

A Task Details

Task	Variation Type	# of Variations	Avg. Keyframes	Language Template
open drawer	placement	3	3.0	“open the __ drawer”
slide block	color	4	4.7	“slide the block to __ target”
sweep to dustpan	size	2	4.6	“sweep dirt to the __ dustpan”
meat off grill	category	2	5.0	“take the __ off the grill”
turn tap	placement	2	2.0	“turn __ tap”
put in drawer	placement	3	12.0	“put the item in the __ drawer”
close jar	color	20	6.0	“close the __ jar”
drag stick	color	20	6.0	“use the stick to drag the cube onto the __ target”
stack blocks	color, count	60	14.6	“stack __ __ blocks”
screw bulb	color	20	7.0	“screw in the __ light bulb”
put in safe	placement	3	5.0	“put the money away in the safe on the __ shelf”
place wine	placement	3	5.0	“stack the wine bottle to the __ of the rack”
put in cupboard	category	9	5.0	“put the __ in the cupboard”
sort shape	shape	5	5.0	“put the __ in the shape sorter”
push buttons	color	50	3.8	“push the __ button, [then the __ button]”
insert peg	color	20	5.0	“put the ring on the __ spoke”
stack cups	color	20	10.0	“stack the other cups on top of the __ cup”
place cups	count	3	11.5	“place __ cups on the cup holder”

Table 3. Language-Conditioned Tasks in RLBench [15].

Setup. Our simulated experiments are set in RLBench [15]. We select 18 out of 100 tasks that involve at least two or more variations to evaluate the multi-task capabilities of agents. While PER-ACT could be easily applied to more RLBench tasks, in our experiments, we were specifically interested grounding diverse language instructions, rather than learning one-off policies for single-variation tasks like “[always] take off the saucepan lid”. Some tasks were modified to include additional variations. See Table 3 for an overview. We report average keyframes extracted from the method described in Section 3.2.

Variations. Task variations include randomly sampled colors, sizes, shapes, counts, placements, and categories of objects. The set of colors include 20 instances: `colors = {red, maroon, lime, green, blue, navy, yellow, cyan, magenta, silver, gray, orange, olive, purple, teal, azure, violet, rose, black, white}`. The set of sizes include 2 instances: `sizes = {short, tall}`. The set of shapes include 5 instances: `shapes = {cube, cylinder, triangle, star, moon}`. The set of counts include 3 instances: `counts = {1, 2, 3}`. The placements and object categories are specific to each task. For instance, `open drawer` has 3 placement locations: `top, middle, and bottom`, and `put in cupboard` includes 9 YCB objects. In addition to these semantic variations, objects are placed on the tabletop at random poses. Some large objects like drawers have constrained pose variations [15] to ensure that manipulating them is kinematically feasible with the Franka arm.

In the following sections, we describe each of 18 tasks in detail. We highlight tasks that were modified from the original RLBench [15] codebase⁴ and describe what exactly was modified.

A.1 Open Drawer

Filename: `open_drawer.py`

Task: Open one of the three drawers: `top, middle, or bottom`.

Modified: No.

Objects: 1 drawer.

Success Metric: The prismatic joint of the specified drawer is fully extended.

A.2 Slide Block

Filename: `slide_block_to_color_target.py`

Task: Slide the block on to one of the colored square targets. The target colors are limited to `red, blue, pink, and yellow`.

⁴<https://github.com/stepjam/RLBench>

Modified: Yes. The original `slide_block_to_target.py` task contained only one target. Three other targets were added to make a total of 4 variations.

Objects: 1 block and 4 colored target squares.

Success Metric: Some part of the block is inside the specified target area.

A.3 Sweep to Dustpan

Filename: `sweep_to_dustpan_of_size.py`

Task: Sweep the dirt particles to either the short or tall dustpan.

Modified: Yes. The original `sweep_to_dustpan.py` task contained only one dustpan. One other dustpan was added to make a total of 2 variations.

Objects: 5 dirt particles and 2 dustpans.

Success Metric: All 5 dirt particles are inside the specified dustpan.

A.4 Meat Off Grill

Filename: `meat_off_grill.py`

Task: Take either the chicken or steak off the grill and put it on the side.

Modified: No.

Objects: 1 piece of chicken, 1 piece of steak, and 1 grill.

Success Metric: The specified meat is on the side, away from the grill.

A.5 Turn Tap

Filename: `turn_tap.py`

Task: Turn either the left or right handle of the tap. Left and right are defined with respect to the faucet orientation.

Modified: No.

Objects: 1 faucet with 2 handles.

Success Metric: The revolute joint of the specified handle is at least 90° off from the starting position.

A.6 Put in Drawer

Filename: `put_item_in_drawer.py`

Task: Put the block in one of the three drawers: `top`, `middle`, or `bottom`.

Modified: No.

Objects: 1 block and 1 drawer.

Success Metric: The block is inside the specified drawer.

A.7 Close Jar

Filename: `close_jar.py`

Task: Put the lid on the jar with the specified color and screw the lid in. The jar colors are sampled from the full set of 20 color instances.

Modified: No.

Objects: 1 block and 2 colored jars.

Success Metric: The lid is on top of the specified jar and the Franka gripper is not grasping anything.

A.8 Drag Stick

Filename: `reach_and_drag.py`

Task: Grab the stick and use it to drag the cube on to the specified colored target square. The target colors are sampled from the full set of 20 color instances.

Modified: Yes. The original `reach_and_drag.py` task contained only one target. Three other targets were added with randomized colors.

