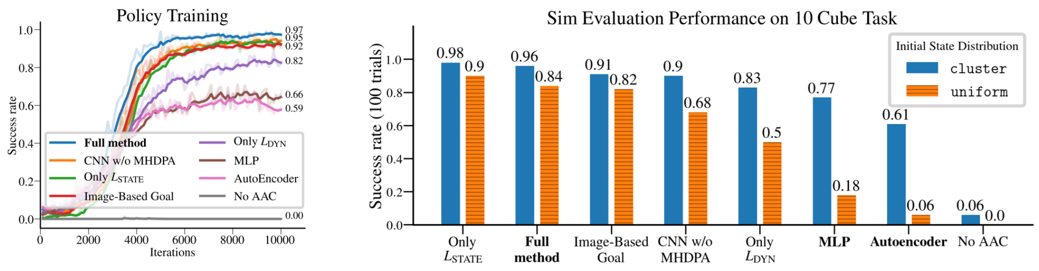

Figure 5: **Left:** Sim training. Our full method reaches a higher success rate in a fewer number of training iterations compared to an MLP approach, an autoencoder approach, and several ablations. **Right:** Evaluation results. We evaluate all trained models on a slight variation of the task that does not rely on thresholds in a learned embedding space. We find that the full method produces nearly the best performance, only being slightly beaten out by a model trained on $\mathcal{L}_{\text{STATE}}$. We test models using both a clustered and uniform initial state distribution of objects, and find that other methods perform worse, especially with objects initially uniformly distributed.

5.1 Simulation Eval Performance

To evaluate our training models in simulation, we run 100 trials of each method and count the success rate of when they are able to push the cubes into the desired regions. We cannot rely on the learned ϕ space to fairly compare model performance, as some models could be more lenient of state similarities. Instead we iterate through the positions of all the objects and check if they fall within the 25cm x

[4]We additionally tried directly learning a CNN policy $\pi(O, \phi_g)$ without pre-training a latent representation from images, but found this to train much slower during RL as we could not use the cached embeddings.

[5] We found that a naive autoencoder approach failed to train on this task when applying domain randomization. We had to apply a semi-novel technique, based on prior canonicalization work [17], to get it to work at all.

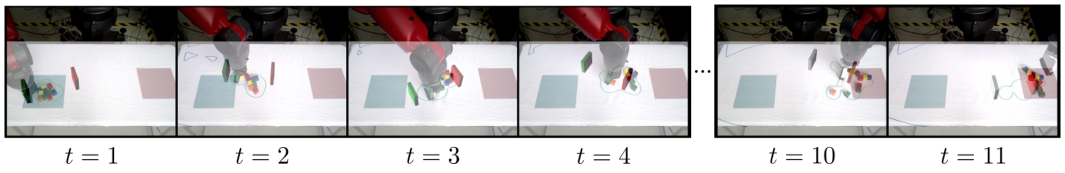

Figure 6: Sequence of states and pushing actions chosen by a policy trained with our full approach leading to successful execution. The blue and red square overlays represent the initial state and goal areas sampled with the `cluster` state distribution. Contours represent the MDN output density estimates of object locations.

25cm goal area. (See more detail in the supplementary material.) The `uniform` distribution (which is sparse across the table) better represents true multi-object handling, whereas a cluster of objects is easier and more like a single-object setting. The results shown in Fig. 5 illustrate that the MDN and full method best handle this more challenging setting.

5.2 Effects of Domain Randomization and Using ϕ_s

We found that applying domain randomization (DomRand) made the ϕ_o estimates noisy and highly variable, even when the underlying state did not change. This ultimately led to our use of ϕ_s in an asymmetric actor critic approach (AAC) to stabilize training signals. To study these effects, we ran our full approach with different combinations of AAC and DomRand. When applying DomRand, we find the use of AAC makes the difference between the policy achieving high success rates and completely failing to learn. AAC seems to help primarily by stabilizing learning signals.

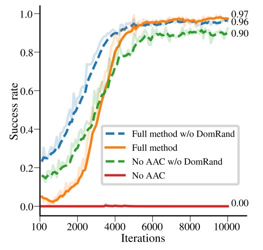

Figure 7: Effects of domain randomization and asymmetric actor critic.

