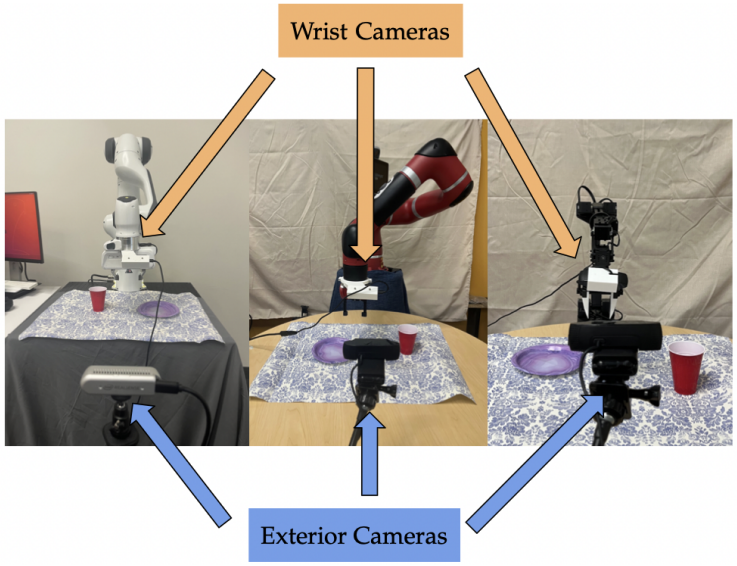
A Appendix

Further details and videos of experiments can be found on our project website: https://sites.google.com/view/cradle-multirobot

A.1 Robotic Setup



Our robot setups are shown above. For each robot, we collect data with both a wrist camera and exterior camera. The cameras are Logitech C920s and Zeds. Although these cameras do have slight differences in brightness and contrast, this does not seem to affect results..

A.2 Dataset Collection

We collect two types of datasets: a shared dataset containing data from similar tasks for all three robots and a target dataset containing a new task from one robot platform which we want to transfer to other platforms. For the purposes of evaluation, our shared dataset consists of the original tasks we defined above. For each variant, we collect data on 3 diverse scenes and backgrounds to ensure that the resulting policies have some degree of robustness to changes in the environment. In order to provide diversity to visual observations, we use cups, plates, and wallpapers with intricate patterns in the background. By collecting our datasets for each task over 3 variations of this scene, we ensure that our policy is robust to changes in lighting conditions. Overall, our dataset contains 6 tasks over 3 robots with 3 different backgrounds per task and 50 demonstrations per (scene, robot, background) combination collected over the course of 60 hours.

A.3 Higher-Level Environment Details

Our higher-level environment has of a shared image server and action processor between robots. We use delta position control as our action space. This is parameterized by a 7-dimensional vector consisting of 3 translational dimensions, 3 rotational dimensions, and 1 dimension indicate the percentage to close the parallel end-effector. The code for processing an action before sending it to the lower-level controller is shown below:

```
def step(self, action):
    start_time = time.time()

    # Process Action
    assert len(action) == (self.DoF + 1)
    assert (action.max() <= 1) and (action.min() >= -1)
```

```
        pos_action, angle_action, gripper = self._format_action(action)
        lin_vel, rot_vel = self._limit_velocity(pos_action, angle_action)
        desired_pos = self._curr_pos + lin_vel
        desired_angle = add_angles(rot_vel, self._curr_angle)

        self._update_robot(desired_pos, desired_angle, gripper)

        comp_time = time.time() - start_time
        sleep_left = max(0, (1 / self.hz) - comp_time)
        time.sleep(sleep_left)
```

Given a delta position, angle, and gripper command, our environment first normalized and clips the commands to ensure that large actions are not sent to the robot. Then, we add the delta position to our current pose and the delta angle to our current angle. We pass the position and angle into our lower-level robot controller.