Objects: 1 block, 1 stick, and 4 colored target squares.

Success Metric: Some part of the block is inside the specified target area.

A.9 Stack Blocks

Filename: `stack_blocks.py`

Task: Stack N blocks of the specified color on the green platform. There are always 4 blocks of the specified color, and 4 distractor blocks of another color. The block colors are sampled from the full set of 20 color instances.

Modified: No.

Objects: 8 color blocks (4 are distractors), and 1 green platform.

Success Metric: N blocks are inside the area of the green platform.

A.10 Screw Bulb

Filename: `light_bulb_in.py`

Task: Pick up the light bulb from the specified holder, and screw it into the lamp stand. The colors of holder are sampled from the full set of 20 color instances. There are always two holders in the scene – one specified and one distractor holder.

Modified: No.

Objects: 2 light bulbs, 2 holders, and 1 lamp stand.

Success Metric: The bulb from the specified holder is inside the lamp stand dock.

A.11 Put in Safe

Filename: `put_money_in_safe.py`

Task: Pick up the stack of money and put it inside the safe on the specified shelf. The shelf has three placement locations: `top`, `middle`, `bottom`.

Modified: No.

Objects: 1 stack of money, and 1 safe.

Success Metric: The stack of money is on the specified shelf inside the safe.

A.12 Place Wine

Filename: `place_wine_at_rack_location.py`

Task: Grab the wine bottle and put it on the wooden rack at one of the three specified locations: `left`, `middle`, `right`. The locations are defined with respect to the orientation of the wooden rack.

Modified: Yes. The original `stack.wine.py` task had only one placement location. Two other locations were added to make a total of 3 variations.

Objects: 1 wine bottle, and 1 wooden rack.

Success Metric: The wine bottle is at the specified placement location on the wooden rack.

A.13 Put in Cupboard

Filename: `put_groceries_in_cupboard.py`

Task: Grab the specified object and put it in the cupboard above. The scene always contains 9 YCB objects that are randomly placed on the tabletop.

Modified: No.

Objects: 9 YCB objects, and 1 cupboard (that hovers in the air like magic).

Success Metric: The specified object is inside the cupboard.

A.14 Sort Shape

Filename: `place_shape_in_shape_sorter.py`

Task: Pick up the specified shape and place it inside the correct hole in the sorter. There are always 4 distractor shapes, and 1 correct shape in the scene.

Modified: Yes. The sizes of the shapes and sorter were enlarged so that they are distinguishable in the RGB-D input.

Objects: 5 shapes, and 1 sorter.

Success Metric: The specified shape is inside the sorter.

A.15 Push Buttons

Filename: `push_buttons.py`

Task: Push the colored buttons in the specified sequence. The button colors are sampled from the full set of 20 color instances. There are always three buttons in scene.

Modified: No.

Objects: 3 buttons.

Success Metric: All the specified buttons were pressed.

A.16 Insert Peg

Filename: `insert_onto_square_peg.py`

Task: Pick up the square and put it on the specified color spoke. The spoke colors are sampled from the full set of 20 color instances.

Modified: No.

Objects: 1 square, and 1 spoke platform with three color spokes.

Success Metric: The square is on the specified spoke.

A.17 Stack Cups

Filename: `stack_cups.py`

Task: Stack all cups on top of the specified color cup. The cup colors are sampled from the full set of 20 color instances. The scene always contains three cups.

Modified: No.

Objects: 3 tall cups.

Success Metric: All other cups are inside the specified cup.

A.18 Place Cups

Filename: `place_cups.py`

Task: Place N cups on the cup holder. This is a very high precision task where the handle of the cup has to be exactly aligned with the spoke of the cup holder for the placement to succeed.

Modified: No.

Objects: 3 cups with handles, and 1 cup holder with three spokes.

Success Metric: N cups are on the cup holder, each on a separate spoke.

B PERACT Details

In this section, we provide implementation details for PERACT. See this [Colab tutorial](#) for a PyTorch implementation.

Input Observation. Following James et al. [14], our input voxel observation is a 100^3 voxel grid with 10 channels: $\mathbb{R}^{100 \times 100 \times 100 \times 10}$. The grid is constructed by fusing calibrated pointclouds with PyTorch’s `scatter_` function⁵. The 10 channels are composed of: 3 RGB, 3 point, 1 occupancy, and 3 position index values. The RGB values are normalized to a zero-mean distribution. The point values are Cartesian coordinates in the robot’s coordinate frame. The occupancy value indicates if a voxel is occupied or empty. The position index values represent the 3D location of the voxel with respect to the 100^3 grid. In addition to the voxel observation, the input also includes proprioception data with 4 scalar values: gripper open, left finger joint position, right finger joint position, and timestep (of the action sequence).

Input Language. The language goals are encoded with CLIP’s language encoder [76]. We use CLIP’s tokenizer to preprocess the sentence, which always results in an input sequence of 77 tokens (with zero-padding). These tokens are encoded with the language encoder to produce a sequence of dimensions $\mathbb{R}^{77 \times 512}$.

