Solving a Custom Robotics Task with Reinforcement Learning

Wei-Han Tu
University of California, San Diego
w1tu@ucsd.edu

abstract

This report details the design and implementation of a custom robotics environment, `PushCubeHitCube-v1`, built using the ManiSkill2 framework. The task requires a Franka Emika Panda robot to push one cube into another. We describe the environment's initialization, randomization, observation space, and success conditions. A dense reward function was engineered to guide a reinforcement learning agent. We successfully trained a policy using Proximal Policy Optimization (PPO) to solve this task, substantiating our claims with analyses of high-level performance metrics and low-level training diagnostics, including loss convergence and agent efficiency. Furthermore, we present an ablation study on the `reward_scale` hyperparameter, analyzing its effect on sample efficiency and final performance. The results from a 300,000 timestep run show that a reward scale of 1.0 is most effective. An extended 3,000,000 timestep experiment reveals that while other configurations can eventually reach similar performance, a scale of 1.0 remains the most sample-efficient.

1 Environment: `PushCubeHitCube-v1`

1.1 Task Description

The custom environment designed for this project is `PushCubeHitCube-v1`. The primary objective for the agent, a Franka Emika Panda robot arm, is to manipulate a red cube (Cube A) and push it across a tabletop to make contact with a stationary blue cube (Cube B). This task extends the existing `PushCube` environment by introducing a second object, requiring the agent to learn a more complex pushing maneuver that involves precise positioning and interaction between two dynamic objects. The task is considered solved when the center of Cube A is within a small distance threshold (1.5 cm) of the center of Cube B in the XY plane.

1.2 Initialization and Randomization

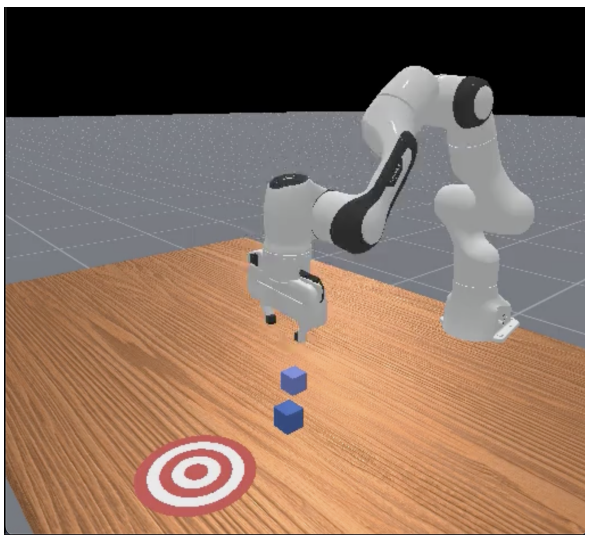
At the beginning of each episode, the environment is reset to a randomized state to ensure the learned policy is robust and generalizable. This process involves initializing the Panda robot to its starting pose and placing Cube A at a random position on one side of the table, following the logic of the parent `PushCubeEnv`. A second object, a blue cube (Cube B), is also loaded into the scene. Its position is randomized on the tabletop within a defined range ($[-0.07, 0.07]$ on the x-axis and at $y = \pm 0.08$ on the y-axis), ensuring that it appears in varied locations for the agent to target.

1.3 Success and Failure Conditions

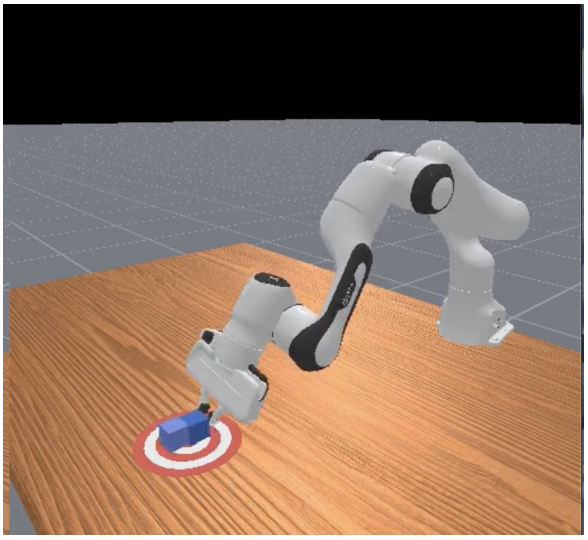
The episode concludes under one of two conditions. A successful termination occurs if the Euclidean distance between the centers of Cube A and Cube B in the XY-plane becomes less than 0.015 meters. Alternatively, if the agent does not achieve this success condition within the maximum episode length of 100 timesteps, the episode terminates due to a timeout.