A.4 Robot-Specific Controller Details

Each robot-specific controller provides the following API to the higher-level environment:

```
def update_pose(pos, angle):

def update_joints(joints):

def update_gripper(close_percentage):

def get_joint_positions():

def get_joint_velocities():

def get_gripper_state():

def get_ee_pose():
```

The functions update_pose, update_joints and update_gripper set targets for moving the robot. For each lower-level controller, we use a shared inverse kinematics solver to take the target poses in update_pose and convert them into joint targets. We also use our ik solver to give us joint positions, joint velocities, gripper states, and end-effector poses. For each robot, we implement two controllers: a blocking version and a nonblocking version. The blocking controller waits for an entire movement command to finish before executing the next command. Meanwhile, the nonblocking or continuous controller continuously interrupts the robot with a new target pose every fixed period of time.

A.5 Network Architecture

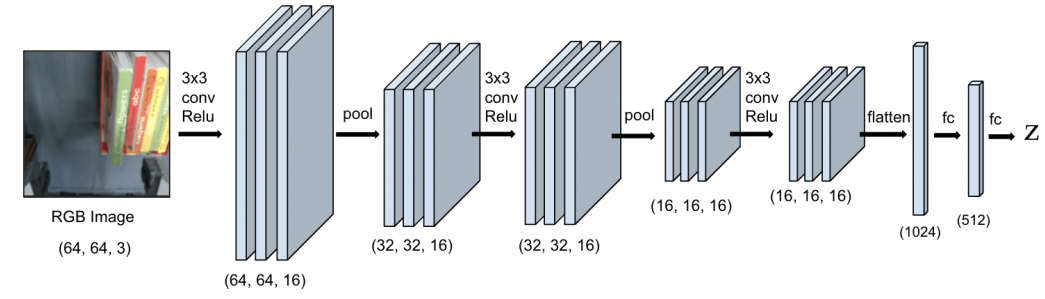


Figure 4: **Our encoder architecture.** We parameterize our encoder as a CNN. The convolutional layers are flattened and then fed into two MLP layers to get a representation z. In order to learn correspondence between robots, we train this encoder with a contrastive loss. We use random crop and color jitter as image augmentations for our encoder.

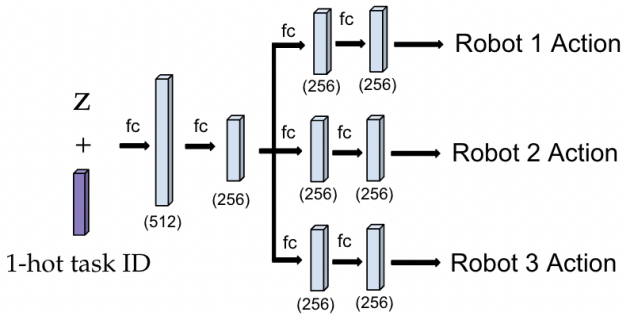


Figure 5: **Our decoder architecture.** The output of our encoder z is concatenated with a one-hot task index and fed into the decoder. This task index specifies a task either in the shared buffer, or a new task which we want to achieve. After passing the input through two MLP layers, we feed in into three-robot specific heads for each of the robots we are evaluating on.

Attribute	Value
Input Width	64
Input Height	64
Input Channels	3
Kernel Sizes	[3, 3, 3]
Number of Channels	[16, 16, 16]
Strides	[1, 1, 1]
Paddings	[1, 1, 1]
Pool Type	Max 2D
Pool Sizes	[2, 2, 1]
Pool Strides	[2, 2, 1]
Pool Paddings	[0, 0, 0]
Image Augmentation	Random Crops/Color Jitter
Image Augmentation Padding	4

Table 5: **CNN hyperparameters for our policy encoder.** Our CNN uses 64 by 64 images, which passes through through 3 convolutional layers. Each layer has a $3by3$ kernel with 16 channels. We augment our architecture with random crop and color jitter.

Hyperparameter	Value
Batch Size	64
Number of Gradient Updates Per Epoch	1000
Learning Rate	3E-4
Optimizer	Adam

Hyperparameter	Value
Batch Size	64
Number of Gradient Updates Per Epoch	1000
Learning Rate	1E-4
Optimizer	Adam

Table 6: **Hyperparameters.** The left table contains hyperparameters for behavior cloning, and the right table contains hyperparameters for contrastive learning.

