Category	# Train Instances	# Test Instances	# Test Pose Initializations
Scissors	37	9	90
Knife	18	4	40
USB	15	3	30
Safe	18	4	40
Pliers	-	24	48
Microwave	-	9	18
Lighter	-	11	22
Eyeglasses	-	33	66
Multilink	-	10	20
Total	88	107	374

Table 3. **Instance Counts.** Details about the number of assets and testing pose initializations.

A. Data Generation

In this section we give more details about our data generation pipeline for Act the Part (AtP).

A.1. Assets

For object asset data, we use instances from eight categories from the Partnet-Mobility [9, 29, 49] dataset and a ninth multilink category of our creation—configured with three links in a chain. Examples from each category are shown in Fig. 10. Number of instances for train splits, test splits, and the number of initial poses per category for testing can be found in Tab. 3.

Partnet-Mobility. Object categories were selected for realism in table-top environments and provide opportunity for reasonable exploration of the hold and push action pair. For this work, we also consider top-down views, which are commonly used in robot bin clearing tasks. Notably, not enough categories with prismatic joint (e.g., furniture) fit these parameters. We leave tackling such objects to future work. We additionally filtered instances with missing 3D object meshes or unstable physical dynamics.

Multilink. Our multilink instances are composed of three links, with an ellipsoid flanked by two prisms. Each link is assigned a random color by sampling R, G, and B values uniformly with replacement. Joint angles take values in the range $[-5\pi/18, 5\pi/18]$ radians. We also introduce tooling to create arbitrary multilink instances with mesh primitives, which can be used in future work and applications. Note: procedurally generated multilink assets follow Partnet-Mobility conventions. Code for generating the multilink objects will be available online.

A.2. Physics Simulator

Our Simulation platform is based on Pybullet [11], a state-of-the-art physics simulator and wrapper for the Bullet Physics Library. Pybullet is widely used in vision, robotics, and reinforcement learning.

Image Backgrounds. We use wood textured backgrounds sourced from Kaggle[1]. The original data is scraped from the Wood Database[2]. We use 421 backgrounds from the dataset and partition into three sets: 141 instances for training, 140 for testing unseen instances from seen categories, and 140 for testing unseen categories. When initializing the Pybullet environment, depending on the train or evaluation setting, a background is drawn uniformly at random. For the testing, this is done *once* and initialization conditions are frozen to ensure a fixed benchmark. All initial poses and corresponding backgrounds will be released in our project code release.

Simulating Touch Feedback. We simulate sheer force touch feedback in our hold gripper. The touch feedback is measured by sensing constraint forces on the hold gripper. These are non-zero when hold and push are both on the same object. The existence of a constraint force indicates that the gripper must hold more forcefully to keep a point on the object fixed. While we can obtain rich signal including direction and magnitude of the forces, we limit our agent to a binary signal, which is more reliable in real world settings.

B. History Aggregation

Full Part memory. It is possible that all channels in part memory V are full when we want to allocate a new part M_{t+1}. In these cases, we assign to the channel c with the largest overlap with M_t: $V^c \leftarrow (V^c - (M_t \cap V^c)) \cup M_{t+1}$. This assignment potentially entangles two masks. All such cases are classified as *entangled parts*.

Perspectives on History Aggregation. The algorithm can be viewed as maintaining V as a hidden state. However, we opt to use our history aggregation module instead of an RNN. Training an RNN would require pseudo-labels, which realistically come from this algorithm or additional ground truth supervision.

Implementation Details. V is implemented as a 3D tensor. In this work we deal with objects with at most three links. However, in practice we use five channel tensor to relax the assumption that we can deal with only three parts. A priority queue maintains order of the most recently allocated and modified channels. This allows us to turn V into a single segmentation mask $\mathcal{M}$, by layering channels in an occlusion aware order.

