# Accelerating Robotic Reinforcement Learning via Parameterized Action Primitives

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

Despite the potential of reinforcement learning (RL) for building general-purpose robotic systems, training RL agents to solve robotics tasks still remains challenging due to the difficulty of exploration in purely continuous action spaces. Addressing this problem is an active area of research with the majority of focus on improving RL methods via better optimization or more efficient exploration. An alternate but important component to consider improving is the interface of the RL algorithm with the robot. In this work, we manually specify a library of robot action primitives (RAPS), parameterized with arguments that are learned by an RL policy. These parameterized primitives are expressive, simple to implement, enable efficient exploration and can be transferred across robots, tasks and environments. We perform a thorough empirical study across challenging tasks in three distinct domains with image input and a sparse terminal reward. We find that our simple change to the action interface substantially improves both the learning efficiency and task performance irrespective of the underlying RL algorithm, significantly outperforming prior methods which learn skills from offline expert data.

## 1 Introduction

Meaningful exploration remains a challenge for robotic reinforcement learning systems. For example, in the manipulation tasks shown in Figure 1, useful exploration might correspond to picking up and placing objects in different configurations. However, random motions in the robot's joint space will rarely, if ever, result in the robot touching the objects, let alone pick them up. Recent work, on the other hand, has demonstrated remarkable success in training RL agents to solve manipulation tasks [4, 24, 26] by sidestepping the exploration problem with careful engineering. Levine et al. [26] use densely shaped rewards estimated with AR tags, while Kalashnikov et al. [24] leverage a large scale robot infrastructure and Andrychowicz et al. [4] require training in simulation with engineered reward functions in order to transfer to the real world. In general, RL methods can be prohibitively data inefficient, require careful reward development to learn, and struggle to scale to more complex tasks without the aid of human demonstrations or carefully designed simulation setups.

An alternative view on why RL is difficult for robotics is that it requires the agent to learn both *what* to do in order to achieve the task and *how* to control the robot to execute the desired motions. For example, in the kitchen environment featured at the bottom of Figure 1, the agent would have to learn how to accurately manipulate the arm to reach different locations as well as how to grasp different objects, while also ascertaining what object it has to grasp and where to move it. Considered independently, the problems of controlling a robot arm to execute particular motions and figuring out the desired task from scalar reward feedback, then achieving it, are non-trivial. Jointly learning to solve both problems makes the task significantly more difficult.

In contrast to training RL agents on raw actions such as torques or delta positions, a common strategy is to decompose the agent action space into higher (i.e., *what*) and lower (i.e., *how*) level structures.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

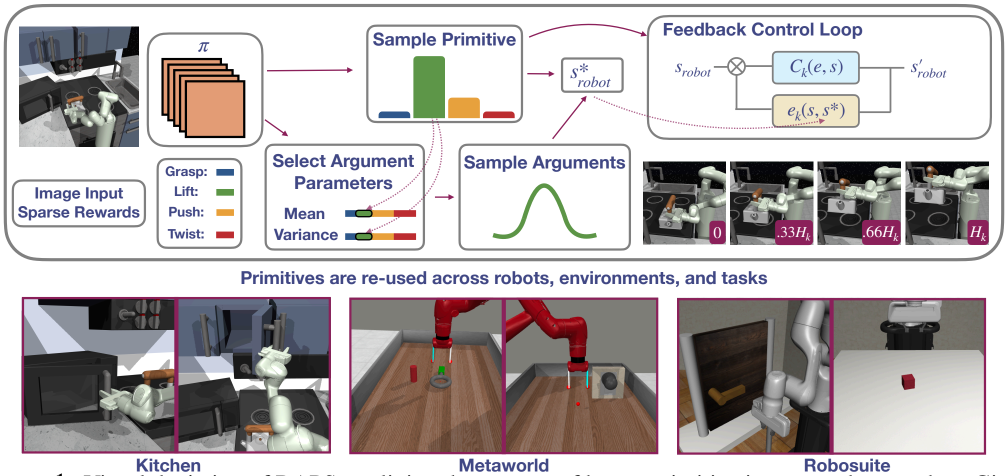


Figure 1: Visual depiction of RAPS, outlining the process of how a primitive is executed on a robot. Given an input image, the policy outputs a distribution over primitives and a distribution over all the arguments of all primitives, samples a primitive and selects its corresponding argument distribution parameters, indexed by which primitive was chosen, samples an argument from that distribution and executes a controller in a feedback loop on the robot for a fixed number of timesteps ($H_k$) to reach a new state.

A number of existing methods have focused on designing or learning this structure, from manually architecting and fine-tuning action hierarchies [14, 27, 32, 46], to organizing agent trajectories into distinct skills [3, 20, 40, 49] to more recent work on leveraging large offline datasets in order to learn skill libraries [29, 39]. While these methods have shown success in certain settings, many of them are either too sample inefficient, do not scale well to more complex domains, or lack generality due to dependence on task relevant data.

In this work, we investigate the following question: instead of learning low-level primitives, what if we were to design primitives with minimal human effort, enable their expressiveness by parameterizing them with arguments and learn to control them with a high-level policy? Such primitives have been studied extensively in task and motion planning (TAMP) literature [22] and implemented as parameterized actions [19] in RL. We apply primitive robot motions to redefine the policy-robot interface in the context of robotic reinforcement learning. These primitives include manually defined behaviors such as `lift`, `push`, `top-grasp`, and many others. The behavior of these primitives is parameterized by arguments that are the learned outputs of a policy network. For instance, `top-grasp` is parameterized by four scalar values: grasp position (x,y), how much to move down (z) and the degree to which the gripper should close. We call this application of parameterized behaviors, Robot Action Primitives for RL (RAPS). A crucial point to note is that these parameterized actions are *easy* to design, need only be defined *once* and can be *re-used* without modification across tasks.

