PERCEIVER-ACTOR: 
A Multi-Task Transformer for Robotic Manipulation

Mohit Shridhar 1,* Lucas Manuelli 2 Dieter Fox 1,2

1University of Washington 2NVIDIA
msahr@cs.washington.edu lmanuelli@nvidia.com fox@cs.washington.edu

peract.github.io

Abstract: Transformers have revolutionized vision and natural language processing with their ability to scale with large datasets. But in robotic manipulation, data is both limited and expensive. Can manipulation still benefit from Transformers with the right problem formulation? We investigate this question with PERACT, a language-conditioned behavior-cloning agent for multi-task 6-DoF manipulation. PERACT encodes language goals and RGB-D voxel observations with a Perceiver Transformer [1], and outputs discretized actions by “detecting the next best voxel action”. Unlike frameworks that operate on 2D images, the voxelized 3D observation and action space provides a strong structural prior for efficiently learning 6-DoF actions. With this formulation, we train a single multi-task Transformer for 18 RLBench tasks (with 249 variations) and 7 real-world tasks (with 18 variations) from just a few demonstrations per task. Our results show that PERACT significantly outperforms unstructured image-to-action agents and 3D ConvNet baselines for a wide range of tabletop tasks.

Keywords: Transformers, Language Grounding, Manipulation, Behavior Cloning

1 Introduction

Transformers [2] have become prevalent in natural language processing and computer vision. By framing problems as sequence modeling tasks, and training on large amounts of diverse data, Transformers have achieved groundbreaking results in several domains [3, 4, 5, 6]. Even in domains that do not conventionally involve sequence modeling [7, 8], Transformers have been adopted as a general architecture [9]. But in robotic manipulation, data is both limited and expensive. Can we still bring the power of Transformers to 6-DoF manipulation with the right problem formulation?

Language models operate on sequences of tokens [10], and vision transformers operate on sequences of image patches [4]. While pixel transformers [11, 1] exist, they are not as data efficient as approaches that use convolutions or patches to exploit the 2D structure of images. Thus, while Transformers may be domain agnostic, they still require the right problem formulation to be data efficient. A similar efficiency issue is apparent in behavior-cloning (BC) agents that directly map 2D images to 6-DoF actions. Agents like Gato [9] and BC-Z [12, 13] have shown impressive multi-task capabilities, but they require several weeks or even months of data collection. In contrast, recent works in reinforcement-learning like C2FARM [14] construct a voxelized observation and action space to efficiently learn visual representations of 3D actions with 3D ConvNets. Similarly, in this work, we aim to exploit the 3D structure of voxel patches for efficient 6-DoF behavior-cloning with Transformers (analogous to how vision transformers [4] exploit the 2D structure of image patches).

To this end, we present PERACT (short for PERCEIVER-ACTOR), a language-conditioned BC agent that can learn to imitate a wide variety of 6-DoF manipulation tasks with just a few demonstrations per task. PERACT encodes a sequence of RGB-D voxel patches and predicts discretized translations, rotations, and gripper actions that are executed with a motion-planner in an observe-act loop. PERACT is essentially a classifier trained with supervised learning to detect actions akin to prior work like CLIPort [16, 17], except our observations and actions are represented with 3D voxels instead of 2D image pixels. Voxel grids are less prevalent than images in end-to-end BC approaches often due

*Work done partly while the author was a part-time intern at NVIDIA.

6th Conference on Robot Learning (CoRL 2022), Auckland, New Zealand.
to scaling issues with high-dimensional inputs. But in PERACT, we use a Perceiver² Transformer [1] to encode very high-dimensional input of up to 1 million voxels with only a small set of latent vectors. This voxel-based formulation provides a strong structural prior with several benefits: a natural method for fusing multi-view observations, learning robust action-centric³ representations [18, 19], and enabling data augmentation in 6-DoF – all of which help learn generalizable skills by focusing on diverse rather than narrow multi-task data.

