COMPOSING COMPLEX SKILLS BY LEARNING TRANSITION POLICIES WITH PROXIMITY REWARD INDUCTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Intelligent creatures acquire complex skills by exploiting previously learned skills and learning to transition between them. To empower machines with this ability, we propose transition policies which effectively connect primitive skills to perform sequential tasks without handcrafted rewards. To effectively train our transition policies, we introduce proximity predictors which induce rewards gauging proximity to suitable initial states for the next skill. The proposed method is evaluated on a diverse set of experiments for continuous control in both bipedal locomotion and robotic arm manipulation tasks in MuJoCo. We demonstrate that transition policies enable us to effectively learn complex tasks and the induced reward computed using the proximity predictor improves training efficiency. Videos of policies learned by our algorithm and baselines can be found at `https://sites.google.com/view/transitions-iclr2019`.

1 INTRODUCTION

While humans are capable of learning complex skills by reusing previously acquired skills, composing and mastering complex tasks is not as trivial as sequentially executing those previously learned skills. Instead, it requires a smooth transition between skills since the final pose of one skill may not be appropriate to initiate the following one. For example, while scoring in basketball with a quick shot after receiving a pass can be simply decomposed into catching and shooting, it is still difficult for beginners who have learned how to catch passes and statically shoot. To master this skill, players must practice adjusting their footwork and body into a comfortable shooting pose after catching a pass.

Similarly, can machines master new and complex tasks by reusing acquired skills and learning transitions between them? Learning to perform composite and long-term tasks from scratch requires extensive exploration and sophisticated reward design, which can introduce undesired behaviors (Riedmiller et al., 2018). Thus, instead of employing intricate reward functions and learning from scratch, modular methods sequentially execute acquired skills with a rule-based meta-policy, enabling machines to solve complicated tasks (Pastor et al., 2009; Mülling et al., 2013; Andreas et al., 2017). These modular approaches assume that a task can be clearly decomposed into several subtasks which are smoothly connected to each other. In other words, an ending state of one subtask falls within the set of starting states, *initiation set*, of the next subtask (Sutton et al., 1999). However, this assumption does not hold in many continuous control problems where a given skill may be executed in starting states not considered during training or designing and thus, fail to achieve its goal.

To bridge the gap between skills, we propose a modular framework that executes skills sequentially and employs *transition policies*, which smoothly navigate from an ending state of a skill to suitable initial states of the following skill. However, learning a transition policy between skills without reward shaping is difficult as the only available learning signal is the sparse reward for the successful execution of the next skill. Sparse success/failure reward is challenging to learn from due to the temporal credit assignment problem (Sutton, 1984) and the lack of information from failing trajectories. To alleviate these problems, we propose an *proximity predictor* which outputs the proximity to the initiation set of the next skill and acts as a dense reward function for the transition policy.

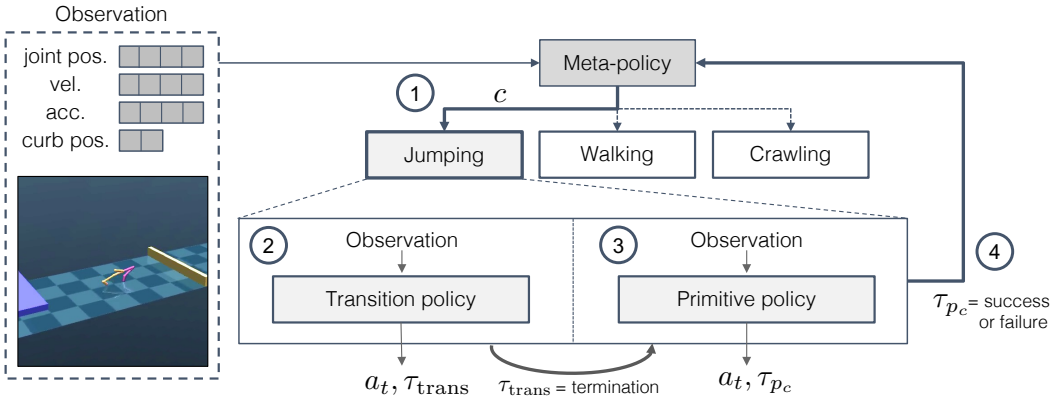


Figure 1: Our modular network augmented with transition policies. To perform a complex task, our model repeats the following steps: (1) The meta-policy chooses a primitive policy of index c; (2) The corresponding transition policy helps initiate the chosen primitive policy; (3) The primitive policy executes the skill; and (4) A success or failure signal for the primitive skill is produced.

The main contributions of this paper include a novel modular framework that reuses skills by employing transition policies to connect skills and a joint training algorithm with the proximity predictor specifically designed for transition policies. This framework is suited for learning complex skills that require sequential execution of primitive skills, which are common in the real world. Our experiments on simulated environments demonstrate that employing transition policies addresses complex continuous control tasks which traditional policy gradient methods struggle at because transitions allow the agent to utilize a diverse set of known skills given only sparse rewards. We will release our environments and primitive skills as well as the code for further research.

2 Related Work

Learning continuous control of diverse behaviors in locomotion (Merel et al., 2017; Heess et al., 2017; Peng et al., 2017) and robotic manipulation (Ghosh et al., 2018) is an active research area in reinforcement learning (RL). While some complex tasks can be solved through extensive reward engineering (Ng et al., 1999), undesired behaviors often emerge (Riedmiller et al., 2018) when tasks require several different primitive skills, and training from scratch is not computationally practical.

Real-world tasks often require diverse behaviors and longer temporal dependencies. In hierarchical reinforcement learning, the option critic framework (Sutton et al., 1999) learns meta actions (options), a series of primitive actions over a period of time, but assumes a global initiation set for all options. Typically, a hierarchical reinforcement learning framework consists of two components: a high-level meta-controller and low-level controllers. A meta-controller determines the order of subtasks to achieve the final goal and chooses corresponding low-level controllers that generate a sequence of primitive actions. Unsupervised approaches to discover meta actions have been proposed recently (Bacon et al., 2017; Daniel et al., 2016; Vezhnevets et al., 2017; Dilokthanakul et al., 2017; Frans et al., 2018; Co-Reyes et al., 2018; Levy et al., 2017). However, to deal with more complex tasks, additional supervision signals (Andreas et al., 2017; Tianmin Shu, 2018; Merel et al., 2017) or pre-defined low-level controllers (Kulkarni et al., 2016; Oh et al., 2017) are required.