Preprocessing. The voxel grid is encoded with a 3D convolution layer with a 1×1 kernel to upsample the channel dimension from 10 to 64. Similarly, the proprioception data is encoded with a linear layer to upsample the input dimension from 4 to 64. The encoded voxel grid is split into 5^3 patches through a 3D convolution layer with a kernel-size and stride of 5, which results in a patch tensor of dimensions $\mathbb{R}^{20 \times 20 \times 20 \times 64}$. The proprioception features are tiled in 3D to match the dimensions of the patch tensor, and concatenated along the channel to form a tensor of dimensions $\mathbb{R}^{20 \times 20 \times 20 \times 128}$. This tensor is flattened into a sequence of dimensions $\mathbb{R}^{8000 \times 128}$. The language features are downsampled with a linear layer from 512 to 128 dimensions, and then appended to the tensor to form the final input sequence to the Perceiver Transformer, which of dimensions $\mathbb{R}^{8077 \times 128}$. We also add learned positional embeddings to the input sequence. These embeddings are represented with trainable `nn.Parameter`(s) in PyTorch.

Perceiver Transformer is a latent-space Transformer [1] that uses a small set of latent vectors to encode extremely long input sequences. See Figure 6 for an illustration of this process. Perceiver first computes cross-attention between the input sequence and the set of latent vectors of dimensions $\mathbb{R}^{2048 \times 512}$. These latents are randomly initialized and trained end-to-end. The latents are encoded with 6 self-attention layers, and then cross-attended with the input to out-

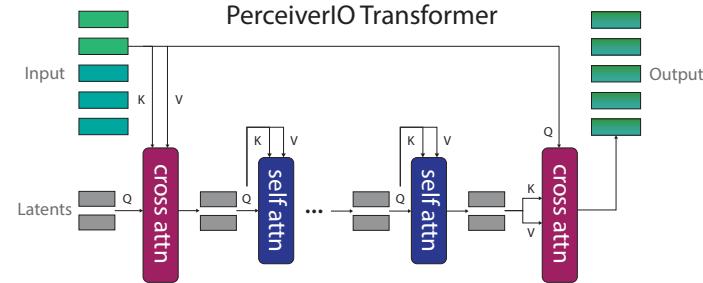


Figure 6. **Perceiver Transformer Architecture.** Perceiver is a latent-space transformer. Q , K , V represent queries, keys, and values, respectively. We use 6 self-attention layers in our implementation.

put a sequence that matches the input-dimensions. This output is upsampled with a 3D convolution layer and tri-linear upsampling to form a voxel feature grid with 64 channels: $\mathbb{R}^{100 \times 100 \times 100 \times 64}$. This feature grid is concatenated with the initial 64-dimensional feature grid from the processing stage as a skip connection to the encoding layers. Finally, a 3D convolution layer with a 1×1 kernel

⁵https://pytorch.org/docs/stable/generated/torch.Tensor.scatter_.html

downsamples the channels from 128 back to 64 dimensions. Our implementation of Perceiver is based on an existing open-source repository⁶.

Decoding. For translation, the voxel feature grid is decoded with a 3D convolution layer with a 1×1 kernel to downsample the channel dimension from 64 to 1. This tensor is the translation Q -function of dimensions $\mathbb{R}^{100 \times 100 \times 100 \times 1}$. For rotation, gripper open, and collision avoidance actions, the voxel feature grid is max-pooled along the 3D dimensions to form a vector of dimensions $\mathbb{R}^{1 \times 64}$. This vector is decoded with three independent linear layers to form the respective Q -functions for rotation, gripper open, and collision avoidance. The rotation linear layer outputs logits of dimensions \mathbb{R}^{216} (72 bins of 5 degree increments for each of the three axes). The gripper open and collide linear layers output logits of dimensions \mathbb{R}^2 .

Our codebase is built on the ARM repository⁷ by James et al. [14].

C Evaluation Workflow

C.1 Simulation

Simulated experiments in Section 4.2 follow a four-phase workflow: (1) generate a dataset with train, validation, and test sets, each containing 100, 25, and 25 demonstrations, respectively. (2) Train an agent on the train set and save checkpoints at intervals of 10K iterations. (3) Evaluate all saved checkpoints on the validation set, and mark the best performing checkpoint. (4) Evaluate the best performing checkpoint on the test set. While this workflow follows a standard train-val-test paradigm from supervised learning, it is not the most feasible workflow for real-robot settings. With real-robots, collecting a validation set and evaluating all checkpoints could be very expensive.

C.2 Real-Robot

For real-robot experiments in Section 4.4, we simply pick the last checkpoint from training. We check if the agent has been sufficiently trained by visualizing Q -predictions on training examples with swapped or modified language goals. While evaluating a trained agent, the agent keeps acting until a human user stops the execution. We also visualize the Q -predictions live to ensure that the agent’s upcoming action is safe to execute.

⁶<https://github.com/lucidrains/perceiver-pytorch>

⁷<https://github.com/stepjam/ARM>

D Robot Setup

D.1 Simulation

All simulated experiments use the four camera setup illustrated in Figure 7. The front, left shoulder, and right shoulder cameras, are static, but the wrist camera moves with the end-effector. We did not modify the default camera poses from RL Bench [15]. These poses maximize coverage of the tabletop, while minimizing occlusions caused by the moving arm. The wrist camera in particular is able to provide high-resolution observations of small objects like handles.

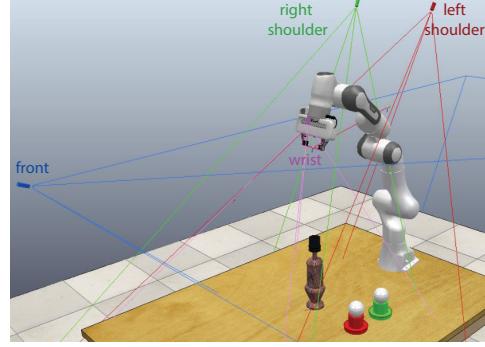


Figure 7. **Simulated Setup.** The four camera setup: front, left shoulder, right shoulder, and on the wrist.