To study the effectiveness of this formulation, we conduct large-scale experiments in the RL-Bench [15] environment. We train a single multi-task agent on 18 diverse tasks with 249 variations that involve a range of prehensile and non-prehensile behaviors like placing wine bottles on a rack and dragging objects with a stick (see Figure 1 a-j). Each task also includes several pose and semantic variations with objects that differ in placement, color, shape, size, and category. Our results show that PERACT significantly outperforms image-to-action agents (by $34 \times$) and 3D ConvNet baselines (by $2.8 \times$), without using any explicit representations of instance segmentations, object poses, memory, or symbolic states. We also validate our approach on a Franka Panda with a multi-task agent trained from scratch on 7 real-world tasks with a total of just 53 demonstrations (see Figure 1 k-o).

In summary, our contributions are as follows:

- **A novel problem formulation** for perceiving, acting, and specifying goals with Transformers.
- An efficient action-centric framework for grounding language in 6-DoF actions.
- **Empirical results** investigating multi-task agents on a range of simulated and real-world tasks.

The code and pre-trained models will be made available at [peract.github.io](https://peract.github.io).

---

²Throughout the paper we refer to PerceiverIO [1] as Perceiver for brevity.
³Action-centric refers to a system that learns perceptual representations of actions; see Appendix J.
2 Related Work

Vision for Manipulation. Traditionally, methods in robot perception have used explicit “object” representations like instance segmentations, object classes, poses [20, 21, 22, 23, 24, 25]. Such methods struggle with deformable and granular items like cloths and beans that are hard to represent with geometric models or segmentations. In contrast, recent methods [26, 17, 16, 27] learn action-centric representations without any “objectness” assumptions, but they are limited to top-down 2D settings with simple pick-and-place primitives. In 3D, James et al. proposed C2FARM [14], an action-centric reinforcement learning (RL) agent with a coarse-to-fine-grain 3D-UNet backbone. The coarse-to-fine-grain scheme has a limited receptive field that cannot look at the entire scene at the finest level. In contrast, PERACT learns action-centric representations with a global-receptive field through a Transformer backbone. Also, PERACT does BC instead of RL, which enables us to easily train a multi-task agent for several tasks by conditioning it with language goals.

End-to-End Manipulation approaches [28, 29, 30, 31] make the least assumptions about objects and tasks, but are often formulated as an image-to-action prediction task. Training directly on RGB images for 6-DoF tasks is often inefficient, generally requiring several demonstrations or episodes just to learn basic skills like rearranging objects. In contrast, PERACT uses a voxelized observation and action space, which is dramatically more efficient and robust in 6-DoF settings. While other works in 6-DoF grasping [32, 33, 34, 35, 36, 37] have used RGB-D and pointcloud input, they have not been applied to sequential tasks or used with language-conditioning. Another line of work tackles data inefficiency by using pre-trained image representations [16, 38, 39] to bootstrap BC. Although our framework is trained from scratch, such pre-training approaches could be integrated together in future works for even greater efficiency and generalization to unseen objects.

Transformers for Agents and Robots. Transformers have become the prevalent architecture in several domains. Starting with NLP [2, 3, 40], recently in vision [4, 41], and even RL [8, 42, 43]. In robotics, Transformers have been applied to assistive teleop [44], legged locomotion [45], path-planning [46, 47], imitation learning [48, 49], morphology controllers [50], spatial rearrangement [51], and grasping [52]. Transformers have also achieved impressive results in multi-domain settings like in Gato [9] where a single Transformer was trained on 16 domains such as captioning, language-grounding, robotic control etc. However, Gato relies on extremely large datasets like 15K episodes for block stacking and 94K episodes for Meta-World [53] tasks. Our approach might complement agents like Gato, which could use our 3D formulation for greater efficiency and robustness.

Language Grounding for Manipulation. Several works have proposed methods for grounding language in robot actions [54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65]. However, these methods use disentangled pipelines for perception and action, with the language primarily being used to guide perception [66]. Recently, a number of end-to-end approaches [13, 12, 67, 68, 69] have been proposed for conditioning BC agents with language instructions. These methods require thousands of human demos or autonomous episodes that are collected over several days or even months. In contrast, PERACT can learn robust multi-task policies with just a few minutes of training data. For benchmarking, several simulation environments exist [70, 17, 53], but we use RLBench [15] for its diversity of 6-DoF tasks and ease of generating demonstrations with templated language goals.

