A Appendix

Videos of the experiments can be found on the project website: https://sites.google.com/view/ariel-berkeley

A.1 Implementation Details and Hyperparameters

We choose advantage-weighted actor-critic (AWAC) as the underlying RL algorithm for both the online and offline phases of ARIEL and all comparison methods because it addresses the specific challenges that arise when learning behaviors offline and then fine-tuning them online [1]. AWAC is an actor-critic method that alternates between a policy evaluation phase (Equation 1), where it trains a parameteric Q-function, $Q_\phi(\mathbf{s}, \mathbf{a})$, to minimize the error between two sides of the Bellman equation on the samples from the offline dataset, and a policy improvement phase (Equation 2) where it improves a parametric policy $\pi_\theta(\mathbf{a}|\mathbf{s})$ via advantage-weighted updates. Advantage-weighted updates perform policy improvement by cloning actions that are highly advantageous under the learned Q-function and are hence, more likely to improve upon the data-collection policy. Denoting the offline dataset as $\mathcal{D} = \{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}_{i=1}^N$, where $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ denotes a single transition, the training objectives for AWAC are given by:

$$\widehat{Q}^\pi_\phi \leftarrow \arg\min_\phi \; \mathbb{E}_{\mathbf{s},\mathbf{a},\mathbf{s}'\sim\mathcal{D}} \left[(Q_\phi(\mathbf{s}, \mathbf{a}) - (r + \gamma Q_\phi(\mathbf{s}', \mathbf{a}')))^2 \right] \tag{1}$$

$$\widehat{\pi} \leftarrow \arg\max_\theta \; \mathbb{E}_{\mathbf{s},\mathbf{a}\sim\mathcal{D}} \left[\log \pi_\theta(\mathbf{a}|\mathbf{s}) \cdot \exp(\widehat{A}^\pi(\mathbf{s}, \mathbf{a})) \right],$$

$$\text{where} \quad \widehat{A}^\pi(\mathbf{s}, \mathbf{a}) := \widehat{Q}^\pi_\phi(\mathbf{s}, \mathbf{a}) - \mathbb{E}_{\mathbf{a}'\sim\pi} \left[\widehat{Q}^\pi_\phi(\mathbf{s}, \mathbf{a}') \right]. \tag{2}$$

Our hyperparameters for AWAC are listed in Table 3. Notably, since AWAC is an off-policy RL algorithm, we have the option of continuing to train on the prior data during the online phase. We hypothesize that including the prior data during online adaptation prevents the agent from overfitting to the new task and improves generalization, so we continue to train on the prior data even during the online fine-tuning phase. As our goal is to make robotic learning as autonomous as possible, we use images observations as part of our input to the RL agent. Image observations allow our system to learn from a diverse range of different tasks without needing to hand-engineer state representations suitable for each task. Because we are learning from images we use convolutional neural networks to model the policy and Q-function. (see Figure 5 and Table 4). The observations additionally consist of the state of the robot's joints, and the task embedding $\mathbf{z}$. Lastly, during the online phase the pose of the robot is moved to a neutral position at the beginning of every trial, since there is no physical limitation that prevents doing this automatically, but the robot must still handle the fact that the objects in the scene maintain their position across trials. We now provide further details specific to each method:

- **Multi-Task RL**: We modify the policy and Q function architectures to accept two additional one-hot task indices. These task indices go unused during offline learning, but we use them to label the new data during online training.
- **R3L**: For the RND networks we use the same CNN architecture as the policy and Q function networks but set the output dimension to 5.
- **Oracle**: We learn a single-task policy with a single-task dataset consisting of 512 trajectories collected in the same manner as the rest of the simulated data.
- **ARIEL**: For CEM, we use a Gaussian mixture model as the sampling distributions with a number of components equal to the number of tasks in the prior data. In simulation, we update the sampling distributions every 10 trajectories, fitting them to the $J = 25$ most recent successful task embeddings. In the real world domains, we update the sampling distributions every 10 trajectories, fitting them to the $J = 10$ most recent successful task embeddings.

A.2 Real-World Experiments

In this section, we provide additional details on our real world experiments, the results for which were presented in Section 5.1. We utilize 3 different robotic setups for our experiments (Figure

Hyperparameter	Value
Target Network Update Frequency	1 step
Discount Factor γ	0.9666
Beta	0.01
Batch Size	64
Meta Batch Size	8
Soft Target τ	$5e^{-3}$
Policy Learning Rate	$3e^{-4}$
Q Function Learning Rate	$3e^{-4}$
Reward Scale	1.0
Alpha	0.0
Policy Weight Decay	$1e^{-4}$
Clip Score	0.5