D.2 Real-Robot

Hardware Setup. The real-robot experiments use a Franka Panda manipulator with a parallel-gripper. For perception, we use a Kinect-2 RGB-D camera mounted on a tripod, at an angle, pointing towards the tabletop. See Figure D for reference. We tried setting-up multiple Kinects for multi-view observations, but we could not fix the interference issue caused by multiple Time-of-Flight sensors. The Kinect-2 provides RGB-D images of resolution 512×424 at 30Hz. The extrinsics between the camera and robot base-frame are calibrated with the `easy_handeye` package⁸. We use an ARUCO⁹ AR marker mounted on the gripper to aid the calibration process.

Data Collection. We collect demonstrations with an HTC Vive controller. The controller is a 6-DoF tracker that provides accurate poses with respect to a static base-station. These poses are displayed as a marker on RViz¹⁰ along with the real-time RGB-D pointcloud from the Kinect-2. A user specifies target poses by using the marker and pointcloud as reference. These target poses are executed with a motion-planner. We use Franka ROS and MoveIt¹¹, which by default uses an RRT-Connect planner.

Training and Execution. We train a PER-ACT agent from scratch with 53 demonstrations. The training samples are augmented with $\pm 0.125\text{m}$ translation perturbations and $\pm 45^\circ$ yaw rotation perturbations. We train on 8 NVIDIA P100 GPUs for 2 days. During evaluation, we simply chose the last checkpoint from training (since we did not collect a validation set for optimization). Inference is done on a single Titan X GPU.

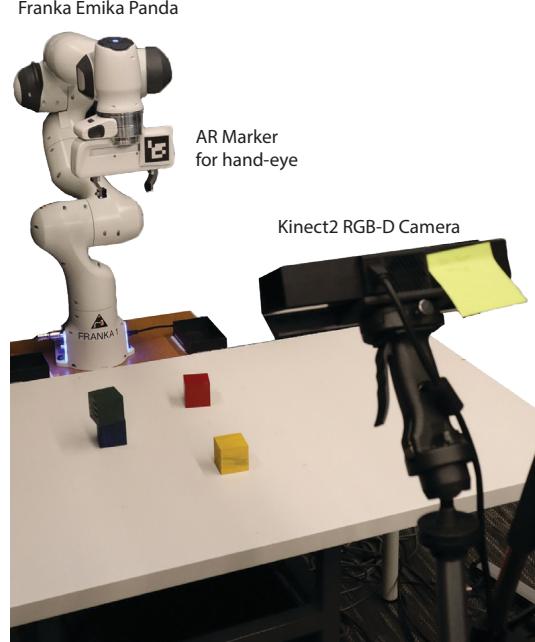


Figure 8. **Real-Robot Setup** with Kinect-2 and Franka Panda.

⁸https://github.com/IFL-CAMP/easy_handeye

⁹https://github.com/pal-robotics/aruco_ros

¹⁰<http://wiki.ros.org/rviz>

¹¹http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/

E Data Augmentation

PERACT’s voxel-based formulation naturally allows for data augmentation with SE(3) transformations. During training, samples of voxelized observations \mathbf{v} and their corresponding keyframe actions \mathbf{k} are perturbed with random translations and rotations. Translation perturbations have a range of $[\pm 0.125\text{m}, \pm 0.125\text{m}, \pm 0.125\text{m}]$. Rotation perturbations are limited to the yaw axis and have a range of $[0^\circ, 0^\circ, \pm 45^\circ]$. The 45° limit ensures that the perturbed rotations do not go beyond what is kinematically reachable for the Franka arm. We did experiment with pitch and roll perturbations, but they substantially lengthened the training time. Any perturbation that pushed the discretized action outside the observation voxel grid was discarded. See the bottom row of Figure 10 for examples of data augmentation.

F Demo Augmentation

Following James et al. [15], we cast every datapoint in a demonstration as a “predict the next (best) keyframe action” task. See Figure 9 for an illustration of this process. In this illustration, k_1 and k_2 are two keyframes that were extracted from the method described in Section 3.2. The orange circles indicate datapoints whose RGB-D observations are paired with the next keyframe action.

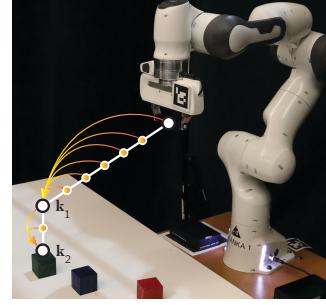


Figure 9. Keyframes and Demo Augmentation.