5.3 Real World Evaluation Performance

We use a similar approach as we did in simulation for evaluating our policies in the real world, but we always use `cluster` as our initial state distribution. To study which loss function generalizes best, we compare the full method, $\mathcal{L}_{\text{DYN}}$, $\mathcal{L}_{\text{STATE}}$, and the autoencoder as these all learn different image models. Along with the 10 cube task we train on, we test on tasks with 1 cube; 20 cubes; 10 spoons, knives, and forks (Silverware); and 5 crumpled paper balls. Fig. 6 shows a successful sequence of states and pushing actions taken by the robot using the trained policy. Fig. 8 shows quantitative results. We see that all variants of our approach significantly outperform the autoencoder, and $\mathcal{L}_{\text{FULL}}$ slightly outperforms the ablations in these trials. See the associated video for more experiment visualizations.

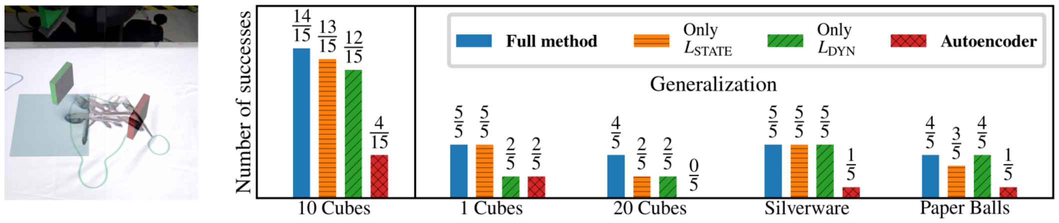

Figure 8: **Left:** Silverware task. **Right:** Real world evaluation results. We evaluate on 15 different start states and goal locations for the 10 Cube training task, and on 5 for the other generalization tasks. We terminate after successes or 15 push actions. We hold start states and goals constant across different methods. The full method produces the greatest performance and generalizability, especially outperforming the autoencoder approach.

6 Discussion

We present an approach for sim-to-real robot manipulation learning that exploits variable and multi-object simulator state. We do this by learning grounded state representations and using them to train an RL policy that outperforms alternative approaches in both simulated and real-world experiments.

Limitations and Future Work: We use a coarse goal specification of 25cm x 25cm areas. It would be interesting to try to achieve more precise goal states, perhaps by gradually annealing the threshold ϵ during training. Our scripted policy is specific to the pushing task we consider, and the MDN loss ($\mathcal{L}_{\text{STATE}}$) is specific to tasks where objects can be considered all of the same type. Exploration-based policies (e.g., [42]) and set-based loss functions (e.g., [45]) could perhaps be used instead. We tried `uniform` initial states in the real world, but found our policy often got stuck. We believe incorporating memory (e.g., with recurrent neural networks) may lead to more adaptive and robust policies.

Acknowledgments

Matthew Wilson was supported in part by the National Science Foundation under Grant #1657596, as well as the Undergraduate Research Opportunities Program at the University of Utah.

References

[1] R. Tella, J. Birk, and R. Kelley. General purpose hands for bin-picking robots. *IEEE Transactions on Systems, Man and Cybernetics*, 12(6):828–837, 1982.

[2] M. Mason and J. Salisbury Jr. *Robot hands and the mechanics of manipulation*. The MIT Press, Cambridge, MA, 1985.

[3] D. Berenson and S. Srinivasa. Grasp Synthesis in Cluttered Environments for Dexterous Hands. In *Humanoids*, 2008.

[4] A. Cosgun, T. Hermans, V. Emeli, and M. Stilman. Push Planning for Object Placement on Cluttered Table Surfaces. In *IROS*, 2011.

[5] T. Hermans, J. M. Rehg, and A. Bobick. Guided Pushing for Object Singulation. In *IROS*, 2012.

[6] J. Mahler, M. Matl, V. Satish, M. Danielczuk, B. DeRose, S. McKinley, and K. Goldberg. Learning ambidextrous robot grasping policies. *Science Robotics*, 4(26), 2019.

[7] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel. Deep spatial autoencoders for visuomotor learning. In *ICRA*, 2016.

[8] A. V. Nair, V. Pong, M. Dalal, S. Bahl, S. Lin, and S. Levine. Visual reinforcement learning with imagined goals. In *NeurIPS*, 2018.

[9] P. Agrawal, A. V. Nair, P. Abbeel, J. Malik, and S. Levine. Learning to Poke by Poking: Experiential Learning of Intuitive Physics. In *NeurIPS*, 2016.

[10] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel. Asymmetric actor critic for image-based robot learning. *arXiv preprint arXiv:1710.06542*, 2017.

[11] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba. Learning dexterous in-hand manipulation. *CoRR*, 2018.

[12] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *ICRA*, 2018.