The main contribution of this work is to support the effectiveness of RAPS via a thorough empirical evaluation across several dimensions:

- How do parameterized primitives compare to other forms of action parameterization?
- How does RAPS compare to prior methods that learn skills from offline expert data?
- Is RAPS agnostic to the underlying RL algorithm?
- Can we stitch the primitives to perform multiple complex manipulation tasks in sequence?
- Does RAPS accelerate exploration even in the absence of extrinsic rewards?

We investigate these questions across complex manipulation environments including Kitchen Suite, Metaworld and Robosuite domains. We find that a simple parameterized action based approach outperforms prior state-of-the-art by a significant margin across most of these settings[1].

## 2 Related Work

**Higher Level Action and Policy Spaces in Robotics** In robotics literature, decision making over primitive actions that execute well-defined behaviors has been explored in the context of

[1]We will be releasing the code to reproduce our results.

task and motion planning [9, 22, 23, 42]. However, such methods are dependent on accurate state estimation pipelines to enable planning over the argument space of primitives. One advantage of using reinforcement learning methods instead is that a neural network policy can learn to adjust its implicit state estimates through trial and error experience. Dynamic Motion Primitive and ensuing policy search approaches [11, 21, 25, 35, 36] leverage dynamical systems to learn flexible, parameterized skills, but are sensitive to hyper-parameter tuning and often limited to the behavior cloning regime. Neural Dynamic Policies [6] incorporate dynamical structure into neural network policies for RL, but evaluate in the state based regime with dense rewards, while we show that simple, parameterized actions can enable RL agents to efficiently explore in sparse reward settings from image input.

**Hierarchical RL and Skill Learning** Enabling RL agents to act effectively over temporally extended horizons is a longstanding research goal in the field of hierarchical RL. Prior work introduced the options framework [44], which outlines how to leverage lower level policies as actions for a higher level policy. In this framework, parameterized action primitives can be viewed as a particular type of fixed option with an initiation set that corresponds to the arguments of the primitive. Prior work on options has focused on discovering [1, 12, 40] or fine-tuning options [5, 14, 27] in addition to learning higher level policies. Many of these methods have not been extended beyond carefully engineered state based settings. More recently, research has focused on extracting useful skills from large offline datasets of interaction data ranging from unstructured interaction data [48], play [28, 29] to demonstration data [2, 34, 38, 39, 43, 45, 52]. While these methods have been shown to be successful on certain tasks, the learned skills are only relevant for the environment they are trained on. New demonstration data must be collected to use learned skills for a new robot, a new task, or even a new camera viewpoint. RAPS does not have the aforementioned limitations as our primitives are manually specified. They can re-use the same implementation details across robots, provided a low-level controller implementation, are defined independent of any task and are only a function of the robot state, not the world state or the observations.

**Parameterized Actions in RL** The parameterized action Markov decision process (PAMDP) formalism was first introduced in Masson et al. [31], though there is a large body of earlier work in the area of hybrid discrete-continuous control, surveyed in [7, 8]. Most recent research on PAMDPs has focused on better aligning policy architectures and RL updates with the nature of parameterized actions and has largely been limited to state based domains [13, 50]. A number of papers in this area have focused on solving a simulated robot soccer domain modeled as either a single-agent [19, 31, 47] or multi-agent [15] problem. In this paper, we consider more realistic robotics tasks that involve the interaction with and manipulation of common household objects. Work on hybrid discrete-continuous control in the context of RL [33] has largely been limited to state based control with dense rewards, while we show that parameterized actions can enable an RL agent to learn challenging manipulation tasks from visual input without dense reward feedback. While prior work [41] has trained RL policies to select hand-designed behaviors for simultaneous execution, we instead train RL policies to leverage more expressive, parameterized behaviors to solve a wide variety of tasks. Most closely related to this work is Chitnis et al. [10], which develops a specific architecture for training policies over parameterized actions from *state* input and sparse rewards in the context of bi-manual robotic manipulation. Our work is orthogonal in that we demonstrate that a simple parameterization of the higher level policy is sufficient to solve a large suite of manipulation tasks from image input, but in principle policy architectures from prior work could be used as well.

# 3 Robot Action Primitives in RL

To address the challenge of exploration and behavior learning in continuous action spaces, we decompose a desired task into the *what* (high level task) and the *how* (control motion). The *what* is handled by the *environment-centric* RL policy while the *how* is handled by a fixed, manually defined set of *agent-centric* primitives parameterized by continuous arguments. This enables the high level policy to reason about the task at a high level by choosing primitives and their arguments while leaving the low-level control to the parameterized actions themselves.

## 3.1 Background

Let the Markov decision process (MDP) be defined as $(\mathcal{S}, \mathcal{A}, \mathcal{R}(s, a, s'), \mathcal{T}(s'|s, a), p(s_0), \gamma, )$ in which $\mathcal{S}$ is the set of true states, $\mathcal{A}$ is the set of possible actions, $\mathcal{R}(s, a, s')$ is the reward function, $\mathcal{T}(s'|s, a)$ is the transition probability distribution, $p(s_0)$ defines the initial state distribution, and $\gamma$ is the discount factor. The agent executes actions in the environment using a policy $\pi(a|s)$ with a corresponding trajectory distribution $p(\tau = (s_0, a_0, ...a_{t-1}, s_T)) = p(s_0)\Pi_t \pi(a_t|s_t)\mathcal{T}(s_{t+1}|s_t, a_t)$. The goal of the RL agent is to maximize the expected sum of rewards with respect to the policy: $\mathbb{E}_{s_0,a_0,...a_{t-1},s_T,\sim p(\tau)}\left[\sum_t \gamma^t \mathcal{R}(s_t, a_t)\right]$. In the case of vision-based RL, the setup is now a partially observed Markov decision process (POMDP); we have access to the true state via image observations. In this case, we include an observation space $\mathcal{O}$ which corresponds to the set of visual observations that the environment may emit, an observation model $p(o|s)$ which defines the probability of emission and policy $\pi(a|o)$ which operates over observations. In this work, we consider various modifications to the action space $\mathcal{A}$ while keeping all other components of the MDP or POMDP the same.