To exploit pre-trained modules as low-level controllers, neural module networks (Andreas et al., 2016) have been proposed, which construct a new network dedicated to a given query using a collection of reusable modules. In the RL domain, a meta-controller is trained to follow instructions (Oh et al., 2017), demonstrations (Xu et al., 2017) and support multi-level hierarchies (Gudimella et al., 2017). In the robotics domain, Pastor et al. (2009); Kober et al. (2010); Mülling et al. (2013) propose a modular approach that learns table tennis by selecting appropriate low-level controllers. On the other hand, Andreas et al. (2017); Frans et al. (2018) learn abstract skills while experiencing a distribution of tasks and then solve a new task with the learned primitive skills. However, these modular approaches result in undefined behavior when two skills are not smoothly connected. Our

Algorithm 1 ROLLOUT

Input: meta policy π_{meta}, primitive policies $\{\pi_{p_1}, ..., \pi_{p_n}\}$, transition policies $\{\pi_{\phi_1}, ..., \pi_{\phi_n}\}$, and proximity predictors $\{D_{\omega_1}, ..., D_{\omega_n}\}$
Initialize an episode and receive initial state s_0
Initialize rollout buffers $\mathcal{B}_1, ..., \mathcal{B}_n$
repeat
  Choose a primitive policy $c \sim \pi_{\text{meta}}(s_t)$
  repeat
    Run transition policy $a_t, \tau_t^{\text{trans}} \sim \pi_{\phi_c}(s_t)$
    $s_{t+1} = \text{ENV}(s_t, a_t)$
    $r_t = D_{\omega_c}(s_{t+1}) - D_{\omega_c}(s_t)$
    Store $(s_t, a_t, r_t, \tau_t^{\text{trans}}, s_{t+1})$ in $\mathcal{B}_c$
  until τ_t^{trans} is $Continue$, $t = t + 1$
  repeat
    Run primitive policy $a_t, \tau_t^{p_c} \sim \pi_{p_c}(s_t)$
    $s_{t+1} = \text{ENV}(s_t, a_t)$
  until $\tau_t^{p_c}$ is $Continue$, $t = t + 1$
  Label a transition trajectory with τ_{t-1}
until episode termination
Return $\mathcal{B}_1, \ldots, \mathcal{B}_n$

Algorithm 2 TRAIN

Input: primitive polices $\{\pi_{p_1}, ..., \pi_{p_n}\}$
Initialize success buffers $\{\mathcal{B}_1^{\text{S}}, ..., \mathcal{B}_n^{\text{S}}\}$ with successful trajectories of primitive policies
Initialize failure buffers $\{\mathcal{B}_1^{\text{F}}, ..., \mathcal{B}_n^{\text{F}}\}$
Randomly initialize parameters of transition policies $\{\phi_1, ..., \phi_n\}$ and initiation predictors $\{\omega_1, ..., \omega_n\}$.
repeat
  Collect trajectories using ROLLOUT
  for $i = 1$ **to** n **do**
    Add trajectories of π_{ϕ_i} to $\mathcal{B}_i^{\text{S}}$ and $\mathcal{B}_i^{\text{F}}$
    Update D_{ω_i} to minimize Equation (1) using samples from $\mathcal{B}_i^{\text{S}}$ and $\mathcal{B}_i^{\text{F}}$
    Update π_{ϕ_i} to maximize Equation (2)
  end for
until convergence

proposed framework aims to bridge this gap by training model-free transition policies to navigate the agent from unseen states for following skills to suitable initial states.

Deep RL techniques for continuous control demand dense reward signals; otherwise, they suffer from long training time. Instead of manual reward shaping for denser reward, adversarial reinforcement learning (Ho & Ermon, 2016; Merel et al., 2017; Wang et al., 2017; Bahdanau et al., 2018) employs a discriminator which learns to judge the state or the policy, and the policy takes as rewards the output of the discriminator. While those methods assume ground truth trajectories or goal states are given, our method collects both success and failure trajectories online to train proximity predictors which provide rewards for transition policies.

3 APPROACH

In this paper, we address the problem of solving a complex task, which requires sequential composition of primitive skills $\{p_1, p_2, \ldots, p_n\}$, given only *sparse and binary rewards* (i.e. subtask completion reward). While our method is agnostic to the form of primitive policies (e.g. rule based, inverse kinematics, etc.), we consider the case of a pre-trained neural network in this paper. To learn a complex task, we propose a modular framework that is able to exploit the given primitive skills with smooth composition by employing *transition policies*.

3.1 PRELIMINARIES

We formulate our problem as a Markov decision process defined by a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{T}, R, \rho, \gamma\}$ of states, actions, transition probability, reward, initial state distribution, and discount factor. An action distribution of an agent is represented as a policy $\pi_\theta(a_t|s_t)$, where $s_t \in \mathcal{S}$ is the current state, $a_t \in \mathcal{A}$ is an action, and θ are the parameters of the policy. An initial state s_0 is randomly sampled from ρ, and then, an agent iteratively takes an action a_t sampled from a policy $\pi_\theta(a_t|s_t)$ and receives a reward r_t until the episode ends. The performance of the agent is evaluated based on a discounted return $R = \sum_{t=0}^{T} \gamma^t r_t$, where T is the episode horizon.

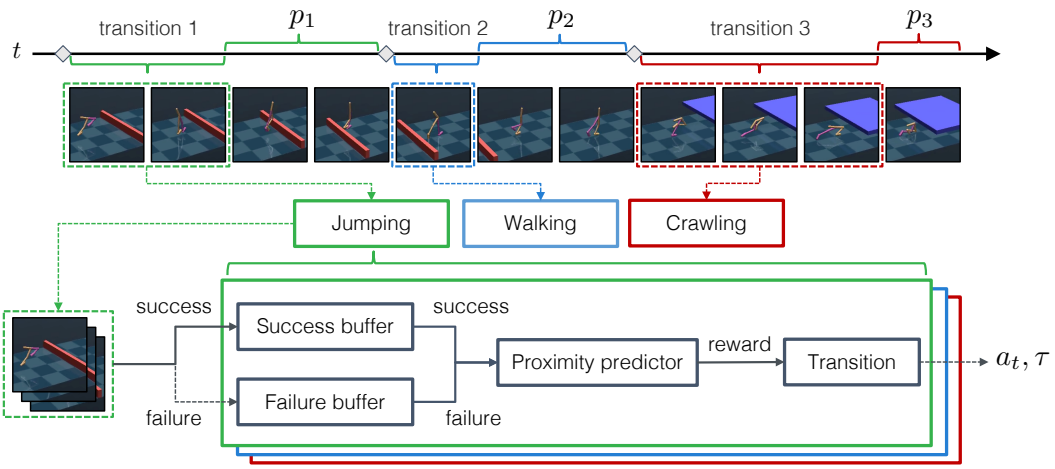


Figure 2: Training of transition policies and proximity predictors. After executing a primitive policy, a previously performed transition trajectory is labeled and added to a replay buffer based on the execution success. A proximity predictor is trained on states sampled from the two buffers to output 0 and 1 for failing and successful states. and serves as a reward function to encourage the transition policy to move toward good initial states to initiate the corresponding primitive policy.