Table 3: Hyperparameters for AWAC

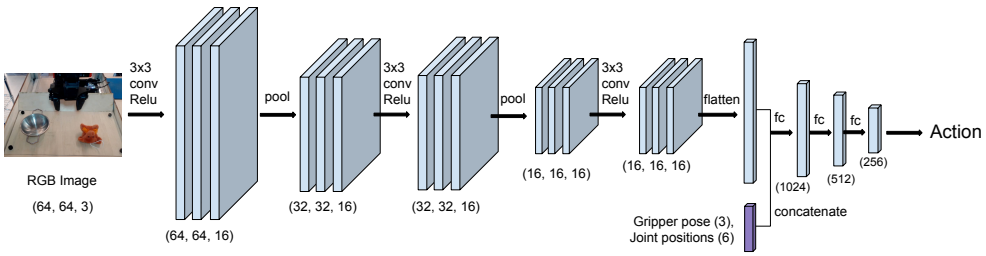

Figure 5: **CNN architecture we use for our policy and Q-function.** Our method learns robotic manipulation skills from raw image inputs. Here we show the architecture for the policy. For our Q-functions, we additionally concatenate the action along with the gripper pose and joint positions and set the output dimension to 1.

Attribute	Value
Input Width	48/64/128
Input Height	48/64/128
Input Channels	3
Kernel Sizes	[3, 3, 3]
Number of Channels	[16, 16, 16]
Strides	[1, 1, 1]
Fully Connected Layers	[1024, 512, 256]
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
Image Augmentation Padding	4

Table 4: CNN architecture for policy and Q-function networks. We use 48x48 images in simulation, 64x64 images in the **Tray Container** and **Tray Drawer** scenes, and 128x128 images in the **Kitchen** scene, however the rest of the architecture is the same across experiments.

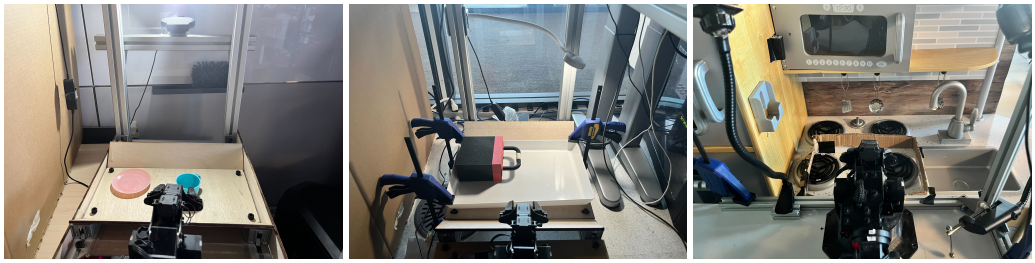

Figure 6: Robotic environments for real-world experiments

6). The first, denoted **Tray Container**, consists of a simple tray with objects and one of multiple containers. The second, denoted **Tray Drawer**, consists of a tray with objects and a 3D-printed drawer. The third, denoted **Kitchen**, is a toy kitchen to showcase the performance of the method under greater visual diversity.

A.2.1 Real-World Dataset Details

The real-robot dataset consists of picking and placing a variety of objects, including stuffed animals, rigid shapes, and toys, into a variety of containers, including plates, pots, baskets, and drawers. We utilize scripted policies in order to collect a large amount of data interacting with these objects. Since the prior dataset needs to initialize both the forward and backward directions for new tasks, we perform scripted collection in a reset-free manner, with the robot attempting to move the objects both into and out of containers. The **Tray Container** and **Kitchen** environments contain distractor objects that are not involved in the tasks so the robot is required to pay attention to the task embedding $\mathbf{z}$ to determine which object to interact with. In Table 5 we provide details on number of tasks, number of trajectories, and other dataset properties.

Attribute	Tray Container	Tray Drawer	Kitchen
Timesteps per Offline Trajectory	15	30	25
Timesteps per Online Trajectory	20	35	25
Forward Tasks	20	4	10
Backward Tasks	20	4	10
Number of Trajectories Per Task	500	150	50
Average Success Rate	0.35	0.93	0.47

Table 5: Real-world data details for the **Tray Container**, **Tray Drawer**, and **Kitchen** scenes.

A.2.2 Backward Controller Performance

In Table 6 we provide additional evaluation of the backward controller for the **Tray Container** and **Tray Drawer** scenes. We find that the performance of the backward controller improves throughout online training similar to the forward controller.

Task	**Offline Only**	**100 Trials**	**600 Trials**
Put Tiger in Drawer	5/10	6/10	**7/10**
Put Elephant in Pot	1/10	8/10	**10/10**
Put Tiger on Lid	2/10	**6/10**	**6/10**

Table 6: Real-world evaluation of the backward controller in the **Tray Container** and **Tray Drawer** scenes. These tasks use the direct transfer setting. Fine-tuning is mostly autonomous with a reset every 20-30 trials.

Target Task	Unseen Task	ARIEL by # Trials			Target Data Only
		100	360	600	Best Epoch
Put Tiger in Drawer	Put Pickle in Drawer	3/5	**5/5**	3/5	1/5
Put Tiger in Drawer	Put Turtle in Drawer	1/5	**4/5**	2/5	2/5
Put Tiger in Drawer	Put Dog in Drawer	2/5	2/5	**4/5**	2/5

Table 7: **Generalization.** We compare the zero-shot generalization performance on unseen objects of ARIEL and a baseline in the **Tray Drawer** scene. A policy pretrained on multi-task prior data and fine-tuned on the *put tiger in drawer* task (ARIEL) generalizes better than a policy trained on only *put tiger in drawer* data (Target Data Only).