[13] Z. Xie, P. Clary, J. Dao, P. Morais, J. Hurst, and M. van de Panne. Iterative reinforcement learning based design of dynamic locomotion skills for cassie. *arXiv preprint arXiv:1903.09537*, 2019.

[14] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 2019.

[15] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

[16] J. Matas, S. James, and A. J. Davison. Sim-to-real reinforcement learning for deformable object manipulation. In *CoRL*, 2018.

[17] S. James, P. Wohlhart, M. Kalakrishnan, D. Kalashnikov, A. Irpan, J. Ibarz, S. Levine, R. Hadsell, and K. Bousmalis. Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks. *arXiv preprint arXiv:1812.07252*, 2018.

[18] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. D. Ratliff, and D. Fox. Closing the Sim-to-Real Loop: Adapting Simulation Randomization with Real World Experience. In *ICRA*, 2019.

[19] F. Zhang, J. Leitner, M. Milford, and P. Corke. Modular deep q networks for sim-to-real transfer of visuo-motor policies. *arXiv preprint arXiv:1610.06781*, 2016.

[20] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[21] A. Nair, D. Chen, P. Agrawal, P. Isola, P. Abbeel, J. Malik, and S. Levine. Combining Self-Supervised Learning and Imitation for Vision-Based Rope Manipulation. In *ICRA*, 2017.

[22] F. Ebert, C. Finn, A. X. Lee, and S. Levine. Self-supervised visual planning with temporal skip connections. In *CoRL*, 2017.

[23] F. Ebert, S. Dasari, A. X. Lee, S. Levine, and C. Finn. Robustness via retrying: Closed-loop robotic manipulation with self-supervised learning. In *CoRL*, 2018.

[24] L. Pinto and A. Gupta. Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours. In *ICRA*, 2016.

[25] M. Zhang, S. Vikram, L. Smith, P. Abbeel, M. J. Johnson, and S. Levine. Solar: Deep structured latent representations for model-based reinforcement learning. *arXiv preprint arXiv:1808.09105*, 2018.

[26] M. A. Lee, Y. Zhu, K. Srinivasan, P. Shah, S. Savarese, L. Fei-Fei, A. Garg, and J. Bohg. Making Sense of Vision and Touch: Self-Supervised Learning of Multimodal Representations for Contact-Rich Tasks. In *ICRA*, 2019.

[27] G. Sutanto, N. D. Ratliff, B. Sundaralingam, Y. Chebotar, Z. Su, A. Handa, and D. Fox. Learning Latent Space Dynamics for Tactile Servoing. In *ICRA*, 2019.

[28] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, et al. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. In *ICRA*, 2018.

[29] A. Byravan and D. Fox. Se3-nets: Learning rigid body motion using deep neural networks. In *ICRA*, 2017.

[30] A. Byravan, F. Leeb, F. Meier, and D. Fox. Se3-pose-nets: Structured deep dynamics models for visuomotor planning and control. In *ICRA*, 2018.

[31] F. Sadeghi and S. Levine. Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*, 2016.

[32] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *IROS*, 2017.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention Is All You Need. In *NeurIPS*, 2017.

[34] A. Santoro, D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap. A simple neural network module for relational reasoning. In *NeurIPS*, 2017.

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

[36] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, M. Shanahan, V. Langston, R. Pascanu, M. Botvinick, O. Vinyals, and P. Battaglia. Deep reinforcement learning with relational inductive biases. In *ICLR*, 2019.

[37] A. Ajay, M. Bauza, J. Wu, N. Fazeli, J. B. Tenenbaum, A. Rodriguez, and L. P. Kaelbling. Combining physical simulators and object-based networks for control. *arXiv preprint arXiv:1904.06580*, 2019.

[38] T. Schaul, D. Horgan, K. Gregor, and D. Silver. Universal Value Function Approximators. In *ICML*, 2015.

[39] C. Fitzgerald. Developing Baxter. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, 2013.

[40] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[41] C. M. Bishop. Mixture density networks. Technical report, Citeseer, 1994.

[42] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven Exploration by Self-supervised Prediction. In *CVPR Workshops*, 2017.

[43] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.

[44] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba. Hindsight experience replay. In *NeurIPS*, 2017.

[45] Y. Zhang, J. Hare, and A. Prügel-Bennett. Deep set prediction networks. *arXiv preprint arXiv:1906.06565*, 2019.

[46] B. D. Ziebart, A. L. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In *Association for the Advancement of Artificial Intelligence*, 2008.