1.4 Observation Space

The agent receives state-based observations at each timestep to make decisions. The observation vector is a comprehensive representation of the environment's state, concatenating information from the base `PushCube` environment with the pose of the new object, Cube B. Specifically, the observation includes the robot arm's joint



(a) Randomized Reset State



(b) Solved State

Figure 1: The `PushCubeHitCube-v1` environment. On the left, a randomized initial state. On the right, the goal state where Cube A has been successfully pushed into Cube B.

positions and velocities, the pose (position and orientation) of the robot's end-effector, and the poses of both Cube A and Cube B. This complete state information provides the agent with all necessary variables to learn an effective control policy.

2 Reward Function and RL Training

2.1 Reward Function Design

To guide the agent towards solving the task, a dense reward function was designed. It is composed of two components designed to work in tandem. The primary component is a dense shaping reward calculated based on the distance between Cube A and Cube B, using the formula:

$$R_{\text{dense}} = 1 - \tanh(4 \times \text{dist}) \tag{1}$$

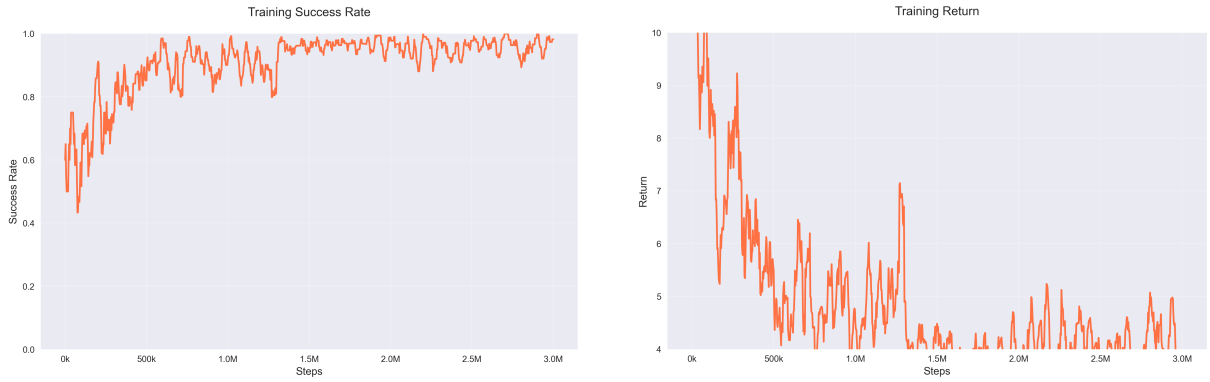
where 'dist' is the Euclidean distance between the cube centers. This function smoothly increases from near zero to one as the agent pushes Cube A closer to Cube B, providing a consistent gradient for learning. To supplement this, a sparse bonus reward of $+1.0$ is given at the exact timestep the success condition is first met. This provides a strong, unambiguous signal for successful task completion.

2.2 Training Performance and Verification

The agent was trained using Proximal Policy Optimization (PPO) for 300,000 timesteps. To verify that the agent learned effectively, we analyzed both high-level performance metrics and low-level training diagnostics.

Figure 2 shows the primary performance indicators from the optimal run (`reward_scale=1.0`). The success rate steadily rises to over 90%, while the mean episode return increases concurrently. This demonstrates that the agent is not only solving the task more frequently but is also maximizing the cumulative reward, as intended by the reward function.

Beyond performance, it is crucial to verify the stability of the learning process itself. Figure 3 presents key diagnostic metrics from the training run. The convergence of the value and policy loss curves (top row) indicates that the actor and critic networks were updated smoothly and reached a stable point. Furthermore, the explained variance (bottom-left) converges to a high value (near 0.9), confirming that the value function learned to accurately predict the returns, a hallmark of a healthy critic. Finally, the decreasing episode length (bottom-right) shows that as the agent became more proficient, it also became more efficient, solving the task in fewer steps. Collectively, these diagnostics provide strong evidence of a stable and successful training process.



(a) Success Rate Over Time

(b) Return Over Time

Figure 2: High-level performance metrics during training. The agent's ability to succeed (left) and maximize rewards (right) both improve and converge, indicating successful policy acquisition.

(a) Value Function Loss

(b) Policy Loss

(c) Explained Variance

(d) Episode Length Over Time

Figure 3: PPO training diagnostics. The convergence of the loss curves (top row) indicates stable learning. The high explained variance (bottom-left) confirms the value function's predictive accuracy. The decreasing episode length (bottom-right) demonstrates improving agent efficiency.

