A. Additional Task Details

The full reward function of the *Rearrange* task is described in Equation (1).

$$r_t = 10\mathbb{I}_{success} + 5\mathbb{I}_{pick} + \Delta^{obj}_{arm} + \Delta^{goal}_{obj} - 0.001C_t \quad (1)$$

Where:

- $\mathbb{I}_{success}$ is the indicator for task success.
- $\mathbb{I}_{pick}$ is the indicator if the agent just picked up the object.
- Δ^{obj}_{arm} is the change in Euclidean distance between the end-effector (arm) and the target object (obj). If d_t is the distance between the two at timestep t, then $\Delta^{obj}_{arm} = d_{t-1} - d_t$).
- Δ^{goal}_{obj} is the change in Euclidean distance between the object (obj) and the goal position (goal).
- C_t Is the squared difference in joint action values between the current and previous time step. If a^k_t is the action for timestep t for moving joint k then $C_t = \sum_k (a^k_t - a^k_{t-1})^2$

The reward signal for the *mobile pick* is identical, but the task ends in a success if the robot picks the correct object ($\mathbb{I}_{success} = \mathbb{I}_{pick}$).

The action space of the monolithic policy consist of 11 actions:

- 7 continuous actions controlling the change to the joints angles. These actions are normalized between -1 and 1 with the minimum and maximum value corresponding to the maximum change to the joint angles allowed in each direction per step.
- 1 continuous action between -1 and 1 corresponding to the robot moving forward. An action with value of 1 corresponds to the robot moving forward by 10cm and -1 to the robot moving backward by 10cm in the simulation.
- 1 continuous action between -1 and 1 corresponding to the robot rotating. An action with value of 1 corresponds to the robot rotating in clockwise by $5°$ and -1 to the robot rotating counter-clockwise by $5°$.
- 1 discrete action with 2 options corresponding to the robot attempting to grasp or release an object. If the value is 0, and the robot is holding an object, the robot will attempt to release it. If the value is 1 and the robot is not holding an object, the robot will attempt to grasp.
- 1 discrete action with 2 options corresponding to the robot attempting to terminate the episode. If the value is 0 the robot will continue the task. If the value is 1, the robot will signal that the task is completed.

B. Method Details

More details about the method architecture here.
Our Hyperparameters are described in Table 3.

Hyperparameter	Value
start learning rate	3.5×10^{-4}
end learning rate	0
learning rate schedule	$linear$
entropy coefficient	1×10^{-3}
clip gradient norm	2.0
time horizon	64
number of epochs per updates	1
number of mini batches per updates	2
RGB and Depth image resolution	128×128
image encoder	$ResNet18$
normalized advantage	$true$

Table 3. Hyperparameters used for DD-PPO training in Galactic

To calculate the entropy of this action space for entropy regularization in DD-PPO, we add the entropy of the discrete and continuous actions distributions together without any scaling.

The SimpleCNN model we use consists of 3 convolution layers followed by a fully connected layer. The kernel sizes for the three convolution layers are 8×8, 4×4 and 3×3, the strides are 4×4, 2×2 and 1×1 and there is no dilation nor padding. This is the same SimpleCNN visual encoder used in Habitat 2.0. The size of the models used are described in Table 4.

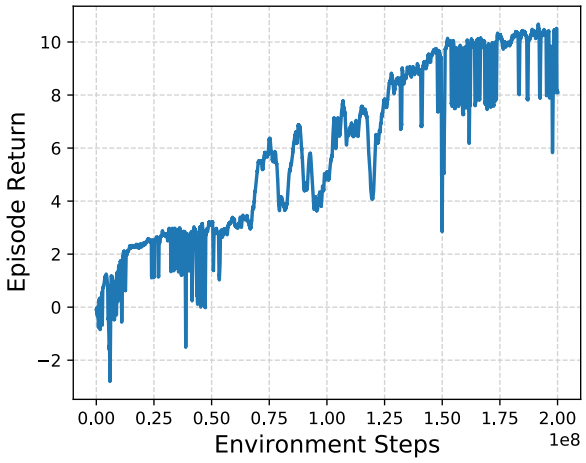
Model	Total number of parameters
SimpleCNN	4,046,999
ResNet9	4,338,007
ResNet18	5,906,647

Table 4. Model sizes for the different visual encoders used. This includes the visual encoder, the actor, and the critic.