A Hardware and Software

Hardware: All models for the full method were trained on a single desktop computer (Intel i7-8700k CPU @ 3.70GHZ x 12) with an NVIDIA GTX 1080 Ti GPU. Some of the comparison models were also trained on a shared NVIDIA DGX machine. We use the Baxter robot from the late great RethinkRobotics [39], with custom 3D-printed plastic end effectors and foam inserts for compliant pushing of objects. We use the ASUS Xtion Pro RGB-D camera to capture images during inference. We only use the RGB channels, and we downscale the $640 \times 480 \times 3$ image the camera produces to size $84 \times 84 \times 3$ for use in our CNN. We use the same render image size in simulation and downscale the same way.

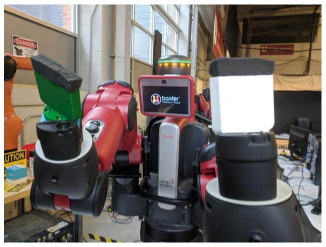
Figure 9: Close-up of compliant pushing end effectors.

Software: We use the Mujoco simulator with its built-in renderer. During training in simulation, for simplicity, we do not articulate the robot arm to push the objects. Instead, we articulate a disembodied paddle using simple PD control (seen in Fig. 2). We also place invisible walls around the table to prevent objects from falling off. To disincentivize policies from learning strategies that would push objects off the table in the real world, we give -1 reward each time one of the objects contacts the walls. In the real world, we use the Move-It! manipulation software, with an RRT Connect planner to plan pushing actions for the 7-dof Baxter arms. We choose to either use the right or left arm based on where the pushing action is on the table. During our evaluation, we run an autonomous loop where we: (1) collect an image from the ASUS camera, (2) feed that image to the policy, (3) sample and execute an action, and (4) repeat until 15 pushes have been executed or the goal has been reached. During training, we always sample actions according to the Gaussian parameterized by the SAC policy network. At inference, we continue to sample, but we reduce the standard deviation by 50%. We find this strikes a good balance between having too much noise, where the policy often executes suboptimal pushes, and having too little noise, where the policy gets stuck in certain states when slightly confused about the location of the objects.

We use TensorFlow as our deep learning framework. On top of that, we use TensorFlow Probability [6] to implement our Mixture Density Network. And we use the Sonnet [7] and graph_nets [8] frameworks developed by DeepMind to implement our CNNs and GNNs, including by using their implementation of multi-head dot product attention (MHDPA).

B Implementation Details

Dataset for Training the Encoders: We collect 1 million transitions in our initial dataset for training the model. This takes about 24 hours of wall time on the desktop machine. We did not experiment with using less data, but we suspect 1 million is a bit overkill; fewer examples may have worked just as well. We also apply data augmentation by adding random image contrast (`tf.image.random_contrast`) and noise (`tf.random.normal`) to the image as well as the embeddings before applying the MDN and dynamics heads.

Autoencoder Details: Our Autoencoder encodes its input into a latent space ϕ and then outputs a set of logits of the same size as the observation image. To better control for differences in the model, we use the same asymmetric actor critic apporach we do with the full method. We use both a CNN and GNN that map to latent spaces ϕ_o and ϕ_s, repsectively. We then independently pass both these latent spaces through a series of upconvolutions to output a 3D array ($W \times H \times C$) of logits that parameterize independent Bernoulli distributions which are trained to maximize the log probability of the true pixel values. The difference between this and a full Variational Autoencoder is that we do not enforce a prior distribution onto the latent space and we do not sample from the latent space. We tried enforcing this prior and sampling, but found that doing this did not allow the model to capture the object positions, at least during initial development. We use the same architecture for

[6]https://www.tensorflow.org/probability
[7]https://github.com/deepmind/sonnet
[8]https://github.com/deepmind/graph_nets

the GNN and CNN as we do in our full method. We find that the GNN-based representation provides a similar performance boost as what we see in the full method.

Initially, we found that applying domain randomization led this Autoencoder approach to fail. All of the capacity of the Autoencoder was wasted modeling the randomized textures and varying camera characteristics and it was not able to localize the objects. To solve this issue, we use the canonicalization approach from James et al. [17]. The idea is to collect another image observation at each time step that is in a "canonical" format which is not randomized and thus the only source of difference is the object positions (see Fig. 10).

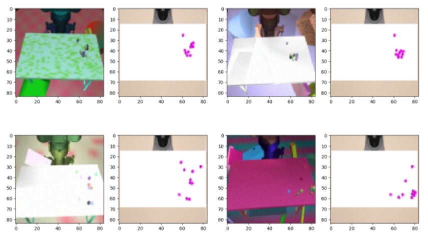
Figure 10: Domain randomized (1st & 3rd columns) and canonical (2nd & 4th) observations. Domain randomized images are used for the inputs of the Autoencoder, whereas a standard canonicalized version of the image is always used for the output prediction. This canonicalized version is always from an overhead view with no background distractors and only majenta objects.