C. Reward

Full Model with Touch. If no flow is observed in the scene, it implies the push is not proper, while not necessarily implying anything about the hold action. Hence, the push reward is 0 and no hold reward is back-propagated.

If flow exists, it implies the push is on the object; however, there is no guarantee of helpful motion. To better identify informative motion, we use touch feedback. (1) When hold and push are both on the object, the hold gripper will

[1] kaggle.com/edhenrivi/wood-samples

[2] wood-database.com

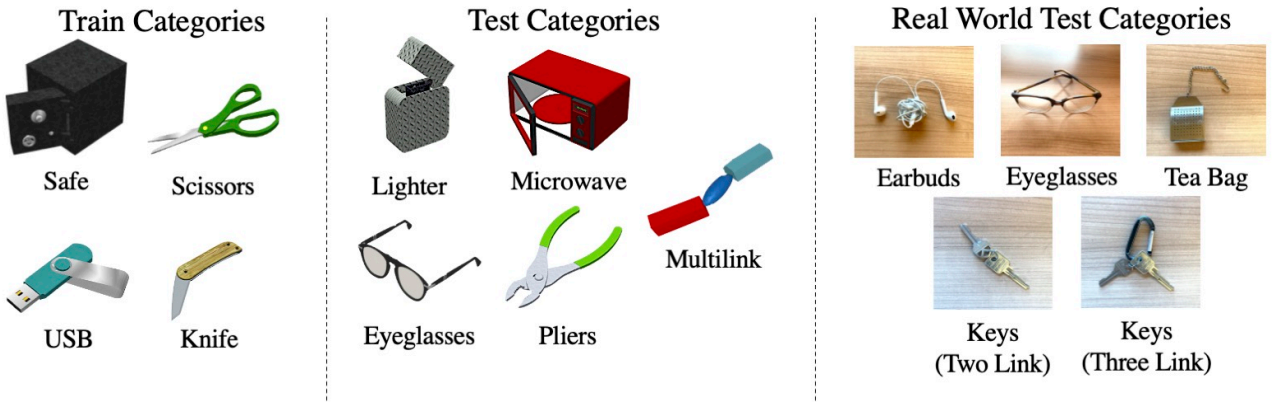

Figure 10. **Categories.** Sample of instances from train and test categories (synthetic and real world). Note during test time, unseen instances of train categories are also evaluated.

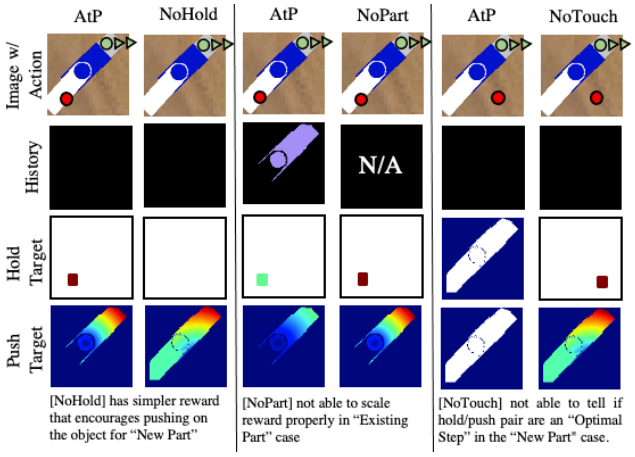

Figure 11. **Reward Shaping.** White represents no backprop, dark red a target 1, and dark blue target 0.

feel sheer force (thresholded at 0 to give binary signal), indicating the need to hold harder to keep a point fixed. In this case, (1a) we provide reward 1 to both hold and push pixels if a new part is discovered, and (1b) 0.5 if an existing part is moved. (2) If the hold gripper feels no force and there is motion, it is clear the push action created motion without anything being pinned. The hold reward is 0 and no push reward is back-propagated. (3) If the agent pushes a previously discovered part along with another undiscovered part, motions are entangled. Here, we penalize hold with reward 0 and do not update push network, as pushing is conditioned on the hold.