G Sensitivity Analysis

In Table 4, we investigate three factors that affect PERACT’s performance: rotation data augmentation, number of Perceiver latents, and voxelization resolution. All multi-task agents were trained with 100 demonstrations per task and evaluated on 25 episodes per task. To briefly summarize these results: (1) 45° yaw perturbations improve performance on tasks with lots of rotation variations like `stack blocks`, but also worsen performance on tasks with constrained rotations like `place wine`. (2) PERACT with just 512 latents is competitive with (and sometimes even better than) the default agent with 2048 latents, which showcases the compression capability of the Perceiver architecture. (3) Coarse grids like 32^3 are sufficient for some tasks, but high-precision tasks like `sort shape`

Table 4. Sensitivity Analysis. Success rates (mean %) of various PERACT agents trained with 100 demonstrations per task. We investigate three factors that affect PERACT’s performance: rotation augmentation, number of Perceiver latents, and voxel resolution.

	open drawer	slide block	sweep to dustpan	meat off grill	turn tap	put in drawer	close jar	drag stick	stack blocks
PERACT	80	72	56	84	80	68	60	68	36
PERACT w/o Rot Aug	92	72	56	92	96	60	56	100	8
PERACT 4096 latents	84	88	44	68	84	48	48	84	12
PERACT 1024 latents	84	48	52	84	84	52	32	92	12
PERACT 512 latents	92	84	48	100	92	32	32	100	20
PERACT 64^3 voxels	88	72	80	60	84	36	40	84	32
PERACT 32^3 voxels	28	44	100	60	72	24	0	24	0
PERACT 7^3 patches	72	48	96	92	76	76	36	96	32
PERACT 9^3 patches	68	64	56	52	96	56	36	92	20
	screw bulb	put in safe	place wine	put in cupboard	sort shape	push buttons	insert peg	stack cups	place cups
PERACT	24	44	12	16	20	48	0	0	0
PERACT w/o Rot Aug	20	32	48	8	8	56	8	4	0
PERACT 4096 latents	32	44	52	8	12	72	4	4	0
PERACT 1024 latents	24	32	36	8	20	40	8	4	0
PERACT 512 latents	48	40	36	24	16	32	12	0	4
PERACT 64^3 voxels	24	48	44	12	4	32	0	4	0
PERACT 32^3 voxels	12	20	52	0	0	60	0	0	0
PERACT 7^3 patches	8	48	76	0	12	16	0	0	0
PERACT 9^3 patches	12	36	72	12	0	20	0	0	0

need higher resolution voxelization. (4) Large patch-sizes reduce memory usage, but they might affect tasks that need sub-patch precision.

H High-Precision Tasks

In Table 1, PERACT achieves zero performance on three high-precision tasks: `place cups`, `stack cups`, and `insert peg`. To investigate if multi-task optimization is itself one of the factors affecting performance, we train 3 separate single-task agents for each task. We find that single-task agents are able to achieve non-zero performance, indicating that better multi-task optimization methods might improve performance on certain tasks.

	Multi	Single
<code>place cups</code>	0	24
<code>stack cups</code>	0	32
<code>insert peg</code>	0	16

Table 5. Success rates (mean %) of multi-task and single-task PERACT agents trained with 100 demos and evaluated on 25 episodes.

I Additional Related Work

In this section, we briefly discuss additional works that were not mentioned in Section 2.

Concurrent Work. Recently, Mandi et al. [83] found that pre-training and fine-tuning on new tasks is competitive, or even better, than meta-learning approaches for RLBench tasks in multi-task (but single-variation) settings. This pre-training and fine-tuning paradigm might be directly applicable to PERACT, where a pre-trained PERACT agent could be quickly adapted to new tasks without the explicit use of meta-learning algorithms.

Multi-Task Learning. In the context of RLBench, Auto- λ [73] presents a multi-task optimization framework that goes beyond uniform task weighting from Section 3.4. The method dynamically tunes task weights based on the validation loss. Future works with PERACT could replace uniform task weighting with Auto- λ for better multi-task performance. In the context of Meta-World [53], Sodhani et al. [84] found that language-conditioning leads to performance gains for multi-task RL on 50 task variations.

Language-based Planning. In this paper, we only investigated single-goal settings where the language instruction does not change throughout the episode. However, language-conditioning naturally allows for composing several instructions in a sequential manner [69]. As such, several prior works [85, 13, 86, 87] have used language as medium for planning high-level actions, which can then be executed with pre-trained low-level skills. Future works could incorporate language-based planning for grounding more abstract goals like “*make dinner*”.

Task and Motion Planning. In the sub-field of Task and Motion Planning (TAMP) [88, 89], Konidaris et al. [90] present an action-centric approach to symbolic planning. Given a set of predefined action-skills, an agent interacts with its environment to construct a set of symbols, which can then be used for planning.

Voxel Representations. Voxel-based representations have been used in several domains that specifically benefit from 3D understanding. Like in object detection [91, 92], object search [93], and vision-language grounding [94, 95], voxel maps have been used to build persistent scene representations [96]. In Neural Radiance Fields (NeRFs), voxel feature grids have dramatically reduced training and rendering times [97, 98]. Similarly, other works in robotics have used voxelized representations to embed viewpoint-invariance for driving [99] and manipulation [100]. The use of latent vectors in Perceiver [1] is broadly related to voxel hashing [101] from computer graphics. Instead of using a location-based hashing function to map voxels to fixed size memory, PerceiverIO uses cross attention to map the input to fixed size latent vectors, which are trained end-to-end. Another major difference is the treatment of unoccupied space. In graphics, unoccupied space does not affect rendering, but in PERACT, unoccupied space is where a lot of “action detections” happen. Thus the relationship between unoccupied and occupied space, i.e., scene, objects, robot, is crucial for learning action representations.

Long-Context and Latent-Space Transformers. Several approaches have been proposed for extending Transformers to longer context lengths [102]. Latent-space Transformers that use fixed-size latents instead of the full context, are one such approach [1, 103]. There is no clear winner in terms of trade-offs between speed, memory, and performance. However, latent-space methods have achieved compelling results in object detection [104] and slot-attention based object discovery [105].