3 PERCEIVER-ACTOR

PERACT is a language-conditioned behavior-cloning agent for 6-DoF manipulation. The key idea is to learn perceptual representations of actions conditioned on language goals. Given a voxelized reconstruction of a scene, we use a Perceiver Transformer [1] to learn per-voxel features. Despite the extremely large input space (100^3), Perceiver uses a small set of latent vectors to encode the input. The per-voxel features are then used to predict the next best action in terms of discretized translation, rotation, and gripper state at each timestep. PERACT relies purely on the current observation to determine what to do next in sequential tasks. See Figure 2 for an overview.

Section 3.1 and Section 3.2 describe our dataset setup. Section 3.3 describes our problem formulation with PERACT, and Section 3.4 provides details on training PERACT. Further implementation details are presented in Appendix B.
3.1 Demonstrations

We assume access to a dataset \( D = \{ \zeta_1, \zeta_2, \ldots, \zeta_n \} \) of \( n \) expert demonstrations, each paired with English language goals \( G = \{ l_1, l_2, \ldots, l_n \} \). These demonstrations are collected by an expert with the aid of a motion-planner to reach intermediate poses. Each demonstration \( \zeta \) is a sequence of continuous actions \( A = \{ a_1, a_2, \ldots, a_l \} \) paired with observations \( O = \{ o_1, o_2, \ldots, o_l \} \). An action \( a \) consists of the 6-DoF pose, gripper open state, and whether the motion-planner used collision avoidance to reach an intermediate pose: \( a = \{ a_{\text{pose}}, a_{\text{open}}, a_{\text{collide}} \} \). An observation \( o \) consists of RGB-D images from any number of cameras. We use four cameras for simulated experiments \( o_{\text{sim}} = \{ o_{\text{front}}, o_{\text{left}}, o_{\text{right}}, o_{\text{wrist}} \} \), but just a single camera for real-world experiments \( o_{\text{real}} = \{ o_{\text{front}} \} \).

3.2 Keyframes and Voxelization

Following prior work by James et al. [14], we construct a structured observation and action space through keyframe extraction and voxelization.

Training our agent to directly predict continuous actions is inefficient and noisy. So instead, for each demonstration \( \zeta \), we extract a set of keyframe actions \( \{ k_1, k_2, \ldots, k_m \} \subset A \) that capture bottleneck end-effector poses \( k \) through triangulation \( \delta \Rightarrow v \) from known camera extrinsics and intrinsics. By default, we use a voxel grid of \( 100^3 \), which corresponds to a volume of \( 1.0 \text{m}^3 \) in metric scale. The keyframe actions \( k \) are discretized such that training our BC agent can be formulated as a “next best action” classification task [14]. Translation is simply the closest voxel to the center of the gripper fingers. Rotation is discretized into 5 degree bins for each of the three rotation axes. Gripper open state is a binary value. Collide is also a binary value that indicates if the motion-planner should avoid everything in the voxel grid or nothing at all; switching between these two modes of collision avoidance is crucial as tasks often involve both contact based (e.g., pulling the drawer open) and non-contact based motions (e.g., reaching the handle without colliding into anything).

3.3 PERACT Agent

PERACT is a Transformer-based [2] agent that takes in a voxel observation and language goal \( (v, l) \), and outputs a discretized translation, rotation, and gripper open action. This action is executed with a motion-planner, after which this process is repeated until the goal is reached.

The language goal \( l \) is encoded with a pre-trained language model. We use CLIP’s [76] language encoder, but any pre-trained language model would suffice [13, 69]. Our choice of CLIP opens up possibilities for future work to use pre-trained vision features that are aligned with the language for better generalization to unseen semantic categories and instances [16].
The voxel observation $v$ is split into 3D patches of size $5^3$ (akin to vision-transformers like ViT [4]). In implementation, these patches are extracted with a 3D convolution layer with a kernel-size and stride of 5, and then flattened into a sequence of voxel encodings. The language encodings are fine-tuned with a linear layer and then appended with the voxel encodings to form the input sequence. We also add learned positional embeddings to the sequence to incorporate voxel and token positions.