When supervising push affordances, we also enforce reward 0 for the hold pixel, which should teach the agent not to push where it holds. We provide example (shaped) targets in Fig. 11 for illustrative cases (also showing analogous ablation targets). The ablated versions are not able to discriminate the optimality of actions to the same degree as the full method. This ultimately affects downstream performance.

Optical Flow	Part Memory	Push Reward
x	-	0
✓	New part	1
✓	Existing part	.5
✓	Entangled part	0

Table 4. **Act-NoHold Reward.** Reward cases for holding are removed.

Optical Flow	Touch Sensor	Hold Reward	Push Reward
x	1/0	N/A	0
✓	1	1	1
✓	0	0	N/A

Table 5. **Act-NoPart.** Reward cases for part-awareness are removed.

Optical Flow	Part Memory	Hold Reward	Push Reward
x	-	N/A	0
✓	New part	1	1
✓	Existing part	.5	.5
✓	Entangled part	0	N/A

Table 6. **Ours-NoTouch.** Touch sensor is removed.

Other Models. We show reward for [Act-NoHold], [Act-NoPart], and [Ours-NoTouch] in Tabs. 4, 5, and 6 respectively.

D. Architecture

Interaction Network. We use a ResNet18 [16] backbone. The conv1 layer is modified to change the number of input channels from three to nine. No pretrained weights are used. The nine channels are justified as follows. The first three channels are used for an RGB image. The next five channels are used for the part memory V. The final channel is a hold channel, where an encoding of the hold location is passed when extracting the push map. When computing the hold map, this channel is filled with placeholder zeros.

We have two decoder heads branching off of the shared encoder, one for holding and the second for pushing. These heads are wired using residual connections similar to the

U-Net architecture [39]. We now describe a single upsampling block. Features are bilinearly upsampled by a factor of 2. A single conv layer is applied to reduce the number of channels by a factor of two. The result is concatenated with intermediate features of the same resolution from the backbone pass. The resulting volume is passed through two residual blocks, following the pytorch ResNet implementation. Each upsampling head is composed of four upsampling blocks. The output of these heads is a 1-channel map of logits used to predict reward for each input pixel. The first head is used to predict hold rewards and the second to predict push rewards, as discussed in Sec. 3.2.

For pushing, we want to simplify learning so the network only has to reason about pushing in a single direction (in our case right). To accomplish this we take eight rotations of our input volume, every 45°. To avoid data loss from rotations, we edge pad images (replicating edge values outward), before rotation and take a 128×128 center crop. 128 is sufficient as it preserve the diagonal of the original 90×90 image.

For the full forward pass, we first extract features from the image and part memory. We then use the hold decoder to predict a hold map and sample to get the hold location. The location is encoded as a gaussian. The encoded hold replaces the channel of zeros in the input volume. Features are again extracted by the backbone from eight rotations of the input. The push decoder is used to get the push maps, where we can then choose the push direction and sample a location.

Part Network. We use a ResNet18 backbone modified to take eight channel inputs. The channels are filled with RGB images at timestep t and $t + 1$. The last two channels are populated with gaussian encodings for the hold and the push. Again, no pretrained weights are used.

The decoder heads are architecturally the same as in the interaction network. They produce logit predictions for each pixel in the motion mask.

E. Training

Policy Rollouts for Data Collection. During each iteration of training, we rollout seven timesteps for each of the 88 train object instances from the scissors, knife, USB, and safe Part-Net Mobility categories. The data and the corresponding reward is saved in a buffer. Our buffer holds the last β iterations of data, whose distribution changes slowly as the model interactions improve. We empirically find $\beta = 10$ is a suitable rolling window to give good training set performance.

In total, the rollouts produce 73,920 interactions, from different stages in training. This corresponds to 73,920 frame pairs, actions, and ground truth flow transitions (consistent forwards and backwards). Flow is thresholded at zero to produce the motion mask ground truth necessary for supervising the part network.