### 3.2 Parameterized Action Primitives

We now describe the specific nature of our parameterized primitives as well as how they can be integrated into existing RL algorithms (see Figure 1 for an end-to-end visualization of the method). In a library of K primitives, the k-th primitive is a function $f_k(s, \text{args})$ that executes a controller $C_k$ on a robot for a fixed horizon $H_k$, $s$ is the robot state and $\text{args}$ is the value of the arguments passed to $f_k$. $\text{args}$ is used to compute a target robot state $s^*$ and then $C_k$ is used to drive $s$ to $s^*$. A primitive dependent error metric $e_k(s, s^*)$ determines the trajectory $C_k$ takes to reach $s^*$. $C_k$ is a general purpose state reaching controller, e.g. an end-effector or joint position controller; we assume access to such a controller for each robot and it is straightforward to define and tune if not provided. Given a low-level controller implementation for the robot, the same exact primitive implementation can be re-used across any robot. In this setup, the choice of controller, error metric and method to compute $s^*$ define the behavior of the primitive motion, how it uniquely forms a movement in space. We refer to Procedure 1 for a general outline of a parameterized primitive.

As an example, consider the "lifting" primitive, which simply involves lifting the robot arm upward. For this action, $\text{args}$ is the amount to lift the robot arm, e.g. by 20cm., the robot state for this primitive is the robot end-effector position, k is the index of the lifting primitive in the library, $C_k$ is an end-effector controller, $e_k(s, s^*) = s^* - s$, and $H_k$ is the end-effector controller horizon, which in our setting ranges from 100-300. The target position $s^*$ is computed as $s + [0, 0, \text{args}]$. $f$ moves the robot arm for $H_k$ steps, driving $s$ towards $s^*$. The other primitives are defined in a similar manner; see the appendix for a precise description of each primitive we define.

**Procedure 1** Parameterized Action Primitive

**Input:** primitive dependent argument vector $\text{args}$, primitive index $k$, robot state $s$

1: compute $s^*(\text{args}, s)$
2: **for** $i = 1, ..., H_k$ low-level steps **do**
3: $\quad e_i = e_k(s_i, s^*)$ ▷ compute state error
4: $\quad a_i = C_k(e_i, s_i)$ ▷ compute torques
5: $\quad$ execute $a_i$ on robot
6: **end for**

Robot action primitives are only a function of the robot state, not the world state. The primitives function only by reaching set points of the robot state as directed by the policy, hence they are *agent-centric*. This design makes primitives agnostic to camera view, visual distractors and even the underlying environment itself. The RL policy, on the other hand, is *environment centric*: it chooses the primitive and appropriate arguments based on environment observations in order to best achieve the task. A key advantage of this decomposition is that the policy no longer has to learn *how* to move the robot and can focus directly on *what* it needs to do. Meanwhile, the low-level control need not be perfect because the policy can account for most discrepancies using the arguments. We note that one issue with using a fixed library of primitives is that it cannot define all possible robot motions. As a result, we include a dummy primitive that corresponds to the raw action space, specifically end-effector position control. This does not provide a complete solution to the problem as the dummy primitive operates on the high level horizon for $H_k$ steps when called. Therefore, it cannot execute every trajectory that a lower level policy could, yet we find the primitive library as a whole performs well in practice.

In order to integrate these parameterized actions into the RL setting, we modify the action space of a standard RL environment to involve two operations at each time step: (a) choose a primitive out of a fixed library (b) output its arguments. As in Chitnis et al. [10], the policy network outputs a distribution over one-hot vectors defining which primitive to use as well as a distribution over all of the arguments for all of the primitives, a design choice which enables the policy network to have a fixed output dimension. After the policy samples an action, the chosen parameterized action and its corresponding arguments are indexed from the action and passed to the environment. The environment then selects the appropriate primitive function $f$ and executes the primitive on the robot with the appropriate arguments. After the primitive completes executing, the final observation and sum of intermediate rewards during the execution of the primitive are returned by the environment.

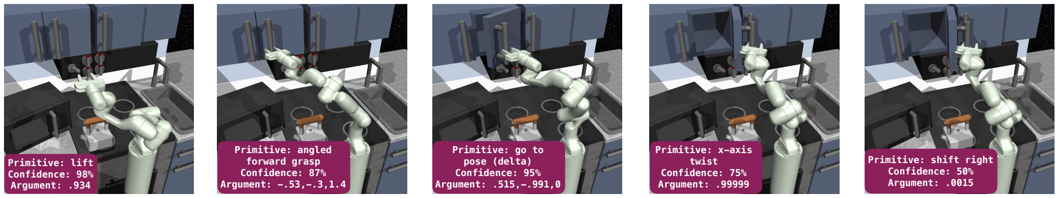

Figure 2: We visualize an execution of an RL agent trained to solve a cabinet opening task from sparse rewards using robot action primitives. At each time-step, we display the primitive chosen, the policy's confidence in the action choice and the corresponding argument passed to the primitive in the bottom left corner.

We do so in order to ensure that if the task is achieved mid primitive execution, the action is still labelled successful. Using this policy architecture and primitive execution format, we train standard RL agents to solve manipulation tasks from sparse rewards. See Figure 2 for a visualization of a full trajectory of a policy solving a hinge cabinet opening task in the Kitchen Suite with RAPS.

## 4 Experimental Setup

In order to perform a robust evaluation of robot action primitives and prior work, we select a set of challenging robotic control tasks, define our environmental setup, propose appropriate metrics for evaluating different action spaces, and summarize our baselines for comparison.