A.6 Contrastive Learning Details

We train our encoder with a triplet loss of margin $m = 0.5$.

$$L(o_a, o_+, o_-) = \max(0, m + ||\tilde{f}_\theta(o_a) - \tilde{f}_\theta(o_+)||_2^2 - ||\tilde{f}_\theta(o_a) - \tilde{f}_\theta(o_-)||_2^2)$$

We provide nearest neighbor lookup for our robot below. We first embed the left image via our encoder. Then, we embed all observations in a dataset for a different robot. For example, in the top-left image, we use the shelf manipulation dataset with only Franka data. Then, we compute the embedding with the closest $l2$ distance from the embedding of the left image. Note that our method also aligns trajectories with same robot.

Figure 6: **Contrastive Nearest Neighbors.** This figure shows nearest neighbors examples across the three robots for embeddings from our pretrained encoder. These examples are computed for both shelf and pick/place trajectories.

A.7 Shelf Tasks

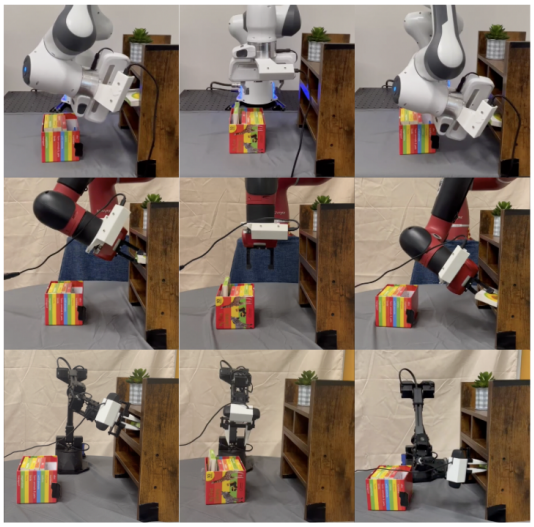


Figure 7: **Shelf Tasks.** The original shelf tasks consists of placing the book on the top compartment. The first target task requires doing the same from a reversed book container, while the second tasks requires placing the book in the lower compartment. The tasks in the first column are part of the shared dataset while the second and third are target tasks to test transfer.

A.8 Error between Commanded Delta Pose Target and Achieved Delta Pose

The following figures depict a plot of the $l2$ norm between the translational components of the delta commanded pose targets and achieved delta poses for demonstration trajectories across 3 robots. At each timestep, the environment receives a delta commanded pose target, which gets added to the robot's current pose then sent to the lower-level controller. Although the controller defines a trajectory to reach this target pose, due to errors in the inverse kinematics solver and limitations on movement imposed by the hardware, it may not reach the pose. We plot the error for each timestep across a trajectory from a Pick/Place task and one from a Shelf Manipulation task. Expectedly, the WidowX has the highest average error, followed by the Sawyer then the Franka. This error varies wildly between robots and timesteps, causing the commanded delta pose to be highly unpredictable from the achieved delta pose.

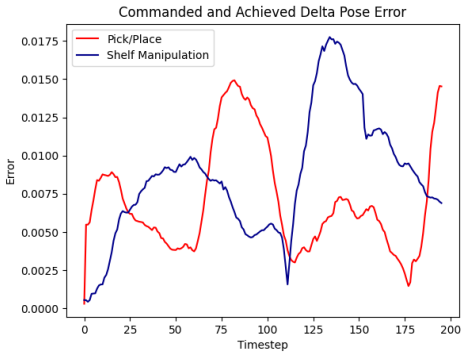


Figure 8: **Action interpretation error for the Franka.**

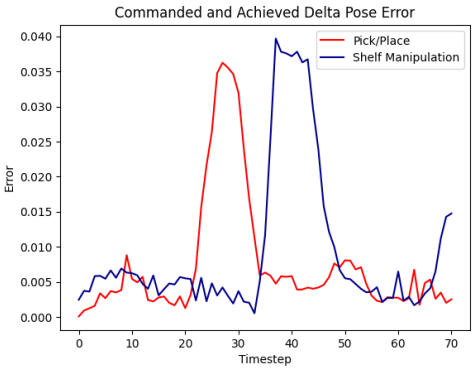


Figure 9: **Action interpretation Error for the Sawyer**

Figure 10: **Action interpretation error for the Widow.**