A Implementation Details

A.1 Behavior Primitives

We elaborate on the manipulation primitives that we outlined in Section 3.2: reaching, grasping, pushing, gripper release, and atomic. We classify all primitives that are not the atomic primitive as *non-atomic* primitives. Under the hood, all non-atomic primitives execute a variable sequence of atomic actions, either until the primitive is successfully executed or until a time limit is reached. All atomic actions specifically interface with the Operational Space Control (OSC) controller, which has 5 degrees of freedom: 3 degrees to control the position of the end effector, 1 degree to control the yaw angle, and (for all tasks but wiping) 1 degree to open and close the gripper.

We elaborate further on our non-atomic primitives. The gripper release primitive executes a fixed number of atomic actions to open the gripper. The reaching, grasping, and pushing primitives are hard-coded closed-loop controllers that all entail a reaching phase, either for reaching the starting location (for pushing) or for reaching the final location (for reaching and grasping). To implement this functionality for table-top environments (all except door), the robot first lifts its end effector to a pre-specified height, then hovers to the target XY position, and finally lowers its end effector to the target location. For other environments (door), the robot moves to toward the target location directory via the OSC controller. During this reaching phase, the reaching primitive keeps its gripper closed (except for the non-tabletop environments like door) and the grasping and pushing primitives keep their grippers open. The grasping and pushing primitives can be configured to achieve a specified yaw angle, which they satisfy during the reaching phase simultaneously while applying end effector displacements. Upon reaching, the grasping primitive emulates grasping by closing its gripper and the pushing the primitive emulates pushing by applying end effector displacements in a specified direction.

A.2 Algorithm

Our algorithm implementation is based on Soft Actor-Critic. We represent the Q network and task policy as single multilayer perceptions (MLPs), while we represent the parameter policy as a collection of MLP sub-networks, with one sub-network dedicated for each primitive. This enables us to accommodate primitives with heterogeneous parameterizations. To allow batch tensor computations across primitives with different parameter dimensions, these parameter policy sub-networks all output a "one size fits all" distribution over parameters $x \in \mathbb{R}^{d_A}$, where $d_A = \max_a d_a$ is the maximum parameter dimension over all primitives. At primitive execution we simply truncate the parameters x to the length d_a of the chosen primitive a.

Our algorithm alternates between collecting on-policy transitions in the environment and performing off-policy training on data sampled from the replay buffer. Training specifically entails optimizing the Q network, task policy, and parameter policy via gradient descent. As in the original SAC implementation, we use the reparameterization trick with respect to the parameter policy loss in order to reduce the variance of our gradient estimates. While we assume continuous primitive parameters we can also represent discrete parameters and apply reparameterization with the Gumbel-Softmax trick [26, 38]. We provide a full outline of our algorihtm in Algorithm 1.

A.3 Affordance Score

We elaborate on the affordance score introduced in Section 3.4:

$$s_{\text{aff}}(s, x; a) = \max_{p \in P} 1 - \tanh\Big(\max(\|x_{reach} - p\| - \tau, 0)\Big)$$

The keypoint p is dependent on the primitive a and the current state s. For example for the cleanup task, the keypoint for the pushing primitive is the location of a pushable object (the jello box), the keypoint for a grasping primitive is the location of a graspable object (the spam can), and the keypoint for the reaching primitive is the location of the bin. If there are multiple keypoints of interest we calculate the affordance score corresponding to each keypoint and consider the maximum score. If no applicable keypoint exists for a primitive (e.g. there are no pushable objects in door opening) we give an affordance score of 0. By default we set the threshold τ to 0.03 for grasping, 0.06 for reaching, and 0.12 for pushing. There are a few exceptions for tasks that need larger affordance regions for reaching large objects.

Algorithm 1 Manipulation **P**rimitive-augmented Reinforcement **Le**arning (MAPLE)