**Tasks and Environments**: We evaluate RAPS on three simulated domains: Metaworld [17], Kitchen [51] and Robosuite [53], containing 16 tasks with varying levels of difficulty, realism and task diversity (see the bottom half of Fig. 1). We use the Kitchen environment because it contains seven different subtasks within a single setting, contains human demonstration data useful for training learned skills and contains tasks that require chaining together up to four subtasks to solve. In particular, learning such temporally-extended behavior is challenging [2, 17, 34]. Next, we evaluate on the Metaworld benchmark suite due to its wide range of manipulation tasks and established presence in the RL community. We select a subset of tasks from Metaworld (see appendix) with different solution behaviors to robustly evaluate the impact of primitives on RL. Finally, one limitation of the two previous domains is that the underlying end-effector control is implemented via a simulation constraint as opposed to true position control by applying torques to the robot. In order to evaluate if primitives would scale to more realistic learning setups, we test on Robosuite, a benchmark of robotic manipulation tasks which emphasizes realistic simulation and control. We select the block lifting and door opening environments which have been demonstrated to be solvable in prior work [53]. We refer the reader to the appendix for a detailed description of each environment.

**Sparse Reward and Image Observations** We modify each task to use the environment success metric as a sparse reward which returns 1 when the task is achieved, and 0 otherwise. We do so in order to establish a more realistic and difficult exploration setting than dense rewards which require significant engineering effort and true state information to compute. Additionally, we plot all results against the mean task success rate since it is a directly interpretable measure of the agent's performance. We run each method using visual input as we wish to bring our evaluation setting closer to real world setups. The higher level policy, primitives and baseline methods are not provided access to the world state, only camera observations and robot state depending on the action.

**Evaluation Metrics** One challenge when evaluating hierarchical action spaces such as RAPS alongside a variety of different learned skills and action parameterizations, is that of defining a fair and meaningful definition of sample efficiency. We could define one sample to be a forward pass through the RL policy. For low-level actions this is exactly the sample efficiency, for higher level actions this only measures how often the policy network makes decisions, which favors actions with a large number of low-level actions without regard for controller run-time cost, which can be significant. Alternatively, we could define one sample to be a single low-level action output by a low-level controller. This metric would accurately determine how often the robot itself acts in the world, but it can make high level actions appear deceptively inefficient. Higher level actions execute far fewer forward passes of the policy in each episode which can result in faster execution on a robot when operating over visual observations, a key point low-level sample efficiency fails to account for.

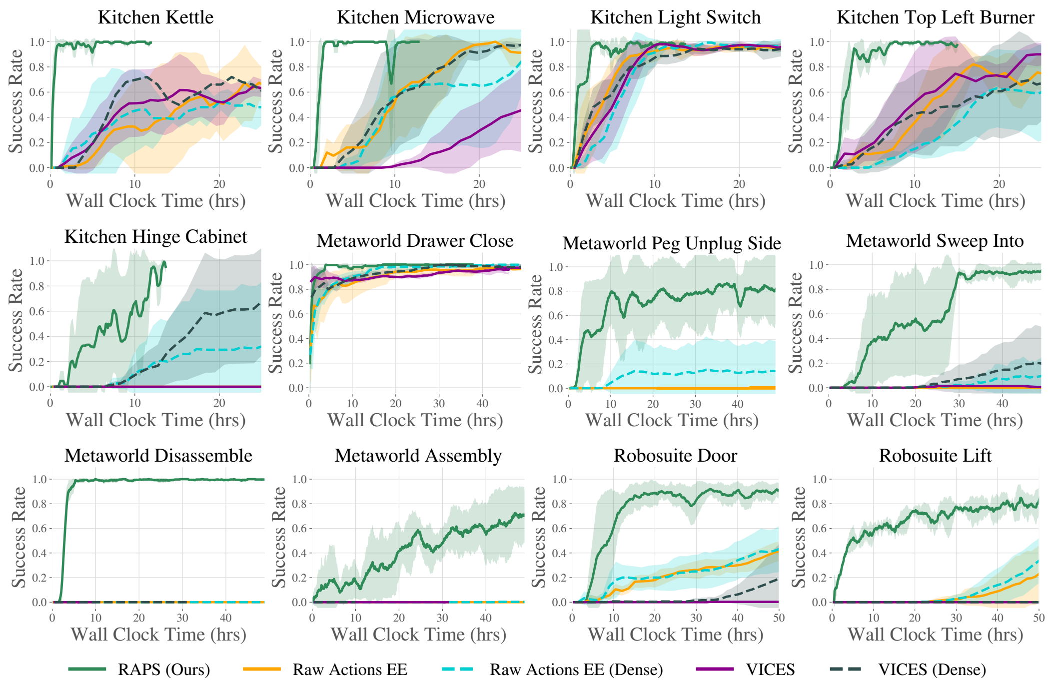


Figure 3: Comparison of various action parameterizations and RAPS across all three environment suites[2] using Dreamer as the underlying RL algorithm. RAPS (green), with sparse rewards, is able to significantly outperform all baselines, particularly on the more challenging tasks, even when they are augmented with dense reward. See the appendix for remaining plots on the `slide-cabinet` and `soccer-v2` tasks.

To ensure fair comparison across methods, we instead propose to perform evaluations with respect to two metrics, namely, (a) **Wall-clock Time**: the amount of total time it takes to train the agent to solve the task, both interaction time and time spent updating the agent, and (b) **Training Steps**: the number of gradient steps taken with a fixed batch size. Wall clock time is not inherently tied to the action space and provides an interpretable number for how long it takes for the agent to learn the task. To ensure consistency, we evaluate all methods on a single RTX 2080 GPU with 10 CPUs and 50GB of memory. However, this metric is not sufficient since there are several possible factors that can influence wall clock time which can be difficult to disambiguate, such as the effect of external processes, low-level controller execution speed, and implementation dependent details. As a result, we additionally compare methods based on the number of training steps, a proxy for data efficiency. The number of network updates is only a function of the data; it is independent of the action space, machine and simulator, making it a non-transient metric for evaluation. The combination of the two metrics provides a holistic method of comparing the performance of different action spaces and skills operating on varying frequencies and horizons.