Augmentations. We apply color jitter using pytorch APIs: brightness=0.3, contrast=0.4, saturation=0.3, hue=0.2. Additionally we randomly set images to grayscale with probability 0.2. After applying these augmentations, we normalize image RGB values using standard ImageNet mean and standard deviation. For the part network, we sample different augmentations for images at timesteps t and $t + 1$. For the part memory, we randomly swap channels to encourages invariance to the channel order. 20% of the time we fuse all the memory channels into one and modify the reward accordingly. This gives the model more context for splitting parts that are allocated to the same channel. For more details please refer to the code.

Training Details. The interaction and part networks are trained using SGD with momentum 0.9, learning rate 1×10^{-3}, weight decay 5×10^{-4}, cosine-annealing schedule with t-max=120, batch size 64, for 120 epochs. Hold map, push map, and part predictions are all supervised with binary cross entropy with logit loss. Each network trains on a single GeForce RTX 2080 11 GB card in less than 12 hours. For CPU rendering in the parallel rollouts, we make use of 48 Intel Xeon Gold 6226 (2.70GHz) cores.

F. Real World Details and Discussion

Pipeline details. To conduct interaction experiments in the real world we follow the following procedure.

- Take a picture of the object using an iPhone (setup shown in Fig. 12).
- Send picture to a laptop.
- Resize image to 90×90.
- Run AtP Interaction Network inference and visualize the selected hold location and push location (displayed in Fig. 13).
- Have human execute the action.
- Snap another picture and send to the computer.
- Run AtP Part Network inference to recover the part mask for this timestep.
- Repeat until last timestep at which point stop.

Discussion. When designing the real world interaction experiments, we abstract away low-level manipulation details to focus on validating our core algorithmic contributions (i.e., learning the part and interaction networks). Humans conduct the interactions predicted by the network. We acknowledge that having a human in the loop could introduce implicit biases for the interactions. Hence, we see immense value in robot experiments, which we leave to future work.

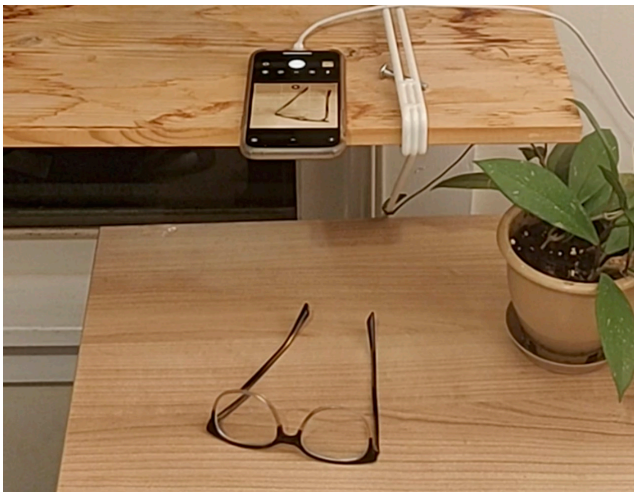

Figure 12. **Real World Setup.** A simple configuration for taking picture of the object to send to a laptop running our model.

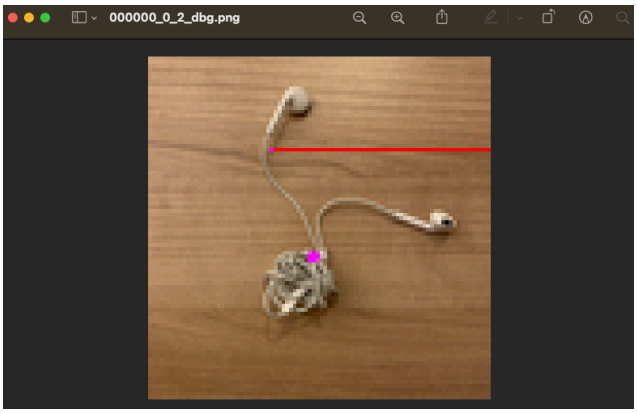

Figure 13. **Instructions.** Our model indicates the action for a human operator to execute. The magenta dot indicates the location to hold. The red line indicates the start location and direction of the push. GUI is preserved to give context for what one would see when conducting real world experiments.