Figure 7: Objects used in zero-shot generalization experiments.

A.2.3 Further Generalization Results

In Table 7 we provide further zero-shot generalization results for the **Tray Drawer** scene. As shown in the main paper, a policy trained on multi-task prior data and fine-tuned on a target task (ARIEL) generalizes to unseen objects better than a policy trained on only target task data (Target Data Only). We evaluate the zero-shot generalization capabilities of the ARIEL policies at different stages during the process of fine-tuning on the target task (*put tiger in drawer*). As in the main paper, ARIEL policies at earlier stages of fine-tuning (360 vs 600 trials) demonstrate better zero-shot generalization capabilities to unseen objects than at later stages. Even if policy is initialized by training on diverse multi-task prior data, as more updates are made on the target task, the policy becomes more specialized and less capable of performing the unseen tasks.

A.2.4 Generalization Test Objects

The objects we use to test the zero-shot generalization capabilities of ARIEL are depicted in Figure 7. Note that although the objects chosen to test generalization in the **Tray Drawer** scene are seen in the prior data for the **Tray Container** scene, they are not contained in the prior data for the **Tray Drawer** scene.

A.3 Simulated Experiments

In this section, we provide additional details on our simulation experiments, the results for which were presented in Section 5.2. We utilize a Pybullet-based simulation [40] with a simulated version of the WidowX robot we use in the real world. The tasks also involve picking and placing objects and putting them into a container. We use 3D object models from the Shapenet dataset [41] to test our method on diverse objects. We utilize a near-convex decomposition of the models in order to maintain good contact physics. Figure 8 shows the simulated experiment setup.

We compare to two prior approaches in our simulated experiments. To evaluate how well our method enables learning with minimal resets, we compare to **R3L** [6], which alternates between training a forward controller to optimize a task-completion reward and training a perturbation controller that optimizes a novelty exploration bonus. We initialize the forward controller in R3L with the policy obtained by running offline multi-task RL on the prior data. We also compare to a method

Figure 8: (Left) The simulated experiment setup. (Right) Simulation training (upper) and test (lower) objects. Our offline prior dataset consists of only training objects.

Attribute	Value
Timesteps per Offline Trajectory	30
Timesteps per Online Trajectory	40
Forward Tasks	8
Backward Tasks	8
Number of Trajectories Per Task	512
Average Success Rate	0.38

Table 8: Simulated tasks prior data details.

that does *not* optimize the task embeddings $\mathbf{z}_f$ and $\mathbf{z}_b$ over the course of autonomous online fine-tuning. We refer to this approach as **Multi-task RL**. This is equivalent to first running multi-task offline RL on the prior dataset, and then fine-tuning the policy using a fixed task index (that was unused during pre-training) in an online phase afterwards. This method is conceptually similar to MT-Opt [2], but adapted to our pre-training and then fine-tuning setup, as opposed to the re-training from scratch approach followed by Kalashnikov et al. [2]. For instructive purposes, we also compare to an "oracle" version of our approach, labeled (**ARIEL + resets**), which assumes access to external resets at the end of each episode. While resetting every episode is prohibitively expensive in the real-world, we can still run this method in simulation for our understanding.

The results in Figure 3 show that ARIEL and its oracle reset variant are the only methods that succeed at learning the new tasks, and the full ARIEL method closely matches final oracle performance. While the poor performance of prior methods might be surprising, recall that these tasks require using raw image observations and sparse 0/1 rewards, which present a significant challenge for any RL approach. While **R3L** and **Multi-Task RL** both succeeded at learning the tasks in the offline phase (see Figure 9), they are unable to make progress during online training of the new task. The comparison to **Multi-task RL** indicates the importance of adapting the task embeddings: if the task embeddings are not adapted to the new task, distributional shift in the task embeddings may severely hamper effective reset-free learning.

A.3.1 Simulation Dataset Details

Just as in our real world experiments, we perform scripted collection in a reset-free manner, with the robot attempting to move the objects both into and out of the container. In Figure 8, we show the set of training and testing objects used in the various pick and place tasks. In Table 8 we provide details on number of tasks, number of trajectories and other dataset properties.

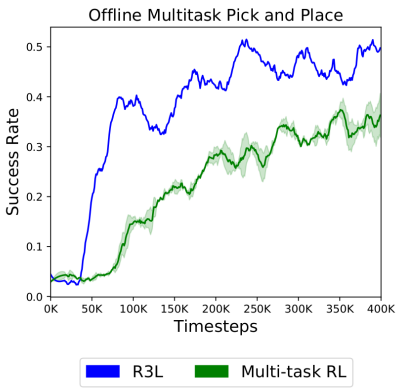

Figure 9: We see that the **R3L** and **Multi-task RL** baselines show a significant amount of learning in the offline stage, even though they perform poorly during fine-tuning as seen in Figure 3.