The input sequence of language and voxel encodings is extremely long. A standard Transformer with $O(n^2)$ self-attention connections and an input of $(100/5)^3 = 8000$ patches is hard to fit on the memory of a commodity GPU. Instead, we use the Perceiver [1] Transformer. Perceiver is a latent-space Transformer, where instead of attending to the entire input, it first computes cross-attention between the input and a much smaller set of latent vectors (which are randomly initialized and trained). These latents are encoded with self-attention layers, and for the final output, the latents are again cross-attended with the input to match the input-size. See Appendix Figure 6 for an illustration. By default, we use 2048 latents of dimension 512 : $\mathbb{R}^{2048 \times 512}$, but in Appendix G we experiment with different latent sizes.

The Perceiver Transformer uses 6 self-attention layers to encode the latents and outputs a sequence of patch encodings from the output cross-attention layer. These patch encodings are upsampled with a 3D convolution layer and tri-linear upsampling to decode 64-dimensional voxel features. The decoder includes a skip-connection from the encoder (like in UNets [77]). The per-voxel features are then used to predict discretized actions [14]. For translation, the voxel features are reshaped into the original voxel grid $(100^3)$ to form a 3D $Q$-function of action-values. For rotation, gripper open, and collide, the features are max-pooled and then decoded with linear layers to form their respective $Q$-function. The best action $\mathcal{T}$ is chosen by simply maximizing the $Q$-functions:

$$
T_{\text{trans}} = \arg\max_{(x,y,z)} Q_{\text{trans}}((x,y,z) | v, 1), \quad T_{\text{rot}} = \arg\max_{(\psi, \theta, \phi)} Q_{\text{rot}}((\psi, \theta, \phi) | v, 1),
$$

$$
T_{\text{open}} = \arg\max_{\omega} Q_{\text{open}}(\omega | v, 1), \quad T_{\text{collide}} = \arg\max_{\kappa} Q_{\text{collide}}(\kappa | v, 1),
$$

where $(x, y, z)$ is the voxel location in the grid, $(\psi, \theta, \phi)$ are discrete rotations in Euler angles, $\omega$ is the gripper open state and $\kappa$ is the collide variable. See Figure 5 for examples of $Q$-predictions.

### 3.4 Training Details

PerAct is trained through supervised learning with discrete-time input-action tuples from a dataset of demonstrations. These tuples are composed of voxel observations, language goals, and keyframe actions $\{(v_1, l_1, k_1), (v_2, l_2, k_2), \ldots\}$. During training, we randomly sample a tuple and supervise the agent to predict the keyframe action $k$ given the observation and goal $(v, l)$. For translation, the ground-truth action is represented as a one-hot voxel encoding $Y_{\text{trans}} : \mathbb{R}^{H \times W \times D}$. Rotations are also represented with a one-hot encoding per rotation axis with $R$ rotation bins $Y_{\text{rot}} : \mathbb{R}^{(360/R) \times 3}$ ($R = 5$ degrees for all experiments). Similarly, open and collide variables are binary one-hot vectors $Y_{\text{open}} : \mathbb{R}^2, Y_{\text{collide}} : \mathbb{R}^2$. The agent is trained with cross-entropy loss like a classifier:

$$
L_{\text{total}} = -E_{Y_{\text{trans}}} [\log Y_{\text{trans}}] - E_{Y_{\text{rot}}} [\log Y_{\text{rot}}] - E_{Y_{\text{open}}} [\log Y_{\text{open}}] - E_{Y_{\text{collide}}} [\log Y_{\text{collide}}],
$$

where $Y_{\text{trans}} = \text{softmax}(Q_{\text{trans}}((x, y, z) | v, 1)), Y_{\text{rot}} = \text{softmax}(Q_{\text{rot}}((\psi, \theta, \phi) | v, 1)), Y_{\text{open}} = \text{softmax}(Q_{\text{open}}(\omega | v, 1)), Y_{\text{collide}} = \text{softmax}(Q_{\text{collide}}(\kappa | v, 1))$ respectively. For robustness, we also augment $v$ and $k$ with translation and rotation perturbations. See Appendix E for more details.

By default, we use a voxel grid size of $100^3$. We conducted validation tests by replaying expert demonstrations with discretized actions to ensure that $100^3$ is a sufficient resolution for execution. The agent was trained with a batch-size of 16 on 8 NVIDIA V100 GPUs for 16 days (600K iterations). We use the LAMB [78] optimizer following Perceiver [1].