**Baselines** The simplest baseline we consider is the default action space of the environment, which we denote as **Raw Actions**. One way to improve upon the raw action space is to train a policy to output the parameters of the underlying controller alongside the actual input commands. This baseline, **VICES** [30], enables the agent to tune the controller automatically depending on the task. Alternatively, one can use unsupervised skill extraction to generate higher level actions which can be leveraged by downstream RL. We evaluate one such method, **Dyn-E** [48], which trains an observation and action representation from random policy data such that the subsequent state is predictable from the embeddings of the previous observation and action. A more data-driven approach to learning skills involves organizing demonstration data into a latent skill space. Since the dataset is guaranteed to contain meaningful behaviors, it is more likely that the extracted skills will be useful for downstream tasks. We compare against **SPIRL** [34], a method that ingests a demonstration dataset to train a fixed length skill VAE $z = e(a_{1:H}), a_{1:H} = d(z)$ and prior over skills $p(z|s)$, which is used to guide downstream RL. Additionally, we compare against **PARROT** [43], which trains an observation conditioned flow model on an offline dataset to map from the raw action space to a latent action space.

[2] In all of our results, each plot shows a 95% confidence interval of the mean performance across three seeds.

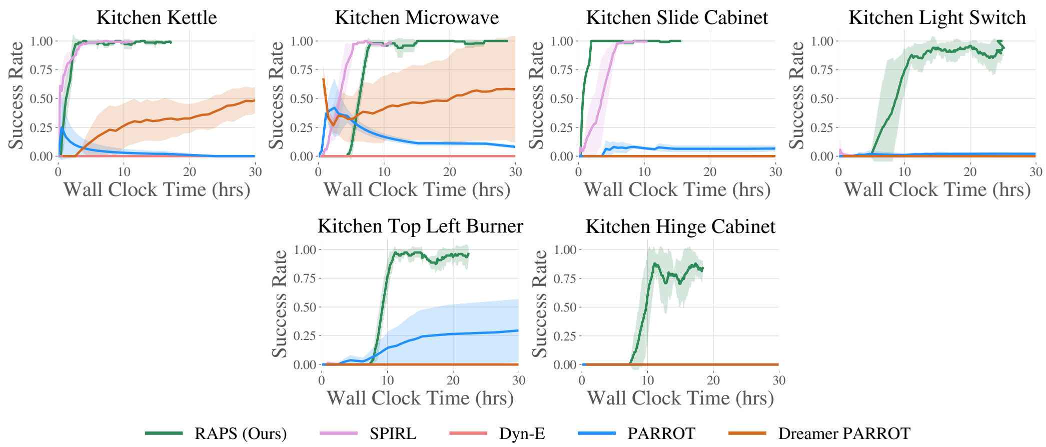


Figure 4: Comparison of RAPS and skill learning methods on the Kitchen domain using SAC as the underlying RL algorithm. While SPIRL and PARROT are competitive or even improve upon RAPS's performance on easier tasks, only RAPS (green) is able to solve `top-left-burner` and `hinge-cabinet`.

| RL Algorithm | Kettle | | Slide Cabinet | | Light Switch | | Microwave | | Top Burner | | Hinge Cabinet | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Raw | RAPS | Raw | RAPS | Raw | RAPS | Raw | RAPS | Raw | RAPS | Raw | RAPS |
| Dreamer | 0.8 | **.93** | 1.0 | 1.0 | 1.0 | 1.0 | .53 | **0.8** | .93 | **1.0** | 0.0 | **1.0** |
| SAC | .33 | **0.8** | .67 | **1.0** | **.86** | .67 | .33 | **1.0** | .33 | **1.0** | 0.0 | **1.0** |
| PPO | .33 | **1.0** | .66 | **1.0** | .27 | **1.0** | 0.0 | **.66** | .27 | **1.0** | 0.0 | **1.0** |

Table 1: Evaluation of RAPS across RL algorithms (Dreamer, PPO, SAC) on Kitchen. We report the final success rate of each method on five evaluation trials trained over three seeds from sparse rewards. While raw action performance (left entry) varies significantly across RL algorithms, RAPS (right entry) is able to achieve high success rates on *every* task with *every* RL algorithm.

In the next section, we demonstrate the performance of our RAPS against these methods across a diverse set of sparse reward manipulation tasks.

## 5 Experimental Evaluation of RAPS

We evaluate the efficacy of RAPS on three different settings: single task reinforcement learning across Kitchen, Metaworld and Robosuite, as well as hierarchical control and unsupervised exploration in the Kitchen environment. We observe across all evaluated settings, RAPS is robust, efficient and performant, in direct contrast to a wide variety of learned skills and action parameterizations.

### 5.1 Accelerating Single Task RL using RAPS

In this section, we evaluate the performance of RAPS against fixed and variable transformations of the lower-level action space as well as state of the art unsupervised skill extraction from demonstrations. Due to space constraints, we show performance against the number of training steps in the appendix.

**Action Parameterizations** We compare RAPS against Raw Actions and VICES using Dreamer [18] as the underlying algorithm across all three environment suites in Figure 3. Since we observe weak performance on the default action space of Kitchen, joint velocity control, we instead modify the suite to use 6DOF end-effector control for both raw actions and VICES. We find Raw Actions and VICES are able to make progress on a number of tasks across all three domains, but struggle to execute the fine-grained manipulation required to solve more difficult environments such as `hinge-cabinet`, `assembly-v2` and `disassembly-v2`. The latter two environments are not solved by Raw Actions or VICES even when they are provided dense rewards. In contrast, RAPS is able to quickly solve every task from sparse rewards.