3.2 Modular Network

To learn a new task given learned primitive polices, we design a modular network that consists of the following components: a meta-policy $\pi_{meta}(\cdot|s)$, primitive policies $\{\pi_{p_1}(a|s), \dots, \pi_{p_n}(a|s)\}$, and transition policies $\{\pi_{\phi_1}(a|s), \dots, \pi_{\phi_n}(a|s)\}$ as illustrated in Figure 1 and Algorithm 1.

The meta-policy chooses the next primitive policy to execute when the current primitive policy is terminated. The action space of the meta-policy is a set of primitive policy indexes $\{1, 2, \dots, n\}$. The observation of the meta-policy contains the low-level information of primitives and task specifications indicating high-level goals (e.g. moving direction and target object position). In this paper, we use a rule-based meta-policy and focus on transitioning between consecutive primitive policies.

Once a primitive policy p_c is chosen to be executed, the agent generates an action $a_t \in \mathcal{A}$ based on the current state $s_t \in \mathcal{S}$, which consists of joint configurations. Note that we did not differentiate state spaces for primitive polices because of the simplicity of notations (e.g. the observation of the jumping primitive contains a distance to a curb while that of the walking primitive only has joint pose and velocities). Every primitive policy is required to generate termination signals $\tau_{p_c} \in \{\text{continue}, \text{success}, \text{fail}\}$ to indicate policy completion and whether the execution is successful or not. Note that the primitive policies can be any form, such as neural network and rule-based policy.

For smooth transitions between primitive policies, we add a transition policy $\pi_{\phi_c}(a|s)$ before executing primitive policy p_c, which guides an agent to p_c's initiation set. The transition policy's observation and action space are the same as the primitive policy's. The transition policy also learns a termination signal τ_{trans} which indicates transition termination to successfully initiate the following primitive policy.

3.3 Training transition policies

Transitioning between a pair of primitive policies (p_i, p_j) can be considered as moving from end states of the former primitive policy p_i to the initiation set of the subsequent primitive policy p_j. After the transition is completed, p_j proceeds with the execution of the primitive skill. The transition policy for p_j is shared across different preceding primitive policies where a successful transition is set by the success of the following primitive policy p_j. For brevity of notation, we omit the primitive policy index j in the following equations where unambiguous.

Transition policies are difficult to train as it is hard to learn a good mapping between end states and viable initial states in a continuous space. Furthermore, by definition, the only available learning

signal for the transition policies is the sparse reward for the completion of the next task. To alleviate the sparsity of rewards and to maximize the objective of moving to viable initial states for the next primitive, we propose an *proximity predictor* that learns and provides a dense reward of how close transition states are to the initiation set of the corresponding primitives, illustrated in Figure 2. We denote an proximity predictor as D_{ω_j} which is parameterized by ω_j. We define the proximity of a state as the future discounted proximity, $v = \delta^{step}$, where $step$ is the number of steps required to reach an initiation set of the following primitive policy. The proximity of transition states in the initiation set is 1 because the initiation set leads to successful execution of the respective primitive policy. We train the proximity predictor by minimizing the following objective:

$$L_D(\omega, \mathcal{B}^S, \mathcal{B}^F) = \frac{1}{2}\mathbb{E}_{(s,v)\sim\mathcal{B}^S}[(D_\omega(s) - v)^2] + \frac{1}{2}\mathbb{E}_{s\sim\mathcal{B}^F}[D_\omega(s)^2], \tag{1}$$

where $\mathcal{B}^S$ and $\mathcal{B}^F$ are buffers that contain sets of states collected from success and failure trajectories, respectively. To estimate the proximity to an initiation set, $\mathcal{B}^S$ contains not only the state that directly lead to the success of the following primitive policy, but also the intermediate states of the successful trajectories with its proximity. By minimizing this objective, given a state, the proximity predictor is learned to predict 1 if the state is in the initiation set, a value that is between 0 and 1 if the state leads the agent to end up with a desired initial states, and 0 when the state leads to a failure.

The goal of a transition policy can be formulated as seeking a state s that is likely to be considered as a plausible initial state by the proximity predictor (i.e. D(s) is close to 1). To achieve this goal, the transition policy learns to maximize proximity prediction at the end state of the transition trajectory $D_\omega(s_T)$. In addition to providing reward at the end, we also use the increase of proximity to the initiation set $D_\omega(s_{t+1}) - D_\omega(s_t)$ at every timestep as a reward, dubbed *proximity reward*, to create a denser reward. The transition policy is trained to maximize the expected discounted return:

$$R_{\text{trans}}(\phi) = \mathbb{E}\Big[\gamma^T D_\omega(s_T) + \sum_{t=0}^{T-1} \gamma^t (D_\omega(s_{t+1}) - D_\omega(s_t))\Big]. \tag{2}$$

In addition to predicting an action distribution, the transition policy also predicts a termination signal $\tau \sim \pi_\phi(s)$ that decides whether the current state s is a plausible initial state for the following skill.

However, in this scenario, ground truth states in $\mathcal{B}^S$ and $\mathcal{B}^F$ are not available. Hence, we collect training data for an proximity predictor online by utilizing the trajectories obtained during execution of the corresponding transition policy and primitive policy. Specifically, we label the states in a transition trajectory as success or failure based on whether the following primitive is successfully executed or not, and add them into the corresponding buffers $\mathcal{B}^S$ or $\mathcal{B}^F$, respectively. As stated in Algorithm 2, we train transition policies and proximity predictors by alternating between an Adam (Kingma & Ba, 2014) gradient step on ω to minimize Equation (1) with respect to D_ω and a PPO (Schulman et al., 2017) step on ϕ to maximize Equation (2) with respect to π_ϕ. We refer readers to the supplementary for further details on training.

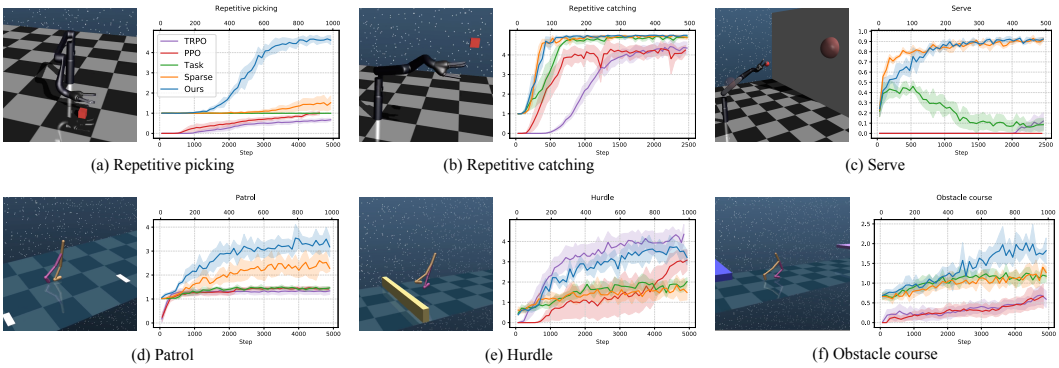
Utilizing the learned reward for training transition policies is beneficial in several perspectives: (1) the proximity predictor provides how proximate the current state is to suitable initial states; (2) the dense rewards speed up transition policy training by differentiating failing states from states in a successful trajectory; and (3) the joint training mechanism prevents a transition policy from getting stuck in local optima. Whenever a transition policy gets into a local optimum (i.e. fails the following skill with a high proximity reward), the proximity predictor learns to lower the reward for the failing transition as those states are added to its failure buffer, escaping the local optimum.