J Additional Q-Prediction Examples

Figure 10 showcases additional Q -prediction examples from trained PERACT agents. Traditional object-centric representations like poses and instance-segmentations struggle to represent piles of beans or tomato vines with high-precision. Whereas action-centric agents like PERACT focus on learning perceptual representations of actions, which elevates the need for practitioners to define *what should be an object* (which is a harder problem and often specific to tasks and embodiments).

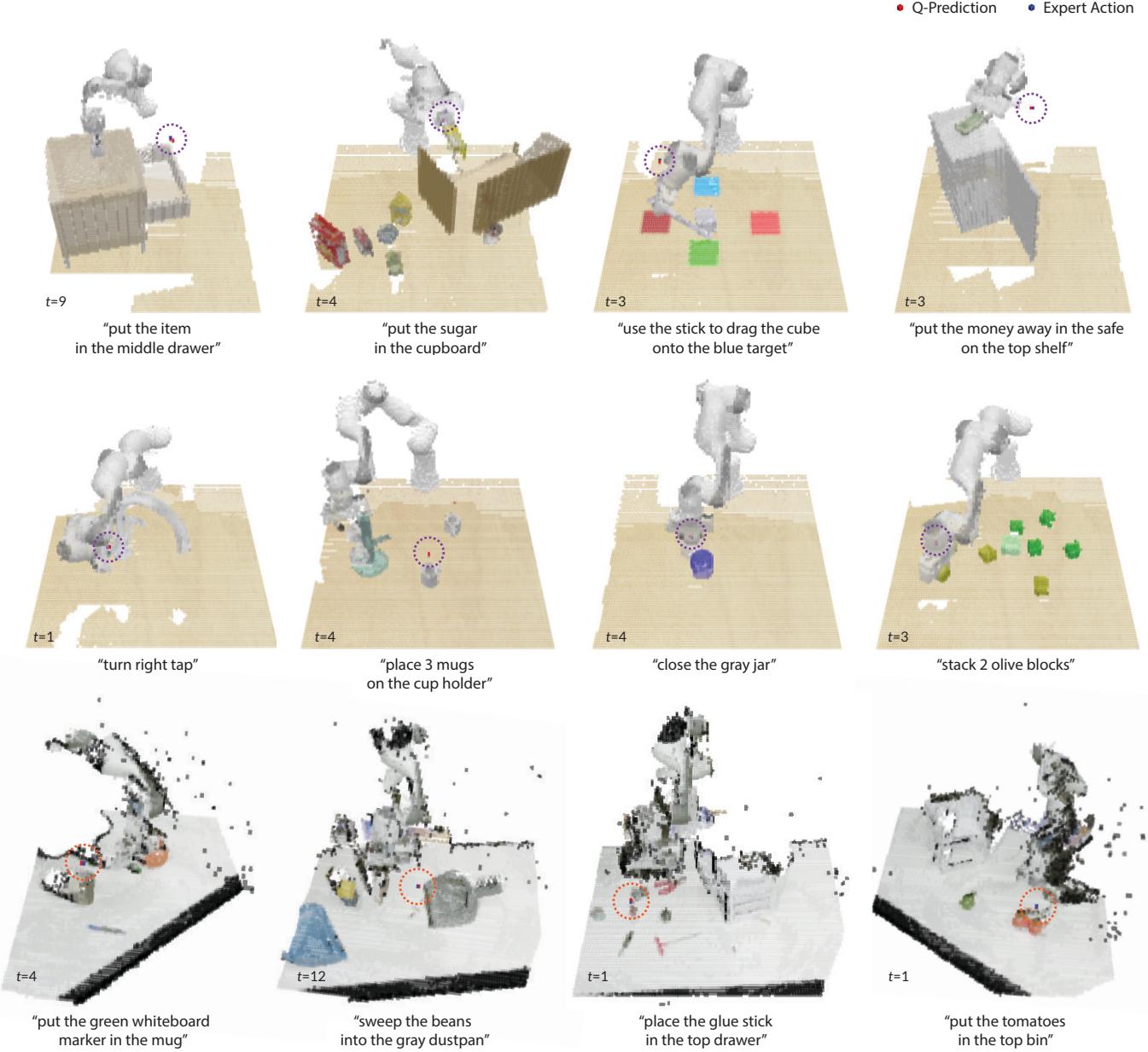


Figure 10. Additional Q-Prediction Examples. Translation Q -Prediction examples from PERACT. The top two rows are from simulated tasks without any data augmentation perturbations, and the bottom row is from real-world tasks with translation and yaw-rotation perturbations.

K Things that did not work

In this section, we describe things we tried, but did not work or caused issues in practice.

Real-world multi-camera setup. We tried setting up multiple Kinect-2s for real-world multi-view observations, but we could not solve interference issues with multiple Time-of-Flight sensors. Particularly, the depth frames became very noisy and had lots of holes. Future works could try turning the cameras on-and-off in a rapid sequence, or use better Time-of-Flight cameras with minimal interference.

Fourier features for positional embeddings. Instead of the learned positional embeddings, we also experimented with concatenating Fourier features to the input sequence like in some Perceiver models [1]. The Fourier features led to substantially worse performance.

Pre-trained vision features. Following CLIPort [16], we tried using pre-trained vision features from CLIP [76], instead of raw RGB values, to bootstrap learning and also to improve generalization to unseen objects. We ran CLIP’s ResNet50 on each of the 4 RGB frames, and upsampled features with shared decoder layers in a UNet fashion. But we found this to be extremely slow, especially since the ResNet50 and decoder layers need to be run on 4 independent RGB frames. With this additional overhead, training multi-task agents would have taken substantially longer than 16 days. Future works could experiment with methods for pre-training the decoder layers on auxiliary tasks, and pre-extracting features for faster training.