For multi-task training, we simply sample input-action tuples from all tasks in the dataset. To ensure that tasks with longer horizons are not over-represented during sampling, each batch contains a uniform distribution of tasks. That is, we first uniformly sample a set of tasks of batch-size length, then pick a random input-action tuple for each of the sampled tasks. With this strategy, longer-horizon tasks need more training steps for full coverage of input-action pairs, but all tasks are given equal weighting during gradient updates.
4 Results

We perform experiments to answer the following questions: (1) How effective is PERACT compared to unstructured image-to-action frameworks and standard architectures like 3D ConvNets? And what are the factors that affect PERACT’s performance? (2) Is the global receptive field of Transformers actually beneficial over methods with local receptive fields? (3) Can PERACT be trained on real-world tasks with noisy data?

4.1 Simulation Setup

We conduct our primary experiments in simulation for the sake of reproducibility and benchmarking.

Environment. The simulation is set in CoppelaSim [79] and interfaced through PyRep [80]. All experiments use a Franka Panda robot with a parallel gripper. The input observations are captured from four RGB-D cameras positioned at the front, left shoulder, right shoulder, and on the wrist, as shown in Appendix Figure 7. All cameras are noiseless and have a resolution of $128 \times 128$.

Language-Conditioned Tasks. We train and evaluate on 18 RL Bench [15] tasks. See peract.github.io for examples and Appendix A for details on individual tasks. Each task includes several variations, ranging from 2-60 possibilities, e.g., in the stack blocks task, “stack 2 red blocks” and “stack 4 purple blocks” are two variants. These variants are randomly sampled during data generation, but kept consistent during evaluations for one-to-one comparisons. Some RL Bench tasks were modified to include additional variations to stress-test multi-task and language-grounding capabilities. There are a total of 249 variations across 18 tasks, and the number of extracted keyframes range from 2-17. All keyframes from an episode have the same language goal, which is constructed from templates (but human-annotated for real-world tasks). Note that in all experiments, we do not test for generalization to unseen objects, i.e., our train and test objects are the same. However during test time, the agent has to handle novel object poses, randomly sampled goals, and randomly sampled scenes with different semantic instantiations of object colors, shapes, sizes, and categories. The focus here is to evaluate the performance of a single multi-task agent trained on all tasks and variants.

Evaluation Metric. Each multi-task agent is evaluated independently on all 18 tasks. Evaluations are scored either 0 for failures or 100 for complete successes. There are no partial credits. We report average success rates on 25 evaluation episodes per task ($25 \times 18 = 450$ total episodes) for agents trained with $n = 10$, 100 demonstrations per task. During evaluation, an agent keeps taking actions until an oracle indicates task-completion or reaches a maximum of 25 steps.

4.2 Simulation Results

Table 1 reports success rates of multi-task agents trained on all 18 tasks. We could not investigate single-task agents due to resource constraints of training 18 individual agents.

Baseline Methods. We study the effectiveness of our problem formulation by benchmarking against two language-conditioned baselines: Image-BC and C2FARM-BC. Image-BC is an image-to-action agent similar to BC-Z [12]. Following BC-Z, we use FiLM [81] for conditioning with CLIP [76] language features, but the vision encoders take in RGB-D images instead of just RGB. We also study both CNN and ViT vision encoders. C2FARM-BC is a 3D fully-convolutional network by James et al. [14] that has achieved state-of-the-art results on RL Bench tasks. Similar to our agent, C2FARM-BC also detects actions in a voxelized space, however it uses a coarse-to-fine-grain scheme to detect actions at two-levels of voxelization: $3^2 \times$ voxels with a $1^4$m grid, and $3^2 \times$ voxels with a 0.15$m$ grid after “zooming in” from the first level. Note that at the finest level, C2FARM-BC has a higher resolution (0.47cm) than PERACT (1cm). We use the same 3D ConvNet architecture as James et al. [14], but instead of training it with RL, we do BC with cross-entropy loss (from Section 3.4). We also condition it with CLIP [76] language features at the bottleneck like in LingUNets [82, 16].