G. Additional Results

Due to space limitations, we only presented interaction step and action results for pliers and multilink categories. Furthermore, we only showed real world results on eyeglasses. Here we provide additional qualitative and quantitative results.

Failure Analysis. See Fig. 14, where comments on the failures are provided in the caption.

Real World Results. We show results on four additional unseen categories: keys (two link, rigid), keys (three link, rigid), tea bag (two link, deformable), and earbuds (three link, deformable) in Fig. 15. Surprisingly our model works on these instances, even in the presence of deformable parts, which are not seen during training.

IoU and Interaction Steps. We present simulation benchmark results for the remaining object categories in Fig. 16. Our method approaches the upper bounds.

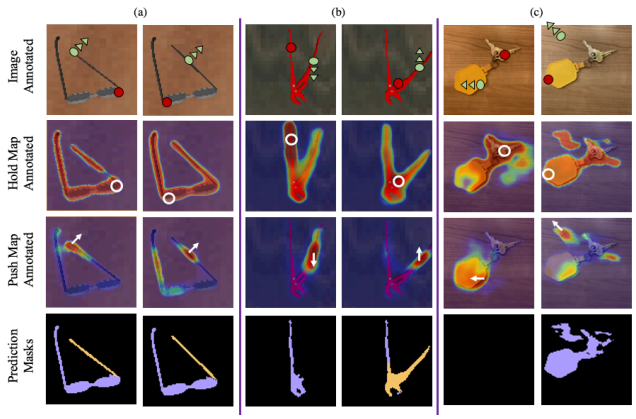

Figure 14. **Failure Modes.** (a) On three link objects our model sometimes struggles to split parts that have been grouped together in the part memory. We conjecture that this is due to the fact that we train on only two link objects. Because there are not many instances of having to split masks, the network might already think it has discovered all the parts. (b) We call attention to the erroneous (over) segmentation of the moved part. We notice that this is a common failure mode of our part network. (c) This is another case of erroneous segmentation, this time in a real world example. Notice that the camera gain is different between the two frames. Because segmentation is poor, subsequent action selection (i.e., the push in the top right) can suffer.

Effective and Optimal Steps. We notice our [Ours-Touch] and [Act-NoPart] perform efficient actions at roughly the same rate. However, [Ours-Touch] is able to find may more optimal steps. See Fig. 17.