Initialize Q network $Q_\theta(s, a, x)$, task policy $\pi_{tsk_\phi}(a|s)$, parameter policy $\pi_{p_\psi}(x|s, a)$, replay buffer $\mathcal{D}$
for iteration $1, \ldots, N$ **do**
  for episode $1, \ldots, M$ **do** {Exploration Phase}
    Initialize timer $t \leftarrow 0$
    Initialize episode s_0
    while episode not terminated **do**
      Sample primitive type a_t from task policy $\pi_{tsk_\phi}(a_t|s_t)$
      Sample primitive parameters x_t from parameter policy $\pi_{p_\psi}(x_t|s_t, a_t)$
      Truncate sampled parameters to dimension of sampled primitive $x_t \leftarrow x_t[: d_{a_t}]$
      Execute a_t and x_t in environment, obtain reward r_t and next state s_{t+1}
      Add affordance score to reward $r_t \leftarrow r_t + \lambda s_{\text{aff}}(s_t, x_t; a_t)$
      Add transition to replay buffer $\mathcal{D} \leftarrow \mathcal{D} \cup \{s_t, a_t, x_t, r_t, s_{t+1}\}$
      Update timer $t \leftarrow t + 1$
    end while
  end for
  for training step $1, \ldots, K$ **do** {Training Phase}
    Update Q network: $\theta \leftarrow \theta - \lambda_{lr} \nabla_\theta J_Q(\theta)$
    Update task policy: $\phi \leftarrow \phi - \lambda_{lr} \nabla_\phi J_{\pi_{tsk}}(\phi)$
    Update parameter policy: $\psi \leftarrow \psi - \lambda_{lr} \nabla_\psi J_{\pi_p}(\psi)$
  end for
end for

A.4 Flat Baseline

We considered two variants for our flat baseline. One variant, which has been explored by prior work [68, 21], outputs a distribution over the primitive type a and the parameters $\{x^1, x^2, \cdots, x^k\}$ for all primitives. As we discussed in Section 3.3 under this approach the number of policy outputs scales linearly with the total number of primitive parameters, which can lead to optimization difficulties for large behavior libraries. Empirically we found this to be the case, as we were unable to make any progress on any of our tasks despite extensive hyperparamter tuning. Neunert et al. [50] proposed an alternative approach of replacing the distribution over all parameter outputs with the "one size fits all" distribution that we described in Appendix A.2. Parameter selection occurs by *independently* sampling a primitive type a and parameters x, and subsequently truncating the sampled parameters by the dimension of the sampled primitive type. This sampling strategy was also adopted by Lee et al. [36]. We note that this independent sampling process is in contrast to our two-stage hierarchical process. We adopted this variant as our flat baseline, and in Figure 4 we see that it often leads to sub-optimal performance. We hypothesize that this is due the fact that the parameter selection process is not informed by the primitive type selection process, which reduces the agent's utility especially when dealing with primitives that have heterogeneous parameter structures.

A.5 Open Loop Baseline

Our open loop baseline follows an open loop task schema, and is inspired from Chitnis et al. [8]. The open loop baseline and our method share identical implementations, except for the input to the task policy: our method takes in the current environment observation while the open loop baseline takes in only the current episode timestep. We highlight that while our implementation is inspired from Chitnis et al. [8], there are notable differences. Their update rule for the "task policy" (or equivalent thereof) does not use gradient descent, relies on on-policy sampling, and is designed for the sparse reward setting only. We found these assumptions to be restrictive for our algorithmic and task setup, and we instead use gradient-based, off-policy reinforcement learning methods which can work with sparse or dense rewards. Despite these differences, we believe that our open loop baseline captures the essence of the ideas proposed in Chitnis et al. [8] – namely that open loop task shemas can enable more efficient and effective learning. As we show in Figure 4 however, we did not find this to be the case for the relatively more complex tasks in our suite of manipulation domains.

A.6 DAC Baseline

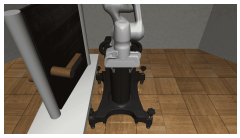
We considered a number of potential methods as our representative option-learning baseline. While prominent prior work [4, 32, 73] has focused on learning options, we verified that Double Actor-Critic (DAC) achieves superior performance on the OpenAI HalfCheetah-v2 task and our lift task, so we chose DAC as our representative options baseline. We also considered Deep Skill Chaining [5], another recent work that learns options; however it was not applicable to our manipulation domains given that it is designed primarily for goal-based navigation agents. We used the implementation publicly released by the DAC authors [2], and we adopted the hyperparameters that they suggested in their paper.

B Experimental Setup

B.1 Environments

We conduct experiments on eight manipulation tasks of varying complexities, spanning diverse prehensile and non-prehensile behaviors. The first six come from the standard robosuite benchmark [74]. We designed the last two (cleanup, peg insertion) to evaluate our method in multi-stage, contact-rich tasks. We elaborate on each as follows:

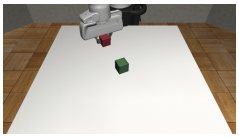
Lift: the robot must pick up a cube and lift it above the table.