On the kitchen environment, from sparse rewards, no prior method makes progress on the hardest manipulation task: grasping the hinge cabinet and pulling it open to 90 degrees, while RAPS is able to quickly learn to solve the task. In the Metaworld domain, `peg-unplug-side-v2`, `assembly-v2` and `disassembly-v2` are difficult environments which present a challenge to even dense reward state based RL [51]. However, RAPS is able to solve all three tasks with *sparse rewards* directly from image input. We additionally include a comparison of RAPS against Raw Actions on all 50 Metaworld tasks with final performance in the appendix. RAPS is able to learn to solve or make progress on **43 out of 50** tasks purely from sparse rewards. Finally, in the Robosuite domain, by leveraging robot action primitives, we are able to learn to solve the tasks more rapidly than raw actions or VICES, with respect to wall-clock time and number of training steps, demonstrating that RAPS scales to more realistic robotic controllers.

**Offline Learned Skills** An alternative point of comparison is to leverage offline data to learn skills and run downstream RL. We train SPIRL and PARROT from images using the kitchen demonstration datasets in D4RL [16], and Dyn-E with random interaction data. We run all agents with SAC as the underlying RL algorithm and extract learned skills using joint velocity control, the type of action present in the demonstrations. See Figure 4 for the comparison of RAPS against learned skills. Dyn-E is unable to make progress across any of the domains due to the difficulty of extracting useful skills from highly unstructured interaction data. In contrast, SPIRL and PARROT manage to leverage demonstration data to extract useful skills; they are competitive or even improve upon RAPS on the easier tasks such as `microwave` and `kettle`, but struggle to make progress on the more difficult tasks in the suite. PARROT, in particular, exhibits a great deal of variance across tasks, especially with SAC, so we include results using Dreamer as well. We note that both SPIRL and PARROT are limited by the tasks which are present in the demonstration dataset and unable to generalize their extracted skills to other tasks in the same environment or other domains. In contrast, parameterized primitives are able to solve *all* the kitchen tasks and are re-used across domains as shown in Figure 3.

**Generalization to different RL algorithms** A general set of skills maintains performance regardless of which RL method leverages them. In this section, we evaluate the performance of RAPS against Raw Actions on three types of RL algorithms: model based (Dreamer), off-policy model free (SAC) and on-policy model free (PPO) on the Kitchen tasks. We use the end-effector version of raw actions as a strong point of comparison on these tasks. As seen in Table 1, unlike raw actions, RAPS is agnostic to the underlying RL algorithm and maintains similarly high final performance across Dreamer, SAC and PPO. These experiments show that parameterized primitive policies generally improve the performance of RL. This result, along with the cross-domain results in Figure 3 suggests it may be feasible to directly apply RAPS to new environments and RL methods.

### 5.2 Enabling Hierarchical Control via RAPS

We next apply RAPS to a more complex setting: sequential RL, in which the agent must learn to solve multiple subtasks within a single episode, as opposed to one task. We evaluate on the Kitchen Multi-Task environments and plot performance across SAC, Dreamer, and PPO in Figure 5. Raw Actions prove to be a strong baseline, eventually solving close to three subtasks on average, while requiring significantly more wall-clock time and training steps. SPIRL initially shows strong performance but after solving one to two subtasks it then plateaus and fails to improve. PARROT is less efficient than SPIRL but also able to make progress on up to two subtasks, though it exhibits a great deal of sensitivity to the underlying RL algorithm. For both of the offline skill learning methods, they struggle to solve any of the subtasks outside of `kettle`, `microwave`, and `slide-cabinet` which are encompassed in the demonstration dataset. Meanwhile, with RAPS, across all three base RL algorithms, we observe that the agents are able to leverage the primitive library to rapidly solve three out of four subtasks and continue to improve. This result demonstrates that RAPS can elicit significant gains in hierarchical RL performance through its improved exploratory behavior.

### 5.3 Leveraging RAPS to enable efficient unsupervised exploration

In many realistic settings, even sparse rewards themselves can be hard to come by. Ideally, we would be able to train robot without train time task rewards for large periods of time and fine-tune to solve new tasks with only a few supervised labels. We use the kitchen environment to test the efficacy of primitives on the task of unsupervised exploration. We run an unsupervised exploration algorithm, Plan2explore [37], for a fixed number of steps to learn a world model, and then fine-tune the model and train a policy using Dreamer to solve specific tasks. We plot the results in Figure 6 on the `top-left-burner` and `hinge-cabinet` tasks. Primitives enable the agent to learn an effective world model that results in rapid learning of both tasks, requiring only **1 hour of fine-tuning** to solve the `hinge-cabinet` task. Meanwhile, the world model learned by exploring with raw actions is unable to quickly finetune as quickly. We draw two conclusions from these results, a) primitives