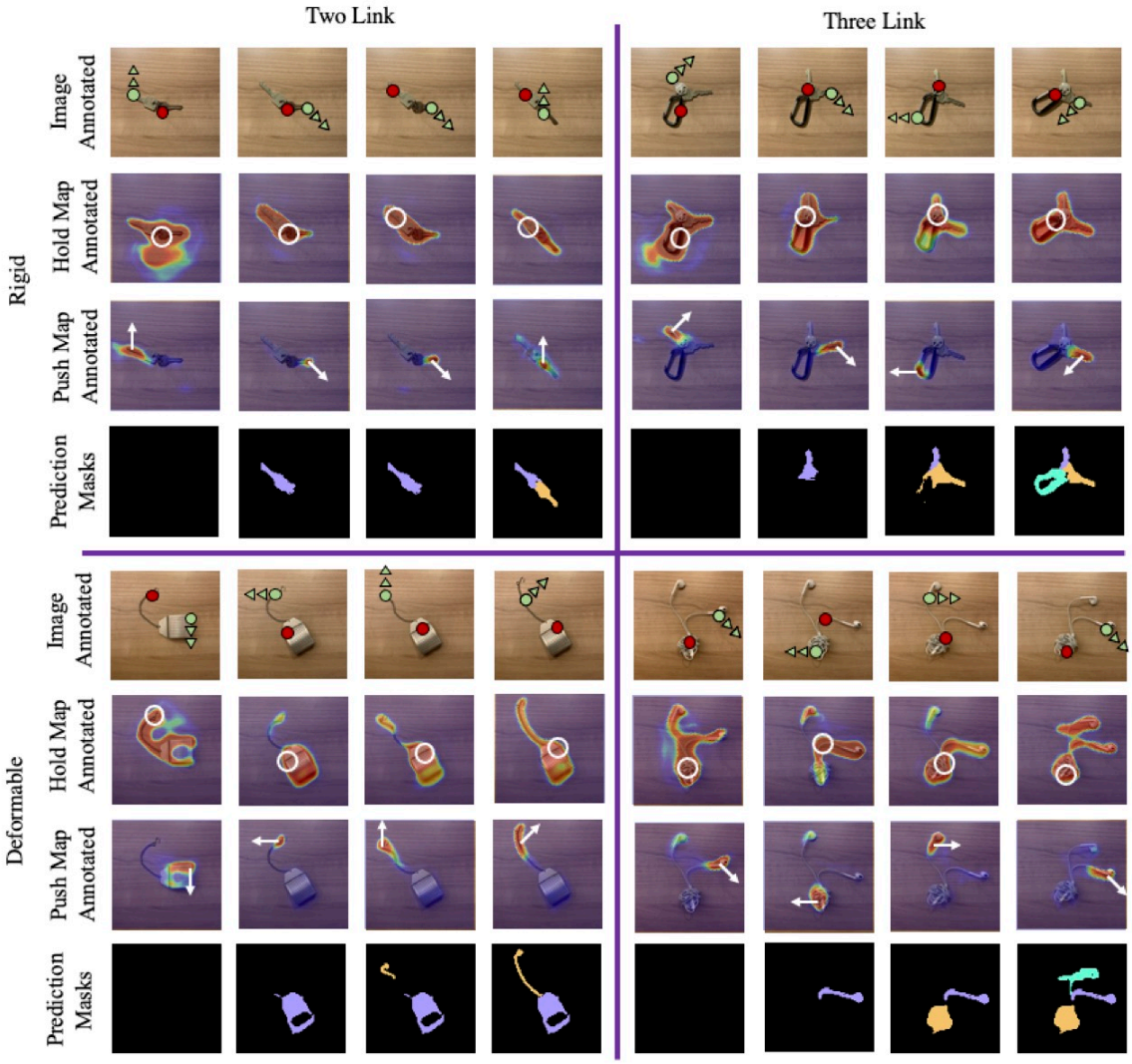

Figure 15. **Real World Results.** More results on real world objects.