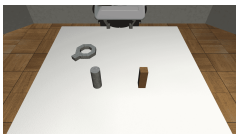
Door Opening: the robot must turn the door handle and open the door.

Pick and Place: the robot must pick up a soda can and place it into a specific target compartment.

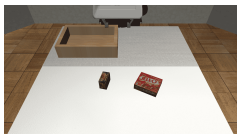
Wipe: the robot must wipe a table containing spilled debris. A penalty is given if the robot presses too hard into the table.



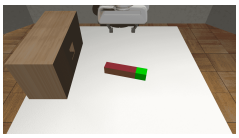
Stack: the robot must stack a cube on top of another cube.



Nut Assembly: the robot must fit a nut tool onto the round peg.



Cleanup: the robot must store a spam can into a storage bin and store a jello box at the upper right corner.



Peg Insertion: the robot must pick up the peg and insert it into the opening of a wooden block.

B.2 Training

We provide a full list of our algorithm hyperparameters in Table 1. We note a few additional details. For a consistent comparison across baselines, our episode lengths are fixed to 150 *atomic* timesteps, meaning that we execute a variable number of primitives until we have exceeded the maximum number of atomic actions for the episode. Also, for the first 600k environment steps we set the target entropy for the task policy and parameter policy to a high value to encourage higher exploration during the initial stages of training.

B.3 Evaluation

We elaborate on the evaluation protocol for our experiments in Figure 4. We evaluate each experimental variant (combination of task and method) over 5 seeds and we (1) plot the agent's rewards throughout training and (2) report the task success rate at the end of training. Specifically for the reward plots, we evaluate the agent's average episodic task rewards (excluding the affordance reward)

[2] `https://github.com/ShangtongZhang/DeepRL/tree/DAC`

Table 1: Hyperparameters for our algorithm

Hyperparameter	Value
Hidden sizes (all networks)	$256, 256$
Q network and policy activation	ReLU
Q network output activation	None
Policy network output activation	tanh
Optimizer	Adam
Batch Size	1024
Learning rate (all networks)	$3\text{e}{-5}$
Target network update rate τ	$1\text{e}{-3}$
# Training steps per epoch	1000
# (Low-level) exploration actions per epoch	3000
Replay buffer size	1e6
Episode length (# low-level actions)	150 (except wipe, 300)
Discount factor	0.99
Reward scale	5.0
Affordance score scale λ	3.0
Automatic entropy tuning	True
Target Task Policy Entropy	$0.50 \times log(k)$, k is number of primitives
Target Parameter Policy Entropy	$-\max_a d_a$

at regularly spaced training checkpoints every $30k$ environment exploration steps. The episodic rewards are averaged over 20 episodes and are normalized between 0 and 100, where 100 corresponds to the agent receiving the maximum possible reward at every single timestep of the episode. We post-process the plots, showing the moving average of results over the last $150k$ environment steps. For reporting the final task success rate, we load the final training checkpoint and report the average task success rate over 20 episodes. Success rates for our tasks are defined as follows:

- Lift: whether the block is above a height threshold
- Door: whether the door angle is past a threshold
- Pick and Place: whether the can is in the correct target bin and the robot is not holding the can
- Wipe: whether all of the debris is wiped off the table
- Stack: whether the smaller cube is on top of the larger cube and the robot is not holding either cube
- Nut Assembly: whether the nut is fitted completely onto the round peg and the robot is not holding the nut
- Cleanup: whether the spam can is in the bin and the jello box is within a threshold distance away from the table corner
- Peg Insertion: whether the peg is inserted into wooden block past a threshold distance

Final task success rates for all baselines across all tasks are outlined in Table 2.

We also elaborate on the evaluation protocol for the compositionality scores that we report in Figure 5. Our compositionality scores are averaged over 5 seeds for each task. For each seed we sample 50 task sketches from the last training checkpoint and we discard sketches that did not correspond to the agent solving the task. Of these remaining task sketches we calculate the compositionality score according to Equation (5).