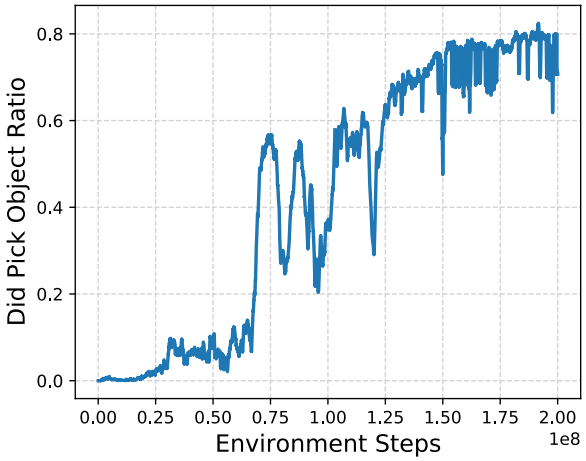
C. Further Habitat 2.0 Results

First, we analyze the poor performance of the policy purely trained in Habitat 2.0, which achieves no success in the Table 2. Figure 7a shows the reward learning curve during training. This learning curve demonstrates that even after 200M steps of training, the reward is still increasing, which provides evidence for the necessity of Galactic to scale training. In this training time, the agent reliably learns to pick the object around 80% of the time as shown by the training plot in Figure 7b comparing the fraction of the time the robot picked the object within an episode versus the number of training steps.

Next, we analyze the source of errors in the zero-shot transfer from Galactic to Habitat 2.0. We show the drop in performance is not due to the dynamic arm control by comparing to transferring to Habitat 2.0 with a kinematic arm controller instead of a dynamics-based torque controller. The agent with



(a) Reward



(b) Picked object ratio

Figure 7. Learning curves for the policy trained purely in Habitat 2.0. Figure 7a shows the episode reward does not saturate even after 200M training steps. Figure 7b shows that even though the agent is never successful, it still learns to pick the object.

the kinematic arm controller achieves a 29.7% success rate on the "Eval" dataset, barely any better than the 26.4% success rate the dynamics-based torque controller achieves.

D. Additional Task visuals

In this section, we visualize observations rendered using the Galactic simulator. Figure 8 are examples of 128×128 RGB images used for training. Figure 9 are examples of 128×128 depth images used for training. We also visualize observations rendered using the Habitat 2.0 simulator also at 128×128 in Figure 10 and Figure 11.

E. Training for 15 Billion steps

To show the usefulness of training for several billions of steps, we trained the *Rearrange* task defined in Section 4 for 15 billion steps. Training and validation success rates are still improving, showing that training still hasn't converged, even after 15 billion steps.

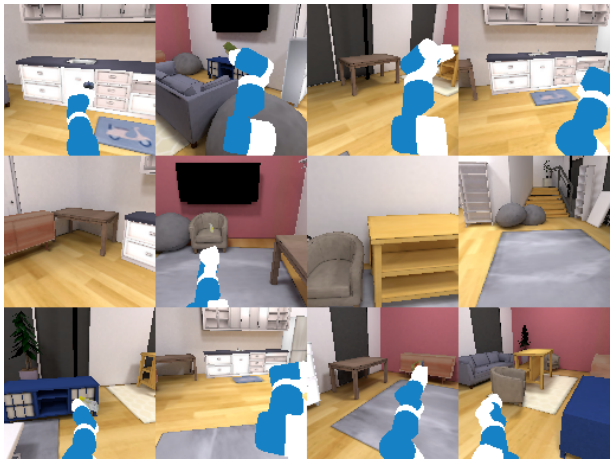


Figure 8. Samples of RGB observations collected in Galactic.

Figure 9. Samples of Depth observations collected in Galactic.

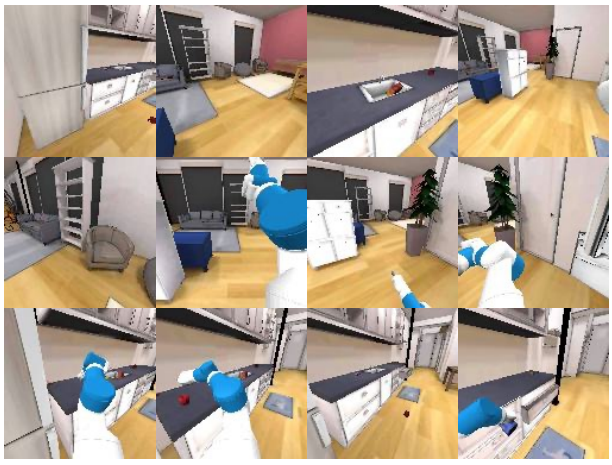


Figure 10. Samples of RGB observations collected in Habitat 2.0.