Multi-Task Performance. Table 1 compares the performance of Image-BC and C2FARM-BC against PERACT. With insufficient demonstrations, Image-BC has near zero performance on most tasks. Image-BC is disadvantaged with single-view observations and has to learn hand-eye coordination from scratch. In contrast, PERACT’s voxel-based formulation naturally allows for integrating multi-view observations, learning 6-DOF action representations, and data-augmentation in 3D, all of which are non-trivial to achieve in image-based methods. C2FARM-BC is the most competitive baseline, but it has a limited receptive field mostly because of the coarse-to-fine-grain scheme and partly due to the convolution-only architecture. PERACT outperforms C2FARM-BC in
25/36 evaluations in Table 1 with an average improvement of 1.33× with 10 demonstrations and 2.83× with 100 demonstrations. For a number of tasks, C2FARM-BC actually performs worse with more demonstrations, likely due to insufficient capacity. Since additional training demonstrations include additional task variants to optimize for, they might end up hurting performance.

In general, 10 demonstrations are sufficient for PERAct to achieve >65% success on tasks with limited variations like open drawer (3 variations). But tasks with more variations like stack blocks (60 variations) need substantially more data, sometimes to simply cover all possible concepts like “teal color block” that might have not appeared in the training data. See the simulation rollouts in the supplementary video to get a sense of the complexity of these evaluations. For three tasks: insert peg, stack cups, and place cups, all agents achieve near zero success. These are very high-precision tasks where being off by a few centimeters or degrees could lead to unrecoverable failures. But in Appendix H we find that training single-task agents, specifically for these tasks, slightly alleviates this issue.

Ablations. Table 1 reports PERAct w/o Lang, an agent without any language conditioning. Without a language goal, the agent does not know the underlying task and performs at chance. We also report additional ablation results on the open drawer task in Figure 3. To summarize these results: (1) the skip connection helps train the agent slightly faster, (2) the Perceiver Transformer is crucial for achieving good performance with the global receptive field, and (3) extracting good keyframes actions is essential for supervised training as randomly chosen or fixed-interval keyframes lead to zero-performance.

Sensitivity Analysis. In Appendix G we investigate factors that affect PERAct’s performance: the number of Perceiver latents, voxelization resolution, and data augmentation. We find that more latent vectors generally improve the capacity of the agent to model more tasks, but for simple short-horizon tasks, fewer latents are sufficient. Similarly, with different voxelization resolutions, some tasks are solvable with coarse voxel grids like 32³, but some high-precision tasks require the full 100³ grid. Finally, rotation perturbations in the data augmentation generally help in improving robustness essentially by exposing the agent to more rotation variations of objects.

4.3 Global vs. Local Receptive Fields

To further investigate our Transformer agent’s global receptive field, we conduct additional experiments on the open drawer task. The open drawer task has three variants: “open the top drawer”, “open the middle drawer”, and “open the bottom drawer”, and with a limited receptive field it is hard to distinguish the drawer handles, which are all visually identical. Figure 4 reports PERAct and C2FARM-BC agents trained with 100 demonstrations. Although the open drawer tasks can be
solved with fewer demonstrations, here we want to ensure that insufficient data is not an issue. We include several versions of C2FARM-BC with different voxelization schemes. For instance, [16, 16] indicates two levels of $16 \times 16$ voxel grids at $1m^3$ and $0.15m^3$, respectively. And [64] indicates a single level of a $64 \times 3$ voxel grid without the coarse-to-fine-grain scheme. PERACT is the only agent that achieves > 70% success, whereas all C2FARM-BC versions perform at chance with $\sim 33\%$, indicating that the global receptive field of the Transformer is crucial for solving the task.

### 4.4 Real-Robot Results

We also validated our results with real-robot experiments on a Franka Emika Panda. See Appendix D for setup details. Without any sim-to-real transfer or pre-training, we trained a multi-task PERACT agent from scratch on 7 tasks (with 18 unique variations) from a total of just 53 demonstrations. See the supplementary video for qualitative results that showcase the diversity of tasks and robustness to scene changes. Table 2 reports success rates from small-scale evaluations. Similar to the simulation results, we find that PERACT is able to achieve > 65% success on simple short-horizon tasks like pressing hand-sanitizers from just a handful number of demonstrations. The most common failures involved predicting incorrect gripper open actions, which often lead the agent into unseen states. This could be addressed in future works by using HG-DAgger style approaches to correct the agent [12]. Other issues included the agent exploiting biases in the dataset like in prior work [16]. This could be addressed by scaling up expert data with more diverse tasks and task variants.