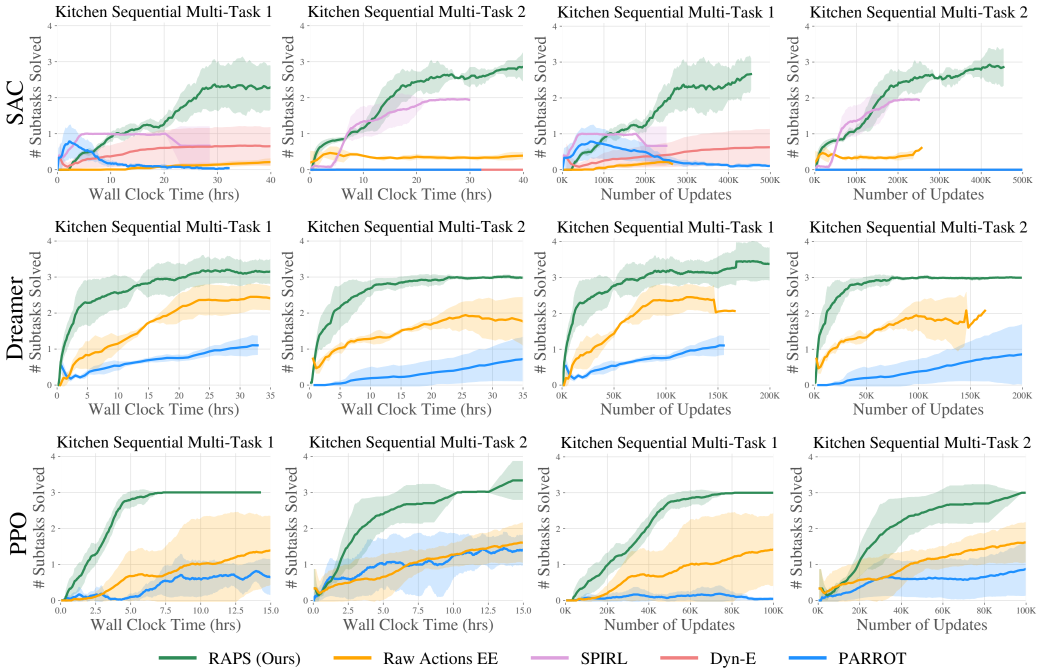


Figure 5: Learning performance of RAPS on sequential multi-task RL. Each row plots a different base RL algorithm (SAC, Dreamer, PPO) while the first two columns plot the two multi-task environment results against wall-clock time and the next two columns plot against number of updates, i.e. training steps. RAPS consistently solves at least three out of four subtasks while prior methods generally fail to make progress beyond one or two.

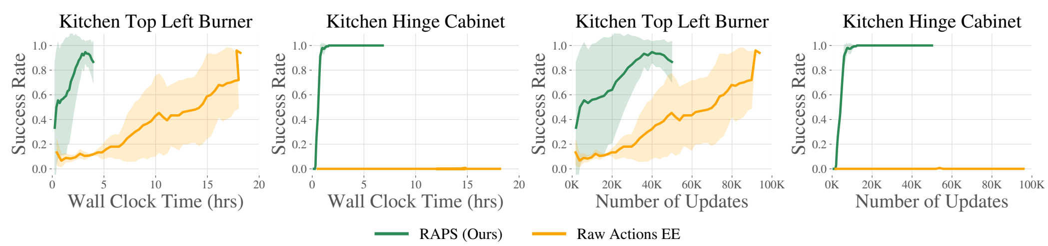


Figure 6: RAPS significantly outperforms raw actions in terms of total wall clock time and number of updates when fine-tuning initialized from reward free exploration.

enable more efficient exploration than raw actions, b) primitives facilitate efficient model fitting, resulting in rapid fine-tuning.

## 6 Discussion

In this work we present an extensive evaluation of RAPS, which leverages parameterized actions to learn high level policies that can quickly solve robotics tasks across three different environment suites. We show that standard methods of re-parameterizing the action space and learning skills from demonstrations are environment and domain dependent. In many cases, prior methods are unable to match the performance of robot action primitives. While primitives are not a general solution to every task, their success across a wide range of environments illustrates the utility of incorporating an agent-centric structure into the robot action space. Given the effectiveness of simple parameterized action primitives, a promising direction to further investigate would be how to best incorporate agent-centric structure into both learned and manually defined skills and attempt to get the best of both worlds in order to improve the interface of RL algorithms with robots.

# References


[1] J. Achiam, H. Edwards, D. Amodei, and P. Abbeel. Variational option discovery algorithms. *arXiv preprint arXiv:1807.10299*, 2018. 3

[2] A. Ajay, A. Kumar, P. Agrawal, S. Levine, and O. Nachum. Opal: Offline primitive discovery for accelerating offline reinforcement learning, 2021. 3, 5

[3] A. Allshire, R. Martín-Martín, C. Lin, S. Manuel, S. Savarese, and A. Garg. Laser: Learning a latent action space for efficient reinforcement learning. *arXiv preprint arXiv:2103.15793*, 2021. 2

[4] O. M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020. 1

[5] P.-L. Bacon, J. Harb, and D. Precup. The option-critic architecture. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017. 3

[6] S. Bahl, M. Mukadam, A. Gupta, and D. Pathak. Neural dynamic policies for end-to-end sensorimotor learning, 2020. 3

[7] A. Bemporad and M. Morari. Control of systems integrating logic, dynamics, and constraints. *Automatica*, 35(3):407–427, 1999. 3

[8] M. S. Branicky, V. S. Borkar, and S. K. Mitter. A unified framework for hybrid control: Model and optimal control theory. *IEEE transactions on automatic control*, 43(1):31–45, 1998. 3

[9] S. Cambon, R. Alami, and F. Gravot. A hybrid approach to intricate motion, manipulation and task planning. *The International Journal of Robotics Research*, 28(1):104–126, 2009. 3

[10] R. Chitnis, S. Tulsiani, S. Gupta, and A. Gupta. Efficient bimanual manipulation using learned task schemas. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1149–1155. IEEE, 2020. 3, 4

[11] C. Daniel, G. Neumann, O. Kroemer, J. Peters, et al. Hierarchical relative entropy policy search. *Journal of Machine Learning Research*, 17:1–50, 2016. 3

[12] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018. 3

[13] Z. Fan, R. Su, W. Zhang, and Y. Yu. Hybrid actor-critic reinforcement learning in parameterized action space. *arXiv preprint arXiv:1903.01344*, 2019. 3

[14] K. Frans, J. Ho, X. Chen, P. Abbeel, and J. Schulman. Meta learning shared hierarchies. *arXiv preprint arXiv:1710.09767*, 2017. 2, 3

[15] H. Fu, H. Tang, J. Hao, Z. Lei, Y. Chen, and C. Fan. Deep multi-agent reinforcement learning with discrete-continuous hybrid action spaces. *arXiv preprint arXiv:1903.04959*, 2019. 3

[16] J. Fu, A. Kumar, O. Nachum, G. Tucker, and S. Levine. D4rl: Datasets for deep data-driven reinforcement learning, 2021. 8

[17] A. Gupta, V. Kumar, C. Lynch, S. Levine, and K. Hausman. Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning, 2019. 5

[18] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi. Dream to control: Learning behaviors by latent imagination, 2020. 7

[19] M. Hausknecht and P. Stone. Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143*, 2015. 2, 3