B.4 Task Sketch Transfer Experiments

For our task sketch experiments we first extract a set of task sketches $\{K_1, \cdots, K_n\}$ from the source task. We subsequently select the sketch K that has the lowest Levenshtein distance with all other

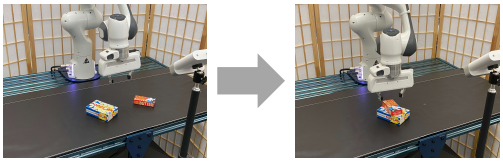
Table 2: Final Task Success Rates (%)

	Lift	**Door**	**Pick and Place**	**Wipe**	**Stack**	**Nut Assembly**	**Cleanup**	**Peg Insertion**
Atomic [20]	**98.0 ± 2.4**	0.0 ± 0.0	0.0 ± 0.0	18.0 ± 18.3	38.0 ± 28.7	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Flat [36, 50]	61.0 ± 47.8	**100.0 ± 0.0**	1.0 ± 2.0	22.0 ± 9.8	**98.0 ± 2.4**	0.0 ± 0.0	0.0 ± 0.0	8.0 ± 13.6
Open Loop [8]	43.0 ± 43.5	**100.0 ± 0.0**	81.0 ± 38.0	16.0 ± 4.9	85.0 ± 3.2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
DAC [73]	75.0 ± 12.7	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	16.0 ± 32.0
MAPLE (Non-At.)	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**	**42.0 ± 9.8**	**99.0 ± 2.0**	93.0 ± 14.0	**100.0 ± 0.0**	0.0 ± 0.0
MAPLE (ours)	**100.0 ± 0.0**	**100.0 ± 0.0**	**95.0 ± 7.7**	**42.0 ± 11.7**	**98.0 ± 2.4**	**99.0 ± 2.0**	91.0 ± 5.8	**100.0 ± 0.0**

task sketches. In the case of our pick and place task this was {Grasp, Reach, Release}. Once we have extracted the sketch K from the source task, we train on the target task with a fixed task sketch of K. For each episode we iterate through the sequence of primitives in K, repeating each primitive up to 5 times until the agent receives high affordance reward, before moving onto the next primitive in the sketch. Upon executing all of the primitives in the task sketch the agent executes 10 atomic primitives to satisfy any behaviors that it was not able to fulfill with the sketch alone, and then the episode terminates.

B.5 Real-World Experiments

We performed evaluations on two real-world manipulation tasks:



Stack: the robot must pick up the butter box and stack it on top of the popcorn box.

Cleanup: the robot must (1) pick up the butter box and place it into the bin and (2) push the popcorn to the right side of the table (the white area).

Both tasks resemble the simulated stack and cleanup tasks outlined in Appendix B.1, but have differences in the size of the objects, table size, and workspace layout. To account for these differences we designed variations of our existing simulated stack and cleanup tasks to match the characteristics of our real-world tasks. We trained policies in simulation (until convergence) and transferred them to the real-world for evaluation.

For perception we use the robot proprioception data from the robot's on-board sensors, in conjunction with pose estimates of the objects using the deep object pose estimation system [65] paired with a single Microsoft Kinect camera. Under this setup the objects are sometimes out of the camera view or are occluded by the robot arm; in these cases the pose estimator does not return estimates of the object. We mostly alleviate such conditions through a three step procedure: (1) the robot lifts its end-effector to a pre-determined location in the air where occlusions are minimized, (2) the pose estimator re-computes the poses of the objects, (3) the robot moves back to its initial location. At the end of step (3), the pose of objects are assumed to be the pose estimates from step (2), with the exception of objects that were moved by the robot during step (1). Such objects comprise objects that the robot was already holding before step (1) and that the robot subsequently lifted into the air during step (1). For such objects, we compute the pose of the object as the final robot pose in addition to the relative pose difference of the robot end effector and object during step (2). In addition to handling occlusions, we noticed that the pose estimation system routinely made small errors when estimating the position and orientation of the objects. To minimize the influence of such errors, we hard-coded the pitch and yaw angles of the objects (as they were always flat either on the table or in the air) and the height of the object whenever it was detected to lie on the table. These constraints were necessary to ensure reliable perception estimates, but we anticipate that with improved perception systems in the future such constraints can be relaxed.

For evaluation, we performed 30 trials for each task, where the robot was allowed a maximum of 20 primitive calls per episode. We recorded an average success rate of 93% for stack (in contrast to 97% in simulation), and 83% for cleanup (in contrast to 93% in simulation). Most failures were either due to the robot repeatedly applying poor grasping actions or the robot hitting its joint limits, subsequently triggering a safety call to halt the robot.