Upsampling at multiple self-attention layers. Inspired by Dense Prediction Transformers (DPT) [106], we tried upsampling features at multiple self-attention layers in the Perceiver Transformer. But this did not work at all; perhaps the latent-space self-attention layers of Perceiver are substantially different to the full-input self-attention layers of ViT [4] and DPT [106].

Extreme rotation augmentation. In addition to yaw rotation perturbations, we also tried perturbing the pitch and roll. While PERACT was still able to learn policies, it took substantially longer to train. It is also unclear if the default latent size of $\mathbb{R}^{2048 \times 512}$ is appropriate for learning 6-DoF polices with such extreme rotation perturbations.

Using Adam instead of LAMB. We tried training PERACT with the Adam [107] optimizer instead of LAMB [78], but this led to worse performance in both simulated and real-world experiments.

L Limitations and Risks

While PERACT is quite capable, it is not without limitations. In the following sections, we discuss some of these limitations and potential risks for real-world deployment.

Sampling-Based Motion Planner. PERACT relies on a sampling-based motion planner to execute discretized actions. This puts PERACT at the mercy of randomized planner to reach poses. While this issue did not cause any major problems with the tasks in our experiments, a lot of other tasks are sensitive to the paths taken to reach poses. For instance, pouring water into a cup would require a smooth path for tilting the water container appropriately. This could be addressed in future works by using a combination of learned and sampled motion paths [108].

Dynamic Manipulation. Another issue with discrete-time discretized actions is that they are not easily applicable to dynamic tasks that require real-time closed-loop maneuvering. This could be addressed with a separate visuo-servoing mechanism that can reach target poses with closed-loop control. Alternatively, instead of predicting just one action, PERACT could be extended to predict a sequence of discretized actions. Here, the Transformer-based architecture could be particularly advantageous. Also, instead of just predicting poses, the agent could also be trained to predict other physical parameters like target velocities [109].

Dexterous Manipulation. Using discretized actions with N-DoF robots like multi-fingered hands is also non-trivial. Specifically for multi-fingered hands, PERACT could be modified to predict fingertip poses that can be reached with an IK (Inverse Kinematics) solver. But it is unclear how feasible or robust such an approach would be with under-actuated systems like multi-fingered hands.

Generalization to Novel Instances and Objects. In Figure 11, we report results from small-scale perturbation experiments on the open drawer task. We observe that changing the shape of the handles does not affect performance. However, handles with randomized textures and colors confuse

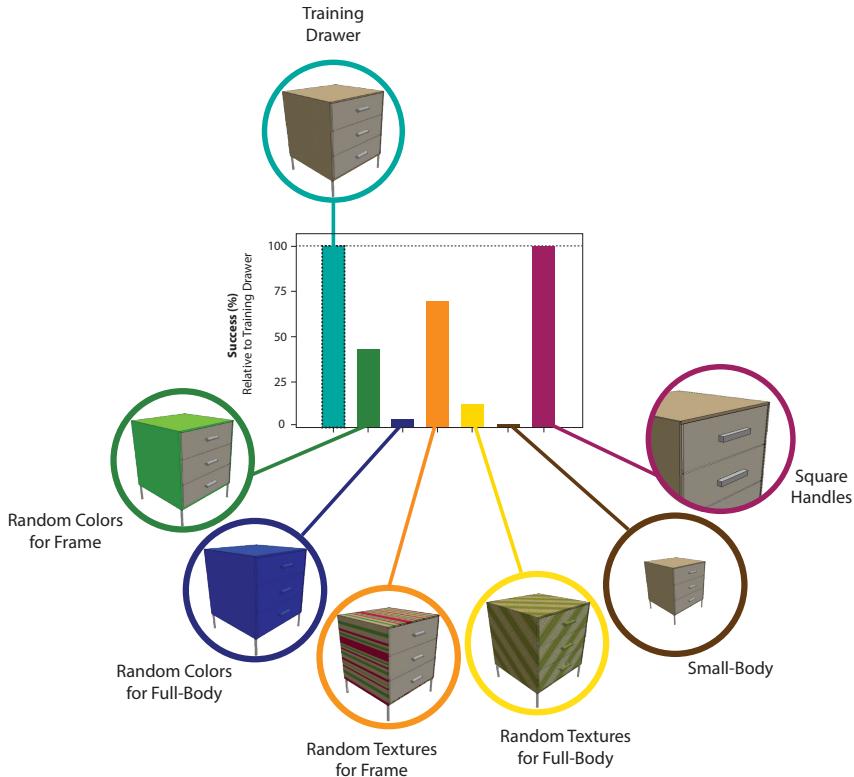


Figure 11. Perturbation Tests. Results from a multi-task PERACT agent trained on a single drawer and evaluated on several instances perturbed drawers. Each perturbation consists of 25 evaluation episodes, and reported successes are relative to the training drawer.

the agent since it has only seen one type of drawer color and texture during training. Going beyond this one-shot setting, and training on several instances of drawers might improve generalization performance. Although we did not explicitly study generalization to unseen objects, it might be feasible to train PERACT’s action-detector on a broad range of objects and evaluate its ability to handle novel objects, akin to how language-conditioned instance-segmentors and object-detectors are used [110]. Alternatively, pre-trained vision features from multi-modal encoders like CLIP [76] or R3M [38] could be used to bootstrap learning.