### 5 Limitations and Conclusion

We presented PERACT, a Transformer-based multi-task agent for 6-DoF manipulation. Our experiments with both simulated and real-world tasks indicate that the right problem formulation, i.e., detecting voxel actions, makes a substantial difference in terms of data efficiency and robustness.

While PERACT is quite capable, extending it to dexterous continuous control remains a challenge. PERACT is at the mercy of a sampling-based motion-planner to execute discretized actions, and is not easily extendable to N-DoF actuators like multi-fingered hands. See Appendix L for an extended discussion on PERACT’s limitations. But overall, we are excited about scaling up robot learning with Transformers by focusing on diverse rather than narrow multi-task data for robotic manipulation.

### Acknowledgments

We thank Selest Nashef and Karthik Desingh for their help with the Franka setup at UW. We thank Stephen James for helping with RL Bench and ARM issues. We are also grateful to Valts Blukis, Zoey Chen, Markus Grotz, Aaron Walsman, and Kevin Zakka, for providing feedback on the initial draft. And thanks to Shikhar Bahl for initial discussions. This work was funded in part by ONR under award #1140209-405780. Mohit Shridhar is supported by the NVIDIA Graduate Fellowship, and was also a part-time intern at NVIDIA throughout the duration of this project.
References


## A Task Details

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

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 PERACT could be easily applied to more RLBench tasks, in our experiments, we were specifically interested in grounding diverse language instructions, rather than learning one-off policies for single-variation tasks like “[always] take off the saucepan lid”\(^4\). 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: \(\text{colors} = \{\text{red}, \text{maroon}, \text{lime}, \text{green}, \text{blue}, \text{navy}, \text{yellow}, \text{cyan}, \text{magenta}, \text{silver}, \text{gray}, \text{orange}, \text{olive}, \text{purple}, \text{teal}, \text{azure}, \text{violet}, \text{rose}, \text{black}, \text{white}\}\). The set of sizes include 2 instances: \(\text{sizes} = \{\text{short}, \text{tall}\}\). The set of shapes include 5 instances: \(\text{shapes} = \{\text{cube}, \text{cylinder}, \text{triangle}, \text{star}, \text{moon}\}\). The set of counts include 3 instances: \(\text{counts} = \{1, 2, 3\}\). The placements and object categories are specific to each task. For instance, open drawer has 3 placement locations: top, middle, and bottom. 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] codebase\(^4\) 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 $90^\circ$ 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](https://colab.research.google.com/github/MMIC-LRG/Paper-Code/blob/main/colab_tutorial.ipynb) for a PyTorch implementation.

**Input Observation.** Following James et al. [14], our input voxel observation is a $100^3$ voxel grid with 10 channels: $\mathbb{R}^{100\times100\times10}$. The grid is constructed by fusing calibrated pointclouds with PyTorch’s `scatter_` function\(^3\). 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 $\mathbb{R}^{77\times512}$.

**Preprocessing.** The voxel grid is encoded with a 3D convolution layer with a $1\times1$ 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 $\mathbb{R}^{20\times20\times64}$\(^5\). 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 $\mathbb{R}^{20\times20\times128}$\(^6\). This tensor is flattened into a sequence of dimensions $\mathbb{R}^{8000\times128}$. 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 $\mathbb{R}^{8077\times128}$.

We also add learned positional embeddings to the input sequence. These embeddings are represented with trainable `nn.Parameter`\(^7\)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 $\mathbb{R}^{2048\times512}$. 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 output 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: $\mathbb{R}^{100\times100\times64}$. 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 $1\times1$ kernel

---


---

20
downsamples the channels from 128 back to 64 dimensions. Our implementation of Perceiver is based on an existing open-source repository\(^6\).

**Decoding.** For translation, the voxel feature grid is decoded with a 3D convolution layer with a \(1 \times 1\) 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 repository\(^7\) 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 RLBench [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 $512 \times 424$ at $30$Hz. The extrinsics between the camera and robot base-frame are calibrated with the easy_handeye package$^8$. We use an ARUCO$^9$ 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 RViz$^{10}$ 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 MoveIt$^{11}$, 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 $\pm 0.125$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.