MLP Details: Our MLP consumes the set of n object coordinates as an array of size $n \times 2$ (which gets flattened) and produces a latent representation ϕ. We design it to have roughly the same number of parameters as the GNN, as shown in Table 3. It consists of 3 hidden layers of size 256.

C Hyperparameters

We use the same hyperparameters, except for ϵ, to train all of the models during the supervised and reinforcement learning phases. We found the GNN and MLP were robust to different learning rates, so we use a value which works well for the CNNs, including the Autoencoder network. We find using fairly large batch sizes was helpful to learn while applying noise via domain randomization and data augmentation. We use tanh (instead of ReLU) activations because prior sim-to-real work suggests that since tanh is a smoother activation, it leads to better domain transfer performance [14].

We list the supervised learning hyperparameters in Table 1, the reinforcement learning hyperparameters in Table 2, and the model parameter counts in Table 3. The asterisks (*) represent hyperparameters from the supervised phase that stay the same in the RL phase.

Table 1: Supervised representation learning hyperparameters to train all models

Hyperparameter	Value
optimizer*	Adam
learning rate	3×10^{-4}
batch size	512
number of iterations	75000
activation*	tanh
conv filters*	64
ϕ size*	128
MDN K*	25
transformer vector size*	128
transformer num heads*	8

Table 2: RL hyperparameters to train all models

Hyperparameter	Value
learning rate	1×10^{-3}
batch size	1024
replay buffer size	1×10^{6}
N (max episode steps before termination)	50
number of parallel environments	8
number of iterations	10000
ϕ noise	0.1
HER [44] k	8
SAC [43] entropy bonus (α)	0.1
polyak averaging (ρ)	0.995

Tuning ϵ Across Methods: To ensure that all methods face a similarly difficult training task, we have to make sure the ϵ we use correspond to similar state similarities. We do this empirically by collecting a set of 4096 transition pairs (s_t, s_{t+1}) using our scripted policy. We filter out all transitions where the objects have

Table 3: Number of parameters in different network models (during supervised learning)

Model	Number of parameters
GNN for Full	232197
CNN for Full	353157
CNN Autoencoder and Decoder	937975
MLP (3 hidden layers of size 256)	253445

Table 4: ϵ values

Model	ϵ Value
Full	0.005
Autoencoder	0.2
Only $\mathcal{L}_{\text{STATE}}$	0.04
Only $\mathcal{L}_{\text{DYN}}$	0.004
MLP	0.005

not moved at all. Then we compute the cosine distances between all remaining pairs for each of the models. Finally, we compute relative scalings between the values produced by each model. Through the manual tuning of ϵ for various models, we found these relative scalings correlated well with similar task difficulties and training performances. The resultant ϵ values are shown in Table 4.

D Extra Background

D.1 Soft (Asymmetric) Actor Critic:

To train a goal-conditioned policy, we use a slightly modified version of the Soft Actor Critic (SAC) algorithm [43]. SAC is a Deep Deterministic Policy Gradient (DDPG) style algorithm, which optimizes a stochastic policy in an off-policy way. It is based on the maximum entropy reinforcement learning framework [46] and has been efficient and robust for learning real-world robotics tasks. As opposed to vanilla SAC, we feed the value networks (V, Q_1, Q_2) a representation encoded from the underlying state, s, while we feed the policy network π a representation encoding the observation (RGB image), o, in an asymmetric actor critic style approach [10]. We have found that it leads to stable training and helped policies from plateuaing too early in training, compared to other model-free RL algorithms we tried.

D.2 Hindsight Experience Replay:

We also use Hindsight Experience Replay (HER) [44]. HER is a method for augmenting replay buffer data by relabeling goals to aid training in sparse reward settings. After sampling a goal g and collecting an episode of experience, HER stores the original transitions $(s_t, a_t, r_t, s_{t+1}, g)$ in the replay buffer, as usual. Additionally, for each step t, it samples a set of virtual goals $(g'_1, g'_2, ...)$ that come from future states of the episode (which it has guaranteed to have reached), and it uses these to create several relabeled transitions $(s_t, a_t, r'_t, s_{t+1}, g'_i)$ that it also stores in the replay buffer. The denser rewards of these relabeled transitions is used to bootstrap learning and make the policy more capable of reaching original goals.