4 EXPERIMENTS

We conducted experiments on two classes of tasks: locomotion and robotic manipulation. To illustrate the potential of the proposed framework, we designed a set of complex tasks that require agents to utilize a diverse library of primitive skills described in the following sections. All of our environments are simulated in the MuJoCo physics engine (Todorov et al., 2012).

4.1 BASELINES

We evaluate our method to answer how transition policies benefit complex task learning and the effectiveness of joint training of transition policies and proximity predictors. To investigate the



(a) Repetitive picking (b) Repetitive catching (c) Serve

(d) Patrol (e) Hurdle (f) Obstacle course

Figure 3: Tasks and success count curves of our model (blue), TRPO (purple), PPO (magenta), and transition policies trained on task reward (green) and sparse proximity reward (yellow). Our model achieves the best performance and convergence time. Different temporal scales are used for TRPO and PPO (bottom) and ours (top). TRPO and PPO are trained 5 times longer than ours.

Table 1: Success count for robotic manipulation, comparing our method with baselines.

	Reward	Repetitive picking	Repetitive catching	Serve
TRPO	dense	0.69 ± 0.46	4.54 ± 1.21	0.32 ± 0.47
PPO	dense	0.95 ± 0.53	4.26 ± 1.63	0.00 ± 0.00
Without transition	sparse	0.99 ± 0.08	1.00 ± 0.00	0.11 ± 0.32
Transition + task reward	sparse	0.99 ± 0.08	4.87 ± 0.58	0.05 ± 0.21
Transition + sparse proximity reward	sparse	1.52 ± 1.12	4.88 ± 0.59	$\mathbf{0.92 \pm 0.27}$
Transition + proximity reward (ours)	sparse	$\mathbf{4.84 \pm 0.63}$	$\mathbf{4.97 \pm 0.33}$	$\mathbf{0.92 \pm 0.27}$

impact of the transition policy, we compare policies learned from dense rewards with our model that learns from sparse and binary rewards by exploiting primitive policies. Moreover, we conducted ablation studies to dissect each component in the training method of transition polices:

- **Trust Region Policy Optimization (TRPO) with dense reward** represents a state-of-the-art policy gradient method, which we use for the standard RL comparison.
- **Proximal Policy Optimization (PPO) with dense reward** is another state-of-the-art policy gradient method, which is more stable than TRPO with small batch sizes.
- **Modular Net without transition policies** sequentially executes primitive policies without transition policies and has no learnable components.
- **Modular Net with transition policies trained on task rewards** represents a modular network augmented with transition policies learned from the sparse and binary reward (subtask completion reward), whereas our model learns from the reward predictor.
- **Modular Net with transition policies trained on sparse proximity rewards** is a sparse variant of our model where the proximity reward is provided only at the end of the transition trajectory.

Initially, we tried comparing baseline methods with our method using only sparse and binary rewards. However, the baselines could not solve any of our environments due to the complex tasks and sparse reward of the environments. To get a more competitive comparison, we hand engineer dense rewards for baselines to boost their performance and show that transitions with sparse rewards can compete with and even outperform dense reward baselines. As the performance of policy gradient methods like TRPO and PPO varies significantly between runs, we perform each experiment with 3 random seeds and report mean and standard deviation in Figure 3.

4.2 Robotic Manipulation

For robotic manipulation, we simulate the Kinova Jaco, a 9 DoF robotic arm with 3 fingers. The agent receives full state information, including the absolute location of external objects. The agent uses joint torque control to perform actions. The results are shown in Figure 3 and Table 1.

Table 2: Success count for locomotion, comparing our method with baselines. Hurdle's TRPO reward was extensively engineered but our method is comparable with sparse reward*.

	Reward	Patrol	Hurdle	Obstacle course
TRPO	dense	1.37 ± 0.52	$\mathbf{4.13 \pm 1.54}$	0.98 ± 1.09
PPO	dense	1.53 ± 0.53	2.87 ± 1.92	0.85 ± 1.07
Without transition	sparse	1.02 ± 0.14	0.49 ± 0.75	0.72 ± 0.72
Transition + task reward	sparse	1.69 ± 0.63	1.73 ± 1.28	1.08 ± 0.78
Transition + sparse proximity reward	sparse	2.51 ± 1.26	1.47 ± 1.53	1.32 ± 0.99
Transition + proximity reward (ours)	sparse	$\mathbf{3.33 \pm 1.38}$	$\mathbf{3.14 \pm 1.69}$*	$\mathbf{1.90 \pm 1.45}$

Pre-trained primitives. There are four pre-trained primitives available: *Picking*, *Catching*, *Tossing*, and *Hitting*. *Picking* requires the robotic arm to pick up a small block, which is randomly placed on the table. If the box is not picked after a certain amount of time, the agent fails. *Catching* learns to catch a block that is thrown towards the arm with random initial position and velocity. The agent fails if it does not catch and stably hold the box for a certain amount of time. *Tossing* requires the robot to pick up a box, toss it vertically in the air, and land the box at a specified position. *Hitting* requires the robot to hit a box dropped overhead at a target.

Repetitive picking. The *Repetitive picking* task requires the agent to complete the *Picking* task 5 times. After a successful pick, the box disappears and is placed randomly on the table again. Our model achieves the best performance and converges the fastest by learning from the proximity reward. With our dense proximity reward at every transition step, we alleviate credit assignment when compared to providing a proximity reward at the end of the trajectory or using sparse task reward. Conversely, TRPO with dense rewards takes significantly longer to learn and is unable to pick the second box as the ending pose after the first pick is too unstable to initialize the next picking.

Repetitive catching. Similar to *Repetitive picking*, the *Repetitive catching* task requires the agent to catch boxes consecutively up to 5 times. In this task, other than the modular network without a transition policy, all baselines are able to eventually learn while our model still learns the fastest. We believe this is because the *Catching* primitive policy has a larger initiation set and therefore, the sparse reward problem is alleviated.