[20] K. Hausman, J. T. Springenberg, Z. Wang, N. Heess, and M. Riedmiller. Learning an embedding space for transferable robot skills. In *International Conference on Learning Representations*, 2018. 2

[21] A. J. Ijspeert, J. Nakanishi, and S. Schaal. Learning attractor landscapes for learning motor primitives. Technical report, 2002. 3

[22] L. P. Kaelbling and T. Lozano-Pérez. Hierarchical task and motion planning in the now. In *2011 IEEE International Conference on Robotics and Automation*, pages 1470–1477. IEEE, 2011. 2, 3

[23] L. P. Kaelbling and T. Lozano-Pérez. Learning composable models of parameterized skills. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 886–893. IEEE, 2017. 3

[24] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, et al. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *arXiv preprint arXiv:1806.10293*, 2018. 1

[25] J. Kober and J. Peters. Learning motor primitives for robotics. In *2009 IEEE International Conference on Robotics and Automation*, pages 2112–2118. IEEE, 2009. 3

[26] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016. 1

[27] A. C. Li, C. Florensa, I. Clavera, and P. Abbeel. Sub-policy adaptation for hierarchical reinforcement learning. *arXiv preprint arXiv:1906.05862*, 2019. 2, 3

[28] C. Lynch and P. Sermanet. Grounding language in play. *arXiv preprint arXiv:2005.07648*, 2020. 3

[29] C. Lynch, M. Khansari, T. Xiao, V. Kumar, J. Tompson, S. Levine, and P. Sermanet. Learning latent plans from play. In *Conference on Robot Learning*, pages 1113–1132. PMLR, 2020. 2, 3

[30] R. Martín-Martín, M. A. Lee, R. Gardner, S. Savarese, J. Bohg, and A. Garg. Variable impedance control in end-effector space: An action space for reinforcement learning in contact-rich tasks, 2019. 6

[31] W. Masson, P. Ranchod, and G. Konidaris. Reinforcement learning with parameterized actions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016. 3

[32] O. Nachum, S. Gu, H. Lee, and S. Levine. Data-efficient hierarchical reinforcement learning. *arXiv preprint arXiv:1805.08296*, 2018. 2

[33] M. Neunert, A. Abdolmaleki, M. Wulfmeier, T. Lampe, T. Springenberg, R. Hafner, F. Romano, J. Buchli, N. Heess, and M. Riedmiller. Continuous-discrete reinforcement learning for hybrid control in robotics. In *Conference on Robot Learning*, pages 735–751. PMLR, 2020. 3

[34] K. Pertsch, Y. Lee, and J. J. Lim. Accelerating reinforcement learning with learned skill priors, 2020. 3, 5, 6

[35] J. Peters, K. Mulling, and Y. Altun. Relative entropy policy search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, 2010. 3

[36] S. Schaal. Dynamic movement primitives-a framework for motor control in humans and humanoid robotics. In *Adaptive motion of animals and machines*, pages 261–280. Springer, 2006. 3

[37] R. Sekar, O. Rybkin, K. Daniilidis, P. Abbeel, D. Hafner, and D. Pathak. Planning to explore via self-supervised world models, 2020. 8

[38] T. Shankar and A. Gupta. Learning robot skills with temporal variational inference. In *International Conference on Machine Learning*, pages 8624–8633. PMLR, 2020. 3

[39] T. Shankar, S. Tulsiani, L. Pinto, and A. Gupta. Discovering motor programs by recomposing demonstrations. In *International Conference on Learning Representations*, 2019. 2, 3

[40] A. Sharma, S. Gu, S. Levine, V. Kumar, and K. Hausman. Dynamics-aware unsupervised discovery of skills. *arXiv preprint arXiv:1907.01657*, 2019. 2, 3

[41] M. Sharma, J. Liang, J. Zhao, A. LaGrassa, and O. Kroemer. Learning to compose hierarchical object-centric controllers for robotic manipulation. *arXiv preprint arXiv:2011.04627*, 2020. 3

[42] A. Simeonov, Y. Du, B. Kim, F. R. Hogan, J. Tenenbaum, P. Agrawal, and A. Rodriguez. A long horizon planning framework for manipulating rigid pointcloud objects. *arXiv preprint arXiv:2011.08177*, 2020. 3

[43] A. Singh, H. Liu, G. Zhou, A. Yu, N. Rhinehart, and S. Levine. Parrot: Data-driven behavioral priors for reinforcement learning. *arXiv preprint arXiv:2011.10024*, 2020. 3, 6

[44] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999. 3

[45] D. Tanneberg, K. Ploeger, E. Rueckert, and J. Peters. Skid raw: Skill discovery from raw trajectories. *IEEE Robotics and Automation Letters*, 6(3):4696–4703, 2021. 3

[46] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning*, pages 3540–3549. PMLR, 2017. 2

[47] E. Wei, D. Wicke, and S. Luke. Hierarchical approaches for reinforcement learning in parameterized action space. *arXiv preprint arXiv:1810.09656*, 2018. 3

[48] W. Whitney, R. Agarwal, K. Cho, and A. Gupta. Dynamics-aware embeddings, 2020. 3, 6

[49] K. Xie, H. Bharadhwaj, D. Hafner, A. Garg, and F. Shkurti. Latent skill planning for exploration and transfer. In *International Conference on Learning Representations*, 2020. 2

[50] J. Xiong, Q. Wang, Z. Yang, P. Sun, L. Han, Y. Zheng, H. Fu, T. Zhang, J. Liu, and H. Liu. Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space. *arXiv preprint arXiv:1810.06394*, 2018. 3

[51] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, C. Finn, and S. Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning*, pages 1094–1100. PMLR, 2020. 5, 8

[52] W. Zhou, S. Bajracharya, and D. Held. Plas: Latent action space for offline reinforcement learning. *arXiv preprint arXiv:2011.07213*, 2020. 3

[53] Y. Zhu, J. Wong, A. Mandlekar, and R. Martín-Martín. robosuite: A modular simulation framework and benchmark for robot learning, 2020. 5