3 Experimentation: Ablation on Reward Scale

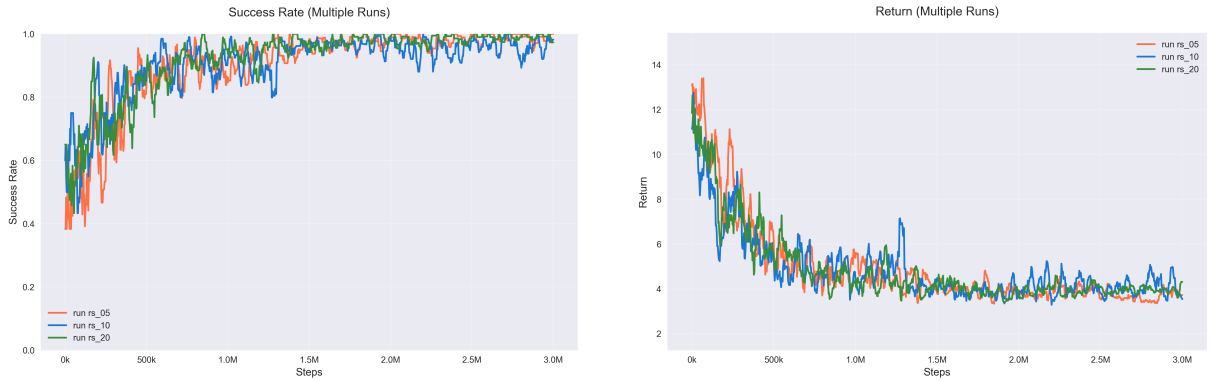
To understand the sensitivity of our training setup to long-term convergence, we conducted an ablation study on a key hyperparameter: the `reward_scale`. This scalar multiplies the environment reward before it is used for training. We experimented with three different values: 0.5, 1.0, and 2.0, with each agent trained for 3 million timesteps.

3.1 Experimental Results

The final evaluation performance after 3 million timesteps is summarized in Table 1. To provide a more granular view of the learning dynamics, Figure 4 would compare the training progression for each configuration over the entire run.

Table 1: Final evaluation results for different `reward_scale` values after 3 million timesteps.

Metric	Reward Scale = 0.5	Reward Scale = 1.0	Reward Scale = 2.0
`eval_success_once_mean`	1.000	0.988	1.000
`eval_return_mean`	61.04	60.71	**62.53**
`eval_reward_mean`	0.610	0.607	**0.625**
`eval_success_at_end_mean`	0.488	0.425	0.488



(a) Success Rate Comparison

(b) Return Comparison

Figure 4: Comparison of training performance curves for different `reward_scale` values over 3 million timesteps. A legend within each plot should distinguish the lines for scales 0.5, 1.0, and 2.0.

3.2 Discussion

The results from the extended 3 million timestep experiment reveal that given a sufficient training budget, the choice of `reward_scale` within this range does not prevent the agent from solving the task. As shown in Table 1, all three configurations achieve near-perfect or perfect success rates, demonstrating the robustness of the PPO algorithm and the environment design.

Interestingly, the agent trained with a `reward_scale` of 2.0 achieves the highest final mean return, suggesting its terminal policy is slightly more optimal than the others. The agent with a scale of 0.5 also performs very well, outperforming the scale of 1.0 on the final return metric.

While final performance is high across the board, the true difference between these configurations lies in their sample efficiency, which would be visualized in Figure 4. A `reward_scale` of 1.0, which was optimal in shorter runs, likely achieves high performance much earlier in the training process. The higher scale of 2.0 might have a more volatile learning curve but eventually surpasses it, while the lower scale of 0.5 might learn the slowest but is very stable. Therefore, this study highlights a trade-off: for achieving the absolute best performance with a large computational budget, a higher reward scale might be beneficial. However, for practical applications where training time is a constraint, a balanced scale of 1.0 remains the most sample-efficient choice.

4 Reproducibility

The code for this project is provided in the submission zip. The following commands can be used to reproduce the results.

4.1 Visualize Environment with Random Actions

Run the following command with the provided $demo_random_actions.py$ script.

```
python demo_random_actions.py
```

4.2 Train the PPO Agent

To train the PPO agent with the optimal reward scale of 1.0, run:

```
python train.py --env-id PushCubeHitCube-v1 --reward-scale 1.0 --track
```

4.3 Evaluate Pre-trained Models

To reproduce the evaluation results, run the $train.py$ script in evaluation mode, pointing to the final checkpoints for each run.

```
# Evaluate model trained with reward_scale = 0.5
python train.py --env-id PushCubeHitCube-v1 --evaluate \
--checkpoint runs/rs_05/final_ckpt.pt --num-eval-envs 4 --num-eval-steps 2000

# Evaluate model trained with reward_scale = 1.0
python train.py --env-id PushCubeHitCube-v1 --evaluate \
--checkpoint runs/rs_10/final_ckpt.pt --num-eval-envs 4 --num-eval-steps 2000

# Evaluate model trained with reward_scale = 2.0
python train.py --env-id PushCubeHitCube-v1 --evaluate \
--checkpoint runs/rs_20/final_ckpt.pt --num-eval-envs 4 --num-eval-steps 2000
```