$^8$https://github.com/IFL-CAMP/easy_handeye
$^9$https://github.com/pal-robotics/aruco_ros
$^{10}$http://wiki.ros.org/rviz
$^{11}$http://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 \( v \) and their corresponding keyframe actions \( k \) are perturbed with random translations and rotations. Translation perturbations have a range of \([±0.125\text{m}, ±0.125\text{m}, ±0.125\text{m}]\). Rotation perturbations are limited to the yaw axis and have a range of \([0^\circ, 0^\circ, ±45^\circ]\). The 45\(^\circ\) 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.

### 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\(^\circ\) 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.

<table>
<thead>
<tr>
<th>Task</th>
<th>80</th>
<th>72</th>
<th>56</th>
<th>84</th>
<th>80</th>
<th>68</th>
<th>60</th>
<th>68</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERACT w/o Rot Aug</td>
<td>92</td>
<td>72</td>
<td>56</td>
<td>92</td>
<td>96</td>
<td>60</td>
<td>56</td>
<td>100</td>
<td>8</td>
</tr>
<tr>
<td>PERACT 4096 latents</td>
<td>84</td>
<td>88</td>
<td>44</td>
<td>68</td>
<td>84</td>
<td>48</td>
<td>48</td>
<td>84</td>
<td>12</td>
</tr>
<tr>
<td>PERACT 1024 latents</td>
<td>84</td>
<td>48</td>
<td>52</td>
<td>84</td>
<td>84</td>
<td>52</td>
<td>32</td>
<td>92</td>
<td>12</td>
</tr>
<tr>
<td>PERACT 512 latents</td>
<td>92</td>
<td>84</td>
<td>48</td>
<td>100</td>
<td>92</td>
<td>32</td>
<td>32</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>PERACT 64(^3) voxels</td>
<td>88</td>
<td>72</td>
<td>80</td>
<td>60</td>
<td>84</td>
<td>36</td>
<td>40</td>
<td>84</td>
<td>32</td>
</tr>
<tr>
<td>PERACT 32(^3) voxels</td>
<td>28</td>
<td>44</td>
<td>100</td>
<td>60</td>
<td>72</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>PERACT 7(^3) patches</td>
<td>72</td>
<td>48</td>
<td>96</td>
<td>92</td>
<td>76</td>
<td>76</td>
<td>36</td>
<td>96</td>
<td>32</td>
</tr>
<tr>
<td>PERACT 9(^3) patches</td>
<td>68</td>
<td>64</td>
<td>56</td>
<td>52</td>
<td>96</td>
<td>56</td>
<td>36</td>
<td>92</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>24</th>
<th>44</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>48</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERACT w/o Rot Aug</td>
<td>20</td>
<td>32</td>
<td>48</td>
<td>8</td>
<td>8</td>
<td>56</td>
<td>8</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>PERACT 4096 latents</td>
<td>32</td>
<td>44</td>
<td>52</td>
<td>8</td>
<td>12</td>
<td>72</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>PERACT 1024 latents</td>
<td>24</td>
<td>32</td>
<td>36</td>
<td>8</td>
<td>20</td>
<td>40</td>
<td>8</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>PERACT 512 latents</td>
<td>48</td>
<td>40</td>
<td>36</td>
<td>24</td>
<td>16</td>
<td>32</td>
<td>12</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>PERACT 64(^3) voxels</td>
<td>24</td>
<td>48</td>
<td>44</td>
<td>12</td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>PERACT 32(^3) voxels</td>
<td>12</td>
<td>20</td>
<td>52</td>
<td>0</td>
<td>0</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PERACT 7(^3) patches</td>
<td>8</td>
<td>48</td>
<td>76</td>
<td>0</td>
<td>12</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PERACT 9(^3) patches</td>
<td>12</td>
<td>36</td>
<td>72</td>
<td>12</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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.

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

<table>
<thead>
<tr>
<th>Task</th>
<th>Multi</th>
<th>Single</th>
</tr>
</thead>
<tbody>
<tr>
<td>place cups</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>stack cups</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>insert peg</td>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

## 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 pre-defined 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
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.

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.