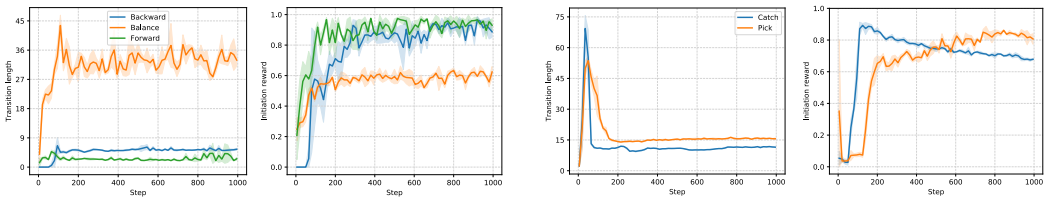
Serve Inspired by tennis, *Serve* requires the robot to toss the ball and hit it at a target. Even with an extensively engineered reward, TRPO and PPO baselines fail to learn because *Hit* needs to learn to cover all terminal states of the *Toss* primitive. On the other hand, learning to recover from *Toss*'s ending states to *Hit*'s initiation set is easier for exploration, which reduces the complexity of the task. In this task, our method and the modular network with sparse proximity reward baseline are able to solve it. We believe this is because the *Hit* primitive has a large initiation set so the transition policy could explore without dense proximity reward.

4.3 LOCOMOTION

For locomotion, we simulate a 9 DoF planar (2D) bi-pedal robot. The observation of the agent includes joint position, rotation, and velocity. When the agent needs to interact with environment, we provide additional input such as distance to the curb and ceiling in front of the agent. The agent uses joint torque control to perform actions. The results are shown in Figure 3 and Table 2.

Pre-trained primitives. *Forward* and *Backward* require the agent to walk forward and backward with a certain velocity, respectively. *Balancing* enables the walker to be robust under the random external forces. *Jumping* lets the walker jump over a randomly located curb and land safely. *Crawling* requires the walker to crawl under a ceiling. In all the aforementioned tasks, the walker fails when the height of the walker is lower than a threshold.

Patrol (Forward and backward). The *Patrol* task involves walking forward and backward toward goal points on either side and balancing in between to smoothly change its direction. As illustrated in Figure 3, our method consistently outperforms TRPO and ablated baselines in stably walking forward and transitioning to walk backward. The agent trained with dense reward is not able to consistently switch directions, whereas our model can utilize previously learned primitives including *Balancing* to stabilize a reversal in velocity.



(a) PATROL (b) MANIPULATION

Figure 4: Average transition length and average proximity reward of transition trajectories over training on *Patrol* (left) and *Manipulation* (right).

Hurdle (Walking forward and jumping). The *Hurdle* task requires the agent to walk forward and jump across curbs and requires a transition between walking and jumping as well as landing the jump to walking forward. As shown in Figure 3, our method outperforms the sparse reward baselines, but TRPO with dense reward can learn this task as well. We extensively design dense rewards for competitive baselines and for the *Hurdle* task, the dense reward consists of eight components, which collectively enable TRPO to learn the task. With an intricately designed dense reward, difficult tasks can be trained with RL baselines. However, our focus is to learn a complex task by reusing acquired skills, avoiding an extensive reward design. Our model with sparse reward and prior knowledge is still able to learn this complex task in comparison to other sparse reward baselines.

Obstacle Course (Walking forward, jumping, and crawling). *Obstacle Course* is the most difficult among the locomotion tasks, where the walker must walk forward, jump across curbs, and crawl underneath ceilings. It requires three different behaviors and transitions between two very different primitive skills: crawling and jumping. Since the task requires significantly different behaviors that are hard to transition between, TRPO fails to learn the task and only tries to crawl toward the curb without attempting to jump. In contrast, our method learns to transition between all pairs of primitive skills and succeeds in crossing multiple obstacles.

4.4 ABLATION STUDY

We conducted additional experiments to understand the contribution of transition policies, proximity predictors, and soft labeling for intermediate states of successful trajectories. Figure 3 shows the gain from each component. The modular network without transition policies tends to fail the execution of the second skill as sequential executions of primitives is not smooth, and the next primitive policy is not trained to cover ending states of the first primitive. Transition policies trained from task completion reward learn to connect consecutive primitives slower as sparse task reward is hard to learn from due to the credit assignment problem. On the other hand, our model alleviates credit assignment and learns quickly by giving predicted rewards for every transition state-action pair.

4.5 TRAINING OF TRANSITION AND PROXIMITY PREDICTOR

In *Patrol* task, transition lengths for *Forward, Balance, Backward* primitives are 5, 33, and 3, respectively. Figure 4 shows that at first the proximity rewards for transition policies increase directly with the transition lengths since transition policies are exploring unseen states with high proximity rewards. However, as failing initial states with high proximity are collected in the failure buffers, the proximity predictor learns to distinguish good and bad initial states.

5 CONCLUSION

In this work, we propose a modular framework with transition policies to empower reinforcement learning agents to learn complex tasks with sparse reward. Specifically, we formulate the problem as executing existing primitives while smoothly transitioning between primitives and propose a joint training method to train transition policies with proximity predictors. Our experimental results on robotic manipulation and locomotion tasks demonstrate the effectiveness of employing transition policies. The proposed framework solves complex tasks without reward shaping and outperforms baseline RL algorithms and other ablated baselines that utilize prior knowledge on many tasks.

REFERENCES

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 39–48, 2016.

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, 2017.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *AAAI*, pp. 1726–1734, 2017.

Dzmitry Bahdanau, Felix Hill, Jan Leike, Edward Hughes, Pushmeet Kohli, and Edward Grefenstette. Learning to follow language instructions with adversarial reward induction, 2018.

John Co-Reyes, YuXuan Liu, Abhishek Gupta, Benjamin Eysenbach, Pieter Abbeel, and Sergey Levine. Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings. In *Proceedings of the 35th International Conference on Machine Learning*, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018.

Christian Daniel, Herke Van Hoof, Jan Peters, and Gerhard Neumann. Probabilistic inference for determining options in reinforcement learning. *Machine Learning*, 104(2-3):337–357, 2016.

Nat Dilokthanakul, Christos Kaplanis, Nick Pawlowski, and Murray Shanahan. Feature control as intrinsic motivation for hierarchical reinforcement learning. *arXiv preprint arXiv:1705.06769*, 2017.

Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=SyX0IeWAW`.

Dibya Ghosh, Avi Singh, Aravind Rajeswaran, Vikash Kumar, and Sergey Levine. Divide and conquer reinforcement learning. In *International Conference on Learning Representations*, 2018.

Aditya Gudimella, Ross Story, Matineh Shaker, Ruofan Kong, Matthew Brown, Victor Shnayder, and Marcos Campos. Deep reinforcement learning for dexterous manipulation with concept networks. *CoRR*, abs/1709.06977, 2017. URL `http://arxiv.org/abs/1709.06977`.

Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin A. Riedmiller, and David Silver. Emergence of locomotion behaviours in rich environments. *CoRR*, abs/1707.02286, 2017. URL `http://arxiv.org/abs/1707.02286`.

Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems*, pp. 4565–4573, 2016.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Jens Kober, Katharina Mülling, Oliver Krömer, Christoph H Lampert, Bernhard Schölkopf, and Jan Peters. Movement templates for learning of hitting and batting. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pp. 853–858. IEEE, 2010.

Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pp. 3675–3683, 2016.

Andrew Levy, Robert Platt, and Kate Saenko. Hierarchical actor-critic. *arXiv preprint arXiv:1712.00948*, 2017.

Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, Zhen Wang, and Stephen Paul Smolley. Least squares generative adversarial networks. In *International Conference on Computer Vision*, 2017.

Josh Merel, Yuval Tassa, Dhruva TB, Sriram Srinivasan, Jay Lemmon, Ziyu Wang, Greg Wayne, and Nicolas Heess. Learning human behaviors from motion capture by adversarial imitation. *CoRR*, abs/1707.02201, 2017. URL `http://arxiv.org/abs/1707.02201`.

Katharina Mülling, Jens Kober, Oliver Kroemer, and Jan Peters. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, 32(3): 263–279, 2013.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pp. 278–287, 1999.

Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. In *International Conference on Machine Learning*, pp. 2661–2670, 2017.

Peter Pastor, Heiko Hoffmann, Tamim Asfour, and Stefan Schaal. Learning and generalization of motor skills by learning from demonstration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pp. 763–768. IEEE, 2009.

Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel Van De Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 36(4):41, 2017.

Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing-solving sparse reward tasks from scratch. *arXiv preprint arXiv:1802.10567*, 2018.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pp. 1889–1897, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

Richard Stuart Sutton. Temporal credit assignment in reinforcement learning. 1984.

Richard Socher Tianmin Shu, Caiming Xiong. Hierarchical and interpretable skill acquisition in multi-task reinforcement learning. *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=SJJQVZW0b`.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 5026–5033. IEEE, 2012.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017.

Ziyu Wang, Josh S Merel, Scott E Reed, Nando de Freitas, Gregory Wayne, and Nicolas Heess. Robust imitation of diverse behaviors. In *Advances in Neural Information Processing Systems*, pp. 5326–5335, 2017.

Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. *arXiv preprint arXiv:1710.01813*, 2017.

A TRANSITION ANALYSIS

As illustrated in Figure 5, transition policies are capable of moving toward states with higher rewards from proximity predictors. Specifically, the learned transition policy moves between the states of the primitives and higher proximity predictor reward is given to states that are close to the initiation set of the corresponding primitive policy.

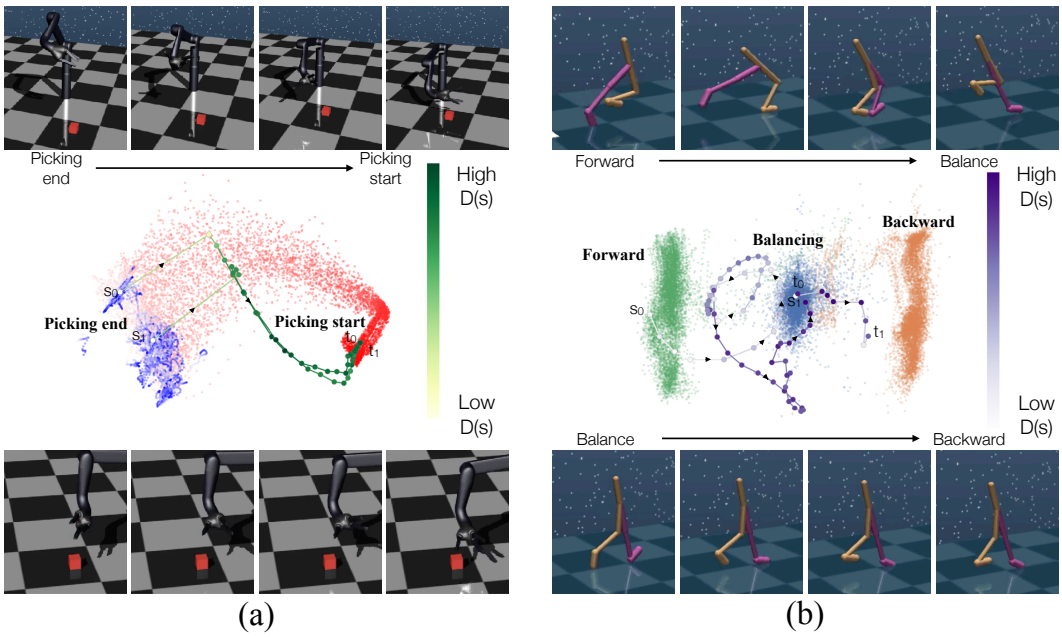


Figure 5: Visualized transitioning trajectories. (a) shows the transitioning trajectories sampled from the manipulation task *Repetitive Picking* and (b) shows the ones from the locomotion task *Patrol*. TOP AND BOTTOM ROWS: rendered frames of transition policies executing transitions. MIDDLE: contains states from each labeled primitive distribution projected onto PCA space. The dots connected with lines are extracted from the same transitioning trajectory, where the marker color indicates the proximity prediction of the proximity predictor $D(s)$. A higher $D(s)$ value indicates proximity to suitable states to initialize the next primitive. On the left, two transitions shown between repetitive picks starts at s_0 and s_1 near the end of the picking skill and transition to t_0 and t_1, respectively, near the start of the picking skill. Similarly, on the right, the forward to balance transition moves between the forward and balance state distributions and the balance to backward transition moves from the balancing states close to the backward states.

B ACQUIRING PRIMITIVE POLICIES

The modular framework proposed in this paper allows a primitive policy to be any of a pre-trained neural network, inverse kinematics module, or hard-coded policy. In this paper, we use neural networks trained with TRPO (Schulman et al., 2015) on dedicated environments as primitive policies (see section D for details on environments and reward functions). A policy network consists of 2 layers of 32 hidden units with tanh nonlinearities and predicts the mean and standard deviation of a Gaussian distribution over an action space. We trained all primitive policies until the total return converged (up to 10,000 iterations).

A primitive skill consists of a primitive policy and a termination signal. Given a state, a primitive policy outputs the next action and the termination signal indicates whether the execution is done and if the skill was successfully performed (see section D).

C TRAINING DETAILS

C.1 IMPLEMENTATION DETAILS

For the TRPO and PPO implementation, we used OpenAI baselines[1] with default hyperparameters including learning rate, KL penalty, and entropy coefficients unless specified below.