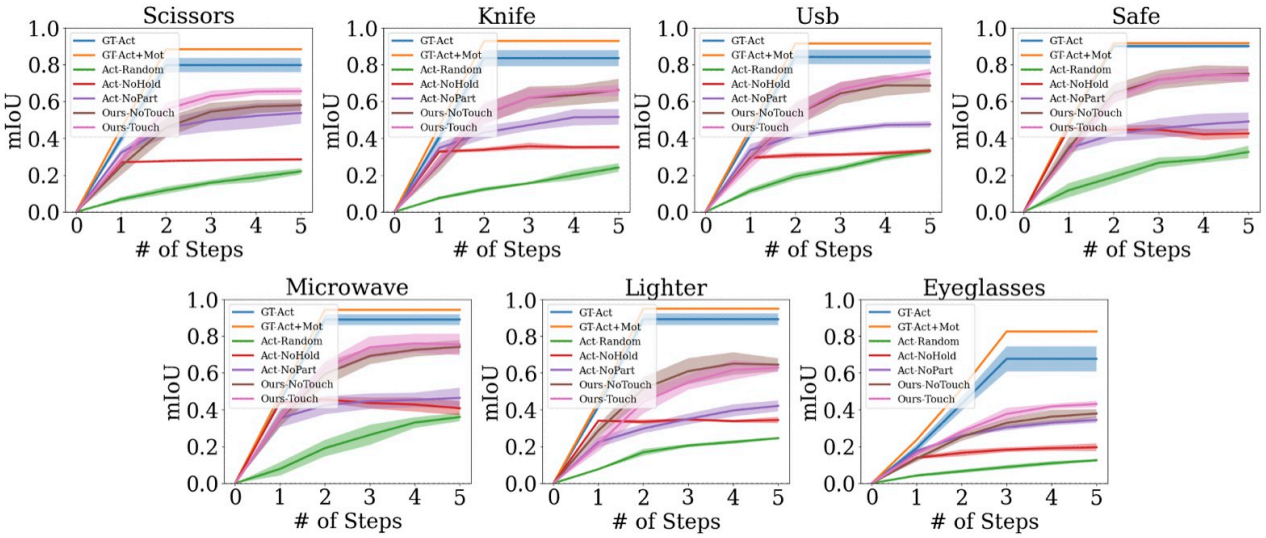

Figure 16. **IoU w.r.t. Interaction Steps.** Results for the remaining simulation categories not shown in the paper due to space restrictions.

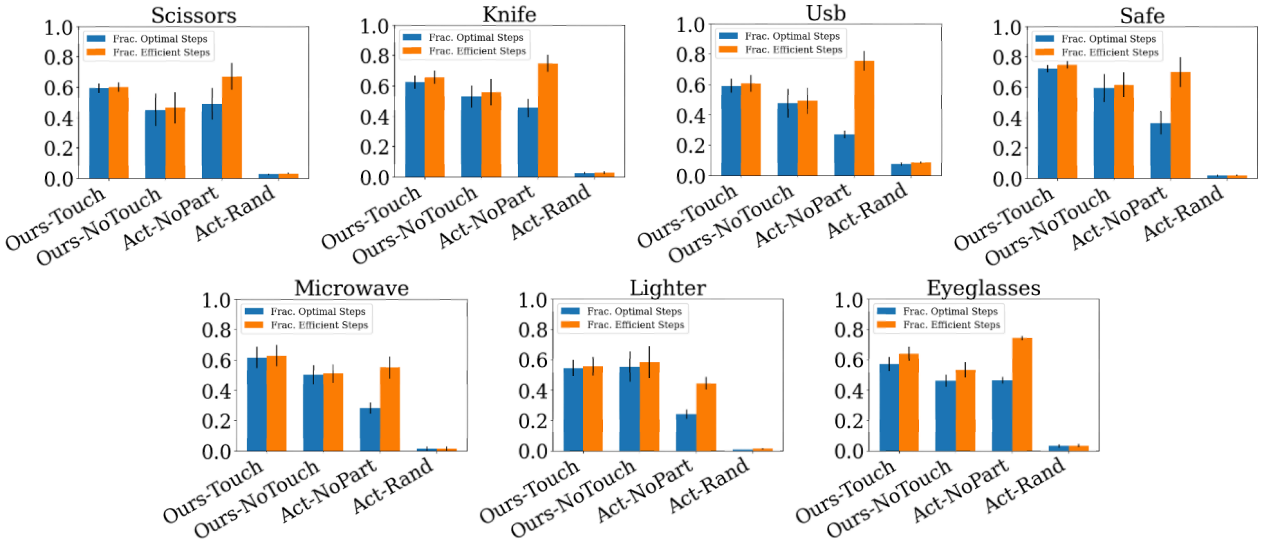

Figure 17. **Effective and Optimal Steps.** Results for the remaining seven simulation categories not shown in the paper due to space restrictions.