Scope of Language Grounding. Like with prior work [16], PERACT’s understanding of verb-noun phrases is closely grounded in demonstrations and tasks. For example, “cleaning” in “*clean the beans on the table with a dustpan*” is specifically associated with the action sequence of pushing beans on to a dustpan, and not “cleaning” in general, which could be applied to other tasks like cleaning the table with a cloth.

Predicting Task Completion. For both real-world and simulated evaluations, an oracle indicates whether the desired goal has been reached. This oracle could be replaced with a success classifier that can be pre-trained to predict task completion from RGB-D observations.

History and Partial Observability. PERACT relies purely on the current observation to predict the next action. As such, tasks that require history like counting or ordering are not feasible, unless accompanied by a task-completion predictor. Similarly, for tasks involving partial observability e.g., looking through drawers one-by-one for a specific object, PERACT does not keep track of what was seen before. Future works could include observations from previous timesteps, or append Perceiver latents, or train a Recurrent Neural Network to encode latents across timesteps.

Data Augmentation with Kinematic Feasibility. The data augmentation method described in Section E does not consider the kinematic feasibility of reaching perturbed actions with the Franka arm. Future works could pre-compute unreachable poses in the discretized action space, and discard any augmentation perturbations that push actions into unreachable zones.

Balanced Datasets. Since PERACT is trained with just a few demonstrations, it occasionally tends to exploit biases in the training data. For instance, PERACT might have a tendency to always “*place blue blocks on yellow blocks*” if such an example is over-represented in the training data. Such issues could be potentially fixed by scaling datasets to include more diverse examples of objects and attributes. Additionally, data visualization methods could be used to identify and fix these biases.

Multi-Task Optimization. The uniform task sampling strategy presented in Section 3.4 might sometimes hurt performance. Since all tasks are weighted equally, optimizing for certain tasks with common elements (e.g., moving blocks), might adversarial affect the performance on other dissimilar tasks (e.g., turning taps). Future works, could use dynamic task-weighting methods like Auto- λ [73] for better multi-task optimization.

Deployment Risks. PERACT is an end-to-end framework for 6-DoF manipulation. Unlike some methods in Task-and-Motion-Planning that can sometimes provide theoretical guarantees on task completion, PERACT is a purely reactive system whose performance can only be evaluated through empirical means. Also, unlike prior works [16], we do not use internet pre-trained vision encoders that might contain harmful biases [111, 112]. Even so, it is prudent to thoroughly study and mitigate any biases before deployment. As such, for real-world applications, keeping humans in the loop both during training and testing, might help. Usage with unseen objects and observations with people is not recommended for safety critical systems.

M Emergent Properties

In this section, we present some preliminary findings on the emergent properties of PERACT.

M.1 Object Tracking

Although PERACT was not explicitly trained for 6-DoF object-tracking, our action detection framework can be used to localize objects in cluttered scenes. In this [video](#), we show an agent that was trained with one hand sanitizer instance on just 5 “press the handsan” demos, and then evaluated on tracking an unseen sanitizer instance. PERACT does not need to build a complete representation of hand sanitizers, and only has to learn *where to press* them. Our implementation runs at an inference speed of 2.23 FPS (or 0.45 seconds per frame), allowing for near real-time closed-loop behaviors.

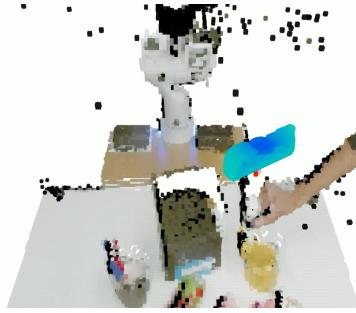


Figure 12. Object Tracker. Tracking an unseen hand sanitizer instance.

M.2 Multi-Modal Actions

PERACT’s problem formulation allows for modeling multi-modal action distributions, i.e., scenarios where multiple actions are valid given a specific goal. Figure 13 presents some selected examples of multi-modal action predictions from PERACT. Since there are several “yellow blocks” and “cups” to choose from, the Q -prediction distributions have several modes. In practice, we observe that the agent has a tendency to prefer certain object instances over others (like the front mug in Figure 13) due to preference biases in the training dataset. We also note that the cross-entropy based training method from Section 3.4 is closely related to Energy-Based Models (EBMs) [113, 114]. In a way, the cross-entropy loss is *pulling up* expert 6-DoF actions, while *pushing-down* every other action in the discretized action space. At test time, we simply maximize the learned Q -predictions, instead of minimizing an energy function with optimization. Future works could look into EBM [114] training and inference methods for better generalization and execution performance.

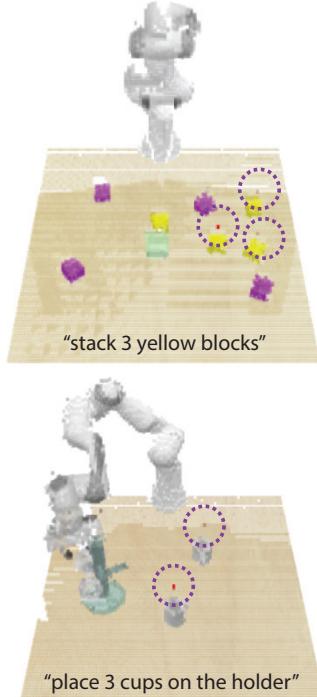


Figure 13. Examples of Multi-Modal Predictions.