Hyperparameters	Transition policy	Proximity predictor	Primitive policy	TRPO	PPO
Learning rate	1e-4	1e-4	1e-3 (for critic)	1e-3 (for critic)	1e-4
# Mini-batch	150	150	32	150	150
Batch size	64	64	64	64	64
Learning rate decay	no	no	no	no	linear decay

Table 3: Hyperparameter values for transition policy, proximity predictor, and primitive policy as well as TRPO and PPO baselines. For all networks, we use the Adam optimizer with batch size of 64.

C.2 REPLAY BUFFERS

The success buffer $\mathcal{B}^S$ contains states and their proximity to the corresponding initiation set in successful transitions. We used $\delta = 0.97$ for all our experiments. On the other hand, the failure buffer $\mathcal{B}^F$ contains states in failure transitions. We implement buffers with a *deque* data structure. New items are added on one end and once a buffer is full, a corresponding number of items are discarded from the opposing end. For all experiments, we use buffers, $\mathcal{B}^S$ and $\mathcal{B}^F$, with a capacity of one million states.

For efficient training of the proximity predictors, we collect successful trajectories of primitive skills which can be sampled during the training of primitive skills. We run 1000 episodes for each primitive and put the first 10%-20% in trajectories into the success buffer as an initiation set. While initiation sets can be discovered via random exploration, we found that this initialization of success buffers improves the efficiency of training by providing initial training data for the proximity predictors.

C.3 THE PROXIMITY PREDICTOR

The proximity predictor takes a state as input which includes joint state information as well as any task specification, such as ceiling and curb information. An proximity predictor consists of 2 fully connected layers of 96 hidden units with ReLU nonlinearities and predicts the proximity to the initiation set based on samples from the success and failure buffers. Iterations consist of 10 epochs over a batch size of 64 and use a learning rate of 10^{-4}. The predictor optimizes the loss in Equation (1), similar to the LSGAN loss (Mao et al., 2017).

C.4 TRAINING TRANSITION POLICIES

A transition policy consists of 2 fully connected layers of 32 hidden units with tanh nonlinearities and predicts the mean and standard deviation of a Gaussian distribution over an action space. A 2-way softmax layer is followed by the last fully connected layer to predict whether to terminate the current transition or not. We train transition policy using PPO (Schulman et al., 2017). Every iteration consists of 5 epochs over batch size.

In this paper, we use neural networks trained with TRPO (Schulman et al., 2015) on dedicated environments as primitive policies (see section D for details on environments and reward functions). A policy network consists of 2 layers of 32 hidden units with tanh nonlinearities and predicts the mean and standard deviation of a Gaussian distribution over actions. For the TRPO and PPO implementation, we used OpenAI baselines[2] with default hyperparameters. We trained all primitive policies until the total return converged (up to 10,000 iterations).

[1] https://github.com/openai/baselines
[2] https://github.com/openai/baselines

C.5 THE PROXIMITY REWARD

Transition policies receive rewards based on the outputs of proximity predictors. Before computing the reward at every time step, we clip the output of the proximity predictor D by $\text{clip}(D(s), 0, 1)$ which indicates how close the state s is to the initiation set of the following primitive (higher values correspond to closer states). We define the proximity of a state to an initiation set as γ^{step}, where step is the shortest number of timesteps required to get to a state in the initiation set. We use $\gamma = 0.95$ for all experiments. To make the reward denser, for every timestep t, we provide the increase in proximity, $D(s_{t+1}) - D(s_t)$, as a reward for transition policy.

C.6 SCALABILITY

Each sub-policy requires its corresponding transition policy, proximity predictor, and two buffers. Hence, both the time and memory complexities of our method are linearly dependent on the number of sub-policies. The memory overhead is affordable since a transition policy (2 layers of 32 hidden units), a discriminator (2 layers of 96 hidden units), and buffers (1M states) are small.

D TASK DESCRIPTIONS

For every task, we add a control penalty, $-0.001 * \|a\|^2$, to regularize the magnitude of actions where a is a torque action performed by an agent. We note that all measures are in meters, and we omit the measures here for clarity of the presentation.

D.1 LOCOMOTION

A bi-pedal planar walker is used for simulating locomotion tasks. The observation consists of the position and velocity of the torso, joint angles, and angular velocities. The observation is represented as a 17-dim vector:
`(torso.y, torso.z, torso.angle, left_thigh.angle, left_leg.angle, left_foot.angle, right_thigh.angle, right_leg.angle, right_foot.angle, torso.x_vel, torso.angular_vel, left_thigh.angular_vel, left_leg.angular_vel, left_foot.angular_vel, right_thigh.angular_vel, right_leg.angular_vel, right_foot.angular_vel)`. The action space is torque control on the 6 joints.

D.1.1 REWARD DESIGN

Different locomotion tasks share many components of reward design, such as velocity, stability, and posture. We use the same form of reward functions, but with different hyperparameters for each task. The basic form of the reward function is as following:

$$\begin{aligned} R(s) =& \lambda_{vel} \cdot \text{abs}(v_x - v_{target}) + \lambda_{alive} - \lambda_{height} \cdot \text{abs}(1.1 - min(1.1, \Delta h)) + \\ & \lambda_{angle} \cdot \cos(angle) - \lambda_{foot}(v_{right_foot} + v_{left_foot}), \end{aligned}$$

where v_x, v_{right_foot}, and v_{left_foot} are forward velocity, right foot angular velocity, left foot angular velocity; and Δh and $angle$ are the distance between the foot and torso and the angle of the torso, respectively. The foot velocities help the agent to move its feet naturally. Δh and $angle$ are used to maintain height of the torso and encourage an upright pose.

Forward: The *Forward* task requires the walker agent to walk forward for 20 meters. To make the agent robust, we apply a random force with arbitrary magnitude and direction to a randomly selected joint every 10 timesteps.

$$\lambda_{vel} = 2, \lambda_{alive} = 1, \lambda_{height} = 2, \lambda_{angle} = 0.1, \lambda_{foot} = 0.01, \text{, and } v_{target} = 3$$

Backward: Similar to *Forward*, the *Backward* task requires the walker to walk backward for 20 meters under random forces.

$$\lambda_{vel} = 2, \lambda_{alive} = 1, \lambda_{height} = 2, \lambda_{angle} = 0.1, \lambda_{foot} = 0.01, \text{, and } v_{target} = -3$$

Balancing: In the *Balancing* task, the agent learns to balance under strong random forces for 1000 timesteps. Similar to other tasks, the random forces are applied to a random joint every 10 timesteps, but with magnitude 5 times larger.

$$\lambda_{vel} = 1, \lambda_{alive} = 1, \lambda_{height} = 0.5, \lambda_{angle} = 0.1, \lambda_{foot} = 0, \text{, and } v_{target} = 0$$