Figure 11. Samples of Depth observations collected in Habitat 2.0.

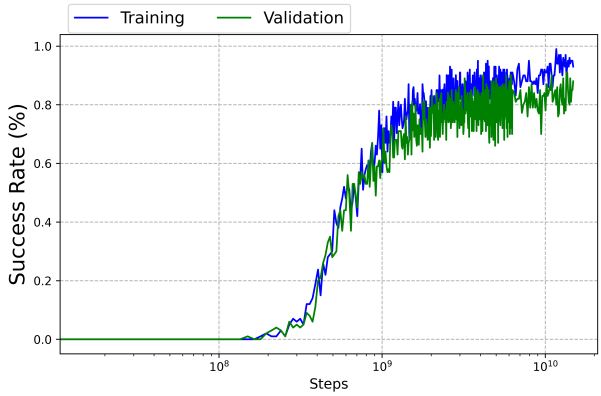


Figure 12. Training and evaluation curves for a 15 billion steps training run. Each checkpoint is evaluated on 100 training or validation episodes.

Visual Encoder	ResNet18	SimpleCNN
Number of Envs	128	512
PyTorch Inference	7.52	8.80
Render Setup	0.77	1.47
Step Post-processing*	1.84	2.05
Step Physics*	1.78	3.76
GPU Rendering	2.27	9.68
Additional CPU	0.74	1.00
Total	11.30 ms	20.95 ms

Table 5. Timing breakdown of a single batch rollout step, for two configurations in milliseconds. * Post-processing and Physics are interleaved with GPU rendering and PyTorch inference and don't contribute to total rollout step time. See also Figure 1. 1x Tesla V100, 10x Intel Xeon Gold 6230 CPU @ 2.10GHz, 128x128 RGBD sensors.

F. Performance Timings

G. Additional Collision-Detection Details

In this section, we'll expand on Section 3.2, in particular, we'll discuss our collision representations and collision-detection queries.

Visual Encoder	ResNet18	SimpleCNN
Number of Envs	128	512
Compute Rollouts	726	1289
Update Agent	1381	856
Total	2204 ms	2215 ms
Training SPS	3716 SPS	14791 SPS

Table 6. Timing breakdown of a single train update for two configurations in milliseconds. 1x Tesla V100, 10x Intel Xeon Gold 6230 CPU @ 2.10GHz, 128x128 RGBD sensors, 64 batch rollout steps.

Galactic scenes include a Fetch robot [43], movable YCB objects [4], and 105 static (non-movable) ReplicaCAD scenes [55]. Note that scenes in the ReplicaCAD dataset include some interactive furniture (e.g. openable cabinet drawers and doors), but we don't simulate these in Galactic as they aren't required for the Rearrange Easy benchmark.

As discussed in Section 3.2, our approximate kinematic sim must perform collision queries between the robot (including grasped object, if any) and the environment (resting movable objects and the static scene). We represent each articulated link of the robot with a set of spheres (green in Figure 3). These are authored manually, with the goal to approximate the shape of the Fetch robot with a minimal number of spheres. We also represent each grasped object with a set of spheres (blue). These are generated offline using a space-filling heuristic. Rather than supporting arbitrary sphere radius, we limit ourselves to a sphere-radius "working set" of$\{$1.5 cm, 5 cm, 12 cm$\}$. This limitation is important as we'll see shortly.

We approximate a ReplicaCAD scene as a voxel-like structure called a column grid (gray in Figure 3). A column grid is generated offline for a particular scene and a particular sphere radius from our working set, so we generate three column grids per scene. A column grid is a dense 2D array of columns in the XZ (ground) plane, with 3-centimeter spacing. For each column, we represent vertical free space as a list of layers. For example, a column in an open area of the room would contain just one layer, storing two floating-point height values roughly corresponding to the height of the floor and the height of the ceiling. A column in the vicinity of a table, meanwhile, would contain two layers: one spanning from the floor to the underside of the table, and another spanning from the table surface to the ceiling. Finally, the stored height values don't actually represent the surface heights themselves, but rather the height of the query sphere (of known radius) in contact with the surface. For ReplicaCAD scenes, the maximum number of layers for any column is approximately 10 and corresponds to columns in the vicinity of a particular bookshelf with many shelves.

A column grid is generated offline using the ReplicaCAD scene's source triangle mesh and Habitat 2.0's sphere-query functionality. We load the scene in Habitat 2.0 and use the scene extents to derive the column grid's XZ (ground-plane) extents. We iterate over this region using our chosen 3-cm

spacing. For each column, we perform a brute-force search of the vertical region at the column's XZ position, using a series of sphere-overlap and vertical sphere-casts to find the free spans.