Crawling: In the *Crawling* task, a ceiling of height 1.0 and length 16 is located in front of the agent, and the agent is required to crawl under the ceiling without touching it. If the agent touches the ceiling, we terminate the episode. The task can be completed when the agent passes a point 1.5 after the ceiling and gets 100 additional reward.

$$\lambda_{vel} = 2, \lambda_{alive} = 1, \lambda_{height} = 0, \lambda_{angle} = 0.1, \lambda_{foot} = 0.01, \text{, and } v_{target} = 3$$

Jumping: In the *Jumping* task, a curb of height 0.4 and length 0.2 is located in front of the walker agent. The observation contains a distance to the curb in addition to the 17-dimensional joint information, where the distance is clipped by 3. The x location of the curb is randomly chosen from [2.5, 5.5]. In addition to the reward function above, it also gets an additional 100 reward for passing the curb and $200 \cdot v_y$ when the agent passes the front, middle, and end slices of the curb, where v_y is y-velocity. If the agent touches the curb, the agent gets -10 penalty and the episode is terminated.

$$\lambda_{vel} = 2, \lambda_{alive} = 1, \lambda_{height} = 2, \lambda_{angle} = 0.1, \lambda_{foot} = 0.01, \text{ and } v_{target} = 3$$

Patrol: The *Patrol* task is repetitive running forward and backward between two goals at $x = -2$ and $x = 2$. Once the agent touches a goal, the target is changed to another goal and the sparse reward +1 is given. The dense reward alternates between the reward functions of *Forward* and *Backward*. The agent gets the reward of *Forward* when the agent is heading toward $x = 2$ and gets the reward of *Backward*, otherwise.

Hurdle: The *Hurdle* environment consists of 5 curbs positioned at $x = \{8, 18, 28, 38, 48\}$ and requires repetitive walking and jumping behaviors. The position of each curb is randomized with a uniformly sampled value from $[-0.5, 0.5]$. The sparse reward +1 is given when the agent jumps over a curb (i.e. pass a point 1.5 after a curb). The dense reward for *Hurdle* has 8 reward components to guide the agent to learn the desired behavior. By extensively designing dense rewards, it is possible to solve complex tasks. In comparison, our proposed method learns from sparse reward by re-using prior knowledge and doesn't require reward shaping.

Obstacle Course: The *Obstacle Course* environment replaces two curbs in *Hurdle* with a ceiling of height 1.0 and length 3. The sparse reward +1 is given when the agent jumps over a curb or passes through a ceiling (i.e. pass a point 1.5 after a curb or a ceiling). The dense reward is alternating between *Jumping* before the curb and *Crawling* before the ceiling.

D.1.2 Termination signal

Locomotion tasks except *Crawling* fail if $h < 0.8$ and *Crawling* fails if $h < 0.3$. *Forward*, *Backward*, and *Balancing* tasks are considered successful when the agent does not fail for 50 timesteps. The agent succeeds on *Jumping* and *Crawling* if the agent passes the obstacles by a distance of 1.5.

D.2 Robotic manipulation

In object manipulation tasks, a 9-DOF Jaco robotic arm is used as an agent and a cube with the side length 0.06 m is used as a target object. We followed the tasks and environment settings proposed in Ghosh et al. (2018). The observation consists of the position of the base of the Jaco arm, joint angles, and angular velocities. The observation is represented as a 31-dim vector:
`(link1.angle, link2.angle, link3.angle, link4.angle, link5.angle, link6.angle, finger1.angle, finger2.angle, finger3.angle, link1.angular_vel, link2.angular_vel, link3.angular_vel, link4.angular_vel, link5.angular_vel, link6.angular_vel, finger1.angular_vel, finger2.angular_vel, finger3.angular_vel, box.x, box.y, box.z, box.x_vel, box.y_vel, box.z_vel, box.quat_i, box.quat_j, box.quat_k, box.quat_w, box.angular_vel_1, box.angular_vel_2, box.angular_vel_3).`

The action space is a torque control on 9 joints.

D.2.1 REWARD DESIGN AND TERMINATION CONDITION

Pick: In the pick task, the position of the box is randomly initialized within a square region of size 0.1 m $\times$ 0.1 m with a center (0.5, 0.2). There is an initial guide reward to guide the arm to the box. There is also an over reward to guide the hand directly over the box. When the arm is not picking up the box, there is a pick reward to incentivize the arm to pick the box up. There is an additional hold reward that makes the arm hold the box in place after picking up. Finally, there is a success reward given after the arm has held the box for 50 frames. The success reward is scaled with number of timesteps to encourage the arm to succeed as quickly as possible.

$$R(s) = \lambda_{guide} \cdot \mathbf{1}_{\text{Box not picked and Box on ground}} + \lambda_{pick} \cdot \mathbf{1}_{\text{Box in hand and not picked}} + \lambda_{hold} \cdot \mathbf{1}_{\text{Box picked and near hold point}}$$

$$\lambda_{guide} = 2, \lambda_{pick} = 100, \lambda_{hold} = 0.1$$

Catch: The position of the box is initialized at (0, 2.0, 1.5) and the directional force of size 110 is applied to throw the box toward the agent with randomness (0.1 m $\times$ 0.1 m).

$$R(s) = \mathbf{1}_{\text{Box in air and Box within 6 of Jaco end-effector}}$$

Toss: The box is randomly initialized on the ground at (0.4, 0.3, 0.05) within a 0.005 $\times$ 0.005 m square region. A guide reward is given to guide the arm to the top of the box. A pick reward is then given to lift the box up to a specified release height. A release reward is given if the box is no longer in the hand. A stable reward is given to minimize variation in the box's x and y direction. An up reward is given while the ball is traveling upwards in air, up until the box hits a specified z height. Finally, a success reward is given based on the landing position of the box and the specified landing position.

Hit: The box is randomly initialized overhead the arm at (0.4, 0.3, 1.2) within a 0.005 $\times$ 0.005 m square region. The box falls and the arm is given a hit reward for hitting the box. Once the box has been hit, a target reward is given based on how close the box is to the target.

Repetitive pick: The *repetitive pick* task has two reward variants. The sparse version gives a reward 1 for every successful pick. The dense reward version gives a guide reward to the box after each successful pick following the reward for the *pick* task.

Repetitive catch: The *repetitive catch* task gives a reward 1 for every successful catch. For dense reward, it uses the same reward function with that of the *catch* task.

Pick-Catch: The *pick catch* task gives a reward 1 for every successful pick or catch. In the dense reward setting, it provides the reward following the *pick* and *catch* according to the box position.

Serve: The *serve* task gives a toss reward for a successful toss and a target reward for successfully hitting the target. The dense reward setting provides the *toss* and *hit reward according to box position.*