At runtime, to detect collisions between the robot (including grasped object, if any) and the static scene, we implement a fast sphere-versus-column-grid query. First, we select the appropriate column grid corresponding to the query sphere's radius. Second, we retrieve the nearest column corresponding to the sphere's XZ position. Finally, we linearly search the column's layers to determine whether the query sphere's Y position is in free versus obstructed space. This linear search is accelerated using caching: we start the search from the same layer index found in recent searches. This leverages spatial and temporal coherency, for example, consider the robot reaching under a table: if one sphere from the robot arm's link is found to be between the floor and the underside of a table, it's likely that other spheres from that same link or other queries from succeeding timesteps will also lie in that vertical layer.

Whereas a grasped movable object is represented with a set of spheres (blue in Figure 3), a resting movable object is approximated as an oriented box (orange). This is computed from the YCB object's triangle mesh in a preprocess. At runtime, to detect collisions between the robot (including grasped object, if any) and the resting movable objects, we perform sphere-versus-box queries. There are generally 30 resting movable objects in the environment (1 target object and 29 distractor objects) and we need to avoid performing all 30 sphere-versus-box queries. So, we use a "regular grid" acceleration structure to quickly retrieve a list of nearby resting objects.

Resting movable objects are inserted into a regular grid at episode initialization. This is a dense 2D array spanning the XZ (ground) plane, with each cell storing a list of objects that overlap it. Objects will generally overlap multiple cells and thus be present in the object lists of multiple cells. When an object is grasped by the robot, it is removed from all relevant cells in the regular grid, and if the object is later dropped, it is re-inserted into the regular grid at its new resting position.

Let's consider how to find the list of nearby resting objects for a given query sphere. The regular grid spacing is chosen such that cells are at least $4\times$ the largest radius in our sphere-radius working set (12 cm). A query sphere may overlap up to four adjacent cells in the regular grid, e.g. the sphere is centered near the shared edge of two cells or the shared corner of four cells. A naive approach here would be to merge and de-duplicate the object lists of the four cells. We avoid this expense and instead maintain *four separate regular grids*, all spanning the entire scene XZ extent, with carefully-chosen varying X and Z offsets for the cell boundaries. In this way, any query sphere is guaranteed to lie fully inside a single cell of one of these grids (not spanning a cell edge or corner). Thus, our list of nearby resting objects is simply the list stored in this cell; we don't have to merge or de-duplicate multiple lists. Note this approach of four somewhat-redundant regular grids comes at the expense of extra memory and added insertion/removal compute time.

H. Simulator Flexibility to new Assets

Galactic can work with various assets (robots, scenes and objects) from different sources. We use a mostly-automated pipeline that includes optimizing assets for the batch renderer and generating collision geometry (see Appendix G). In Fig. 13 we added Stretch and Spot robots loaded in a scene from the MP3D dataset [41] with new objects.

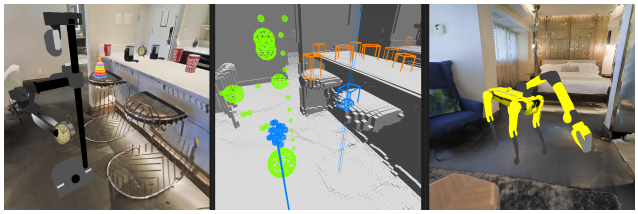


Figure 13. Galactic with an MP3D scene, Google Scanned Objects [13], Stretch robot (left), Stretch debug viz (center), and Spot robot (right).

I. Description of Heuristics

I.1. Sliding Heuristic

We implement the robot sliding heuristic as the following steps: (1) start from a candidate pose, (2) if the pose penetrates the scene, compute a jitter direction in the ground plane (3) jitter the robot base in the horizontal plane. Repeat from step 1 until a penetration-free pose is found, up to 3 times. If this fails, the robot does not move on that step.

I.2. Object Placing Heuristic

The object placement heuristic is as follows: (1) start from a candidate pose, (2) if the pose penetrates the scene, compute a jitter direction in the ground plane, with some randomness, (3) jitter the dropped object and re-cast down to a support surface. Repeat from step 1 until a penetration-free pose is found, up to 6 times. If this fails, we restore the dropped object to its resting position prior to grasp. This approximates the dropped object bouncing or rolling away. Snap-to-surface is an instantaneous operation that resolves within one physics step; objects do